
Electronic Thesis and Dissertation Repository

11-8-2016 12:00 AM

Towards Comprehensive Parametric Code Generation Targeting Graphics Processing Units in Support of Scientific Computation

Ning Xie

The University of Western Ontario

Supervisor

Marc Moreno Maza

The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

© Ning Xie 2016

Follow this and additional works at: <https://ir.lib.uwo.ca/etd>



Part of the [Numerical Analysis and Scientific Computing Commons](#), and the [Programming Languages and Compilers Commons](#)

Recommended Citation

Xie, Ning, "Towards Comprehensive Parametric Code Generation Targeting Graphics Processing Units in Support of Scientific Computation" (2016). *Electronic Thesis and Dissertation Repository*. 4257.

<https://ir.lib.uwo.ca/etd/4257>

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlsadmin@uwo.ca.

Abstract

The most popular multithreaded languages based on the fork-join concurrency model (CilkPlus, OpenMP) are currently being extended to support other forms of parallelism (vectorization, pipelining and single-instruction-multiple-data (SIMD)). In the SIMD case, the objective is to execute the corresponding code on a many-core device, like a GPGPU, for which the CUDA language is a natural choice. Since the programming concepts of CilkPlus and OpenMP are very different from those of CUDA, it is desirable to automatically generate optimized CUDA-like code from CilkPlus or OpenMP.

In this thesis, we propose an accelerator model for annotated C/C++ code together with an implementation that allows the automatic generation of CUDA code. One of the key features of this CUDA code generator is that it supports the generation of CUDA kernel code where program parameters (like number of threads per block) and machine parameters (like shared memory size) are treated as unknown symbols. Hence, these parameters need not to be known at code-generation-time: machine parameters and program parameters can be respectively determined when the generated code is installed on the target machine.

In addition, we show how these parametric CUDA programs can be optimized at compile-time in the form of a case discussion, where cases depend on the values of machine parameters (e.g. hardware resource limits) and program parameters (e.g. dimension sizes of thread-blocks).

This generation of parametric CUDA kernels requires to deal with non-linear polynomial expressions during the dependence analysis and tiling phase of the input code. To achieve these algebraic calculations, we take advantage of techniques from computer algebra, in particular in the `RegularChains` library of `MAPLE`. Various illustrative examples are provided together with performance evaluation. Our preliminary implementation uses `LLVM`, `MAPLE` and `PPCG`; moreover, it successfully processes a variety of standard test-examples.

Keywords: Many-core machine model; Parametric CUDA code generation; Polynomial arithmetic; Compiler optimization

Acknowledgements

With your greatest help, I become who I am today. With your enlightenment, I achieve what I have today.

Special thanks sincerely for the insightful guidance from Prof. Marc Moreno Maza and the great collaboration with Prof. Robert M. Corless, Dr. Changbo Chen, Dr. Yuzhen Xie, Dr. Sardar A. Haque, Svyatoslav Covanov, Farnam Mansouri, Robert H.C. Moir and Xiaohui Chen. Many thanks to industry partners: Dr. Jürgen Gerhard from Maplesoft R&D department, and Wang Chen, Abdoul-Kader Keita and Jeeva Paudel from IBM compiler group for making this research work have practical uses.

Big thanks to the supervisory committees: Prof. Mark Daley and Prof. Yuri Boykov for advices and comments. It is my honor to have Dr. Matteo Frigo, Prof. Robert M. Corless, Prof. Michael Bauer and Prof. Roberto Solis-Oba as the examiners. I would like to express my gratitude for their comments and questions. With all my heart, I appreciate the continued support of my parents, colleagues, friends, the ORCCA laboratory and the Computer Science Department of The University of Western Ontario.

The work was supported by NSERC of Canada, MITACS, Maplesoft Inc. and IBM Corp..

Dream my life, live my dream. To a better self.

Contents

Abstract	i
Acknowledgements	ii
List of Figures	vi
List of Tables	x
List of Algorithms	xii
List of Appendices	xiii
1 Introduction	1
1.1 Contributions of this thesis	5
1.2 Outline of this thesis	6
2 Background	10
2.1 Models of computation	10
2.1.1 Fork-join model	10
2.1.2 PRAM model	11
2.2 General-purpose GPU computing	13
2.2.1 The <i>Compute Unified Device Architecture</i> (CUDA)	14
2.2.2 Modern GPU architectures	15
2.3 Dense arithmetic over finite fields with the CUMODP library	16
2.4 The MetaFork language	17
2.5 Automatic parallelization in the polyhedral model	19
2.5.1 \mathbb{Z} -polyhedron	20
2.5.2 Polyhedral iteration domain	21
2.5.3 Data dependence graph	21
2.5.4 Dependence polyhedron	22
2.5.5 Affine transformation	23
2.5.6 Farkas multipliers	24
2.5.7 Feautrier’s algorithm	24
2.5.8 PLUTO’s algorithm	25
2.6 Solving systems of polynomial equations and inequalities	26
3 The Basic Polynomial Algebra Subprograms	30

3.1	Design and specification	30
3.2	User interface	31
3.3	Implementation techniques	34
3.4	Experimental evaluation	35
3.5	Application	37
4	A Many-Core Machine Model	41
4.1	Introduction	41
4.2	A many-core machine model	43
4.2.1	Characteristics of the abstract many-core machines	43
4.2.2	Many-core machine programs	46
4.2.3	Complexity measures for the many-core machine model	46
4.2.4	A Graham-Brent theorem with parallelism overhead	47
4.3	The Euclidean algorithm	48
4.4	Fast Fourier Transform	52
4.4.1	Cooley & Tukey algorithm	52
4.4.2	Stockham algorithm	53
4.4.3	Comparison of running time estimates	54
4.5	Polynomial multiplication	55
4.5.1	Plain multiplication	55
4.5.2	FFT-based multiplication	58
4.5.3	Comparison of running time estimates	59
4.6	Radix sort	60
4.7	Conclusion	62
5	MetaFork-to-CUDA: Generation of Parametric CUDA Kernels	63
5.1	Optimizing CUDA kernels depending on program parameters	64
5.2	Automatic parametric CUDA kernel generation	65
5.3	The MetaFork-to-CUDA code generator	67
5.4	Experimentation	73
5.5	Conclusion	80
6	Generation of Optimized CUDA Kernel Code	84
6.1	Case study: matrix multiplication	84
6.2	Experimentation	90
6.3	Conclusion	94
7	Towards Comprehensive Parametric CUDA Kernel Generation	96
7.1	Comprehensive optimization	99
7.1.1	Hypotheses on the input code fragment	100
7.1.2	Hardware resource limits and performance measures	100
7.1.3	Evaluation of resource and performance counters	102
7.1.4	Optimization strategies	102
7.1.5	Comprehensive optimization	103
7.1.6	Data-structures	104

7.1.7	The algorithm	104
7.2	Comprehensive translation of an annotated C program into CUDA kernels	109
7.2.1	Input MetaFork code fragment	110
7.2.2	Comprehensive translation into parametric CUDA kernels	111
7.3	Implementation details	111
7.4	Experimentation	113
7.5	Conclusion	121
8	Conclusion and Future Work	123
	Bibliography	125
A	Sample Code in the BPAS Library	135
A.1	Adaptive algorithms	135
A.2	User interfaces	136
B	Theoretical Analysis of Fundamental Algorithms Using the MCM Model	139
C	Documentation for MetaFork-to-CUDA Code Generator	181
C.1	Assumptions on the syntax of MetaFork statements	181
C.2	Schedule tree for MetaFork and parametric CUDA code	183
D	Examples Generated by PPCG	188
E	The Implementation for Generating Comprehensive MetaFork Programs	193
	Curriculum Vitae	196

List of Figures

1.1	A C program for reversing a one-dimensional array	2
1.2	The CUDA program for reversing a one-dimensional array	3
1.3	Parametric CUDA kernel for reversing a one-dimensional array	4
1.4	Two CUDA kernels based on possible values of machine and program parameters for reversing a one-dimensional array, where Z is the maximum number of shared memory words per processor supported by the hardware architecture and R is the maximum number of registers per thread supported by the hardware architecture	6
1.5	Overview of the thesis	8
2.1	An example of computation DAG: Fourth Fibonacci	11
2.2	Matrix multiplication written in CilkPlus	12
2.3	Overview of a hybrid CPU-GPU system	13
2.4	Execution of a CUDA program	14
2.5	Using MetaFork to translate a given CilkPlus program into a OpenMP program	18
2.6	Using MetaFork to translate a given OpenMP program into a CilkPlus program	18
2.7	Using MetaFork to translate a given OpenMP program into a CilkPlus program	19
2.8	For-loop nest in the polyhedral model	20
2.9	An example of the data dependence graph of the source program	22
2.10	The transformed code based on time and processor coordinates	24
2.11	A triangular decomposition into semi-algebraic systems computed with the RealTriangularize command	28
2.12	Output of the RealTriangularize command for the <i>EVE</i> surface	28
3.1	A subset of BPAS algebraic data structures	32
3.2	Another subset of BPAS algebraic data structures	33
3.3	A snapshot of BPAS code	33
3.4	Multiplication scheme for dense univariate integer polynomials	35
3.5	Dense integer polynomial multiplication: BPAS vs FLINT vs MAPLE	36
3.6	The htop screenshot of multiplying two large integer polynomials in BPAS . . .	37
3.7	An example of matchable interval lists	39
3.8	A sample output of realSymbolicNumericIntegrate	40
4.1	Overview of an abstract many-core machine	44
4.2	Overview of a many-core machine program	44
4.3	An example of a thread-block DAG	48

4.4	Illustration of reads and writes by a thread-block in either ping-ping or ping-pong phase of the Euclidean algorithm	50
4.5	Running time on GeForce GTX 670 of our multithreaded Euclidean algorithm for univariate polynomials of sizes n and m over $\mathbb{Z}/p\mathbb{Z}$, where p is a 30-bit prime, whereas the program parameter takes values $s = 1$ and $s = 256$	52
4.6	Multiplication phase: illustration of a thread-block reading coefficients from a , b and writing to the auxiliary array M	57
4.7	Addition phase: illustration of a thread-block reading and writing to the auxiliary array M	57
4.8	Running time of the plain polynomial multiplication algorithm with polynomials a ($\deg(a) = n - 1$) and b ($\deg(b) = m - 1$) and the parameter s on GeForce GTX 670	59
4.9	Running time of the plain and FFT-based multiplication algorithms with the input size n on GeForce GTX 670	60
5.1	Overview of the implementation of the MetaFork-to-CUDA code generator	69
5.2	One-dimensional stencil computation, namely Jacobi, written in C	70
5.3	1D Jacobi written in MetaFork	70
5.4	Generated parametric CUDA kernel for 1D Jacobi	71
5.5	Generated host code for 1D Jacobi	72
5.6	Serial code, MetaFork code and generated parametric CUDA kernel for array reversal	74
5.7	Serial code, MetaFork code and generated parametric CUDA kernel for 2D Jacobi	76
5.8	Serial code, MetaFork code and generated parametric CUDA kernel for LU decomposition	78
5.9	Serial code, MetaFork code and generated parametric CUDA kernel for matrix transpose	79
5.10	Serial code, MetaFork code and generated parametric CUDA kernel for matrix addition	80
5.11	Serial code, MetaFork code and generated parametric CUDA kernel for matrix vector multiplication	81
5.12	Post-processing CUDA kernel with coalesced accesses for matrix vector multiplication	82
5.13	Serial code, MetaFork code and generated parametric CUDA kernel for matrix matrix multiplication	83
6.1	Multiplication of two matrices	84
6.2	The MetaFork code with the granularity loop and good data alignment for matrix multiplication	85
6.3	Post-processing the generated CUDA kernel code for matrix multiplication with the granularity loop	87
6.4	CUDA kernel with unrolling the granularity loop for matrix multiplication	89
6.5	The serial C code with good data locality for matrix multiplication	90
6.6	The MetaFork code and its kernel code with the granularity loop for reversing a one-dimensional array	92

6.7	The MetaFork code and its kernel code with the granularity loop for 1D Jacobi	93
6.8	The MetaFork code and its kernel code with the granularity loop for matrix addition	93
6.9	The MetaFork code and its kernel code with the granularity loop for matrix transpose	94
6.10	The MetaFork code and its kernel code with the granularity loop for matrix vector multiplication	95
7.1	Matrix addition written in C (the left-hand portion) and in MetaFork (the right-hand portion) with a <code>meta_for</code> loop nest, respectively	97
7.2	Comprehensive translation of MetaFork code to two kernels for matrix addition	98
7.3	The decision tree for comprehensive parametric CUDA kernels of matrix addition	99
7.4	Matrix vector multiplication written in C (the left-hand portion) and in MetaFork (the right-hand portion), respectively	102
7.5	The decision subtree for resource or performance counters	108
7.6	The serial elision of the MetaFork program for matrix vector multiplication . .	111
7.7	The software tools involved for the implementation	112
7.8	Computing the amount of words required per thread-block for reversing a 1D array	113
7.9	The first case of the optimized MetaFork code for array reversal	114
7.10	The second case of the optimized MetaFork code for array reversal	115
7.11	The third case of the optimized MetaFork code for array reversal	115
7.12	The first case of the optimized MetaFork code for matrix vector multiplication	115
7.13	The second case of the optimized MetaFork code for matrix vector multiplication	116
7.14	The third case of the optimized MetaFork code for matrix vector multiplication	116
7.15	The MetaFork source code for 1D Jacobi	117
7.16	The first case of the optimized MetaFork code for 1D Jacobi	117
7.17	The second case of the optimized MetaFork code for 1D Jacobi	118
7.18	The third case of the optimized MetaFork code for 1D Jacobi	118
7.19	The first case of the optimized MetaFork code for matrix addition	119
7.20	The second case of the optimized MetaFork code for matrix addition	119
7.21	The third case of the optimized MetaFork code for matrix addition	119
7.22	The first case of the optimized MetaFork code for matrix transpose	120
7.23	The second case of the optimized MetaFork code for matrix transpose	120
7.24	The third case of the optimized MetaFork code for matrix transpose	120
7.25	The first case of the optimized MetaFork code for matrix matrix multiplication	121
7.26	The second case of the optimized MetaFork code for matrix matrix multiplication	121
7.27	The third case of the optimized MetaFork code for matrix matrix multiplication	122
C.1	An example of the <code>meta_schedule</code> statement	182
D.1	PPCG code and generated CUDA kernel for array reversal	188
D.2	PPCG code and generated CUDA kernel for matrix addition	188
D.3	PPCG code and generated CUDA kernel for 1D Jacobi	189
D.4	PPCG code and generated CUDA kernel for 2D Jacobi	189

D.5	PPCG code and generated CUDA kernel for LU decomposition	190
D.6	PPCG code and generated CUDA kernel for matrix vector multiplication	191
D.7	PPCG code and generated CUDA kernel for matrix transpose	191
D.8	PPCG code and generated CUDA kernel for matrix matrix multiplication	192

List of Tables

3.1	One-dimensional modular FFTs: <code>Modpn</code> vs <code>BPAS</code>	34
3.2	Cilkview analysis of <code>BPAS</code> and <code>KS</code> (* shows the number of instructions)	37
3.3	Univariate real root isolation running times (in secs.) for four examples	38
3.4	Running times (in secs.) of multivariate real root isolation: <code>BPAS</code> vs <code>MAPLE 17</code> <code>RealRootIsolate</code> vs <code>C</code> (with <code>MAPLE 17</code> interface) <code>Isolate</code>	40
4.1	Running time (in secs) of the Cooley & Tukey and Stockham FFT algorithms with the input size n on GeForce GTX 670	55
5.1	Speedup comparison of reversing a one-dimensional array between PPCG and <code>MetaFork</code> kernel code	74
5.2	Speedup comparison of 1D Jacobi between PPCG and <code>MetaFork</code> kernel code . .	75
5.3	Speedup comparison of 2D Jacobi between PPCG and <code>MetaFork</code> kernel code . .	75
5.4	Speedup comparison of LU decomposition between PPCG and <code>MetaFork</code> ker- nel code	77
5.5	Speedup comparison of matrix transpose between PPCG and <code>MetaFork</code> kernel code	77
5.6	Speedup comparison of matrix addition between PPCG and <code>MetaFork</code> kernel code	79
5.7	Speedup comparison of matrix vector multiplication among PPCG kernel code, <code>MetaFork</code> kernel code and <code>MetaFork</code> kernel code with post-processing	80
5.8	Speedup comparison of matrix multiplication between PPCG and <code>MetaFork</code> kernel code	81
5.9	Timings (in sec.) of quantifier elimination for eight examples	82
6.1	Experimental results of matrix multiplication for the CUDA kernel with the shared memory for the output matrix and the granularity of threads	86
6.2	Experimental results of matrix multiplication for the CUDA kernel with the local memory for the output matrix and the granularity of threads	88
6.3	For input matrices of order 2^{10} , speedup factors of the matrix multiplication kernel unrolling the computation	88
6.4	For input matrices of order 2^{10} , speedup factors of the matrix multiplication kernel unrolling the copy-in, computation and copy-out phases with a compi- lation flag <code>--maxrregcount=40</code>	90
6.5	Speedup factors obtained with kernels generated by PPCG and <code>MetaFork</code> with post-processing, respectively, w.r.t. the serial C code with good data locality for matrix multiplicationm	91

6.6	Speedup factors of reversing a one-dimensional array for input vector of length 2^{25}	91
6.7	Speedup factors of 1D Jacobi for time iteration 4 and input vector of length $2^{15}+2$	92
6.8	Speedup factors of matrix addition for input matrix of order 2^{12}	92
6.9	Speedup factors of matrix transpose for input matrix of order 2^{14}	94
6.10	Speedup factors of matrix vector multiplication for input matrix of order 2^{13} and input vector of length 2^{13} (An error indicates that the total amount of re- quired shared memory exceeds the hardware limit.)	94
7.1	Optimization strategies with their codes	114

List of Algorithms

1	OptGcdKer(a, b, s, da, db)	49
2	PlainMultiplicationGPU(a, b, s)	56
3	MulKer(a, b, M, n, m, s)	56
4	AddKer(M, f, y, s, x, i)	57
5	ComprehensiveOptimization ($Q(S)$)	105
6	Optimize	106
7	MultiParametricCodeOptimizer($fileName$)	193
8	Optimize($plan, task$)	194
9	Optimize($plan, task$)	195

List of Appendices

Appendix A Sample Code in the BPAS Library	135
Appendix B Theoretical Analysis of Fundamental Algorithms Using the MCM Model . .	139
Appendix C Documentation for MetaFork-to-CUDA Code Generator	181
Appendix D Examples Generated by PPCG	188
Appendix E The Implementation for Generating Comprehensive MetaFork Programs . .	193

Chapter 1

Introduction

It is well known that the impact of the high-performance software for both numerical and exact linear algebra on engineering and scientific computing is tremendous, worthy of the great efforts that have been put for more than thirty years. The most significant results are software projects like BLAS [76], LAPACK [3], ATLAS [121] and LinBox [43]. The same remark should be extended to the processing of linear transforms, in particular, Fast Fourier Transforms, where notable works are the SPIRAL [98] and FFTW [48] projects, as well as to computer algebra, see the recent proceedings of the international workshop on parallel symbolic computation (PASCO) [44, 88, 89].

The successful techniques employed in those software projects include implementation of highly efficient algorithms for basic routines, block-based algorithms for better exploiting the memory hierarchies, parallel and distributed processing, as well as automatically optimizing and tuning code on different platforms. This trend has been stimulated by the advent of hardware acceleration technologies (multicore processors, cell processors, general-purpose graphics processing units (GPGPUs), field programmable gate arrays (FPGAs)) provide vast opportunities for innovation in computing. In particular, GPGPUs combined with *low-level heterogeneous programming models*, such as CUDA (the *Compute Unified Device Architecture*, see [95, 73]), brought super-computing to the level of the desktop computer.

However, these low-level programming models carry notable challenges, even to expert programmers. Indeed, fully exploiting the power of hardware accelerators by writing CUDA code often requires significant code optimization efforts. While such efforts can yield high performance, it is desirable for many programmers to avoid the explicit management of the hardware accelerator, e.g. data transfer between the host (or CPU) and the device (or GPGPU) or between memory levels of the device. For this reason, the most popular multithreaded languages, in particular CilkPlus [10, 79] and OpenMP [41, 13, 9], are based on the fork-join concurrency model targeting multi-core architectures rather than GPGPUs. Currently, these multithreaded languages are being extended to support other forms of parallelism, such as vectorization, pipe-lining and single-instruction-multiple-data (SIMD). In the SIMD case, the case we are interested in this thesis, the objective is to execute the corresponding code on a many-core GPGPU, for which the CUDA language is a natural choice.

To overcome the challenge of developing software targeting many-core GPGPUs, *high-level* models for accelerator programming, notably OpenMP and OpenACC [113, 56], have become an important research direction. With these models, programmers only need to annotate

their C/C++ (or FORTRAN) code to indicate which portion of code is to be executed on the device, and how data is mapped between the host and the device.

In OpenMP and OpenACC, the work distributed among the processors of the device can be expressed in a loose manner or even ignored. This implies that code optimization techniques must be applied in order to derive efficient CUDA code. Moreover, existing software packages (e.g. PPCG [115], C-to-CUDA [6], hiCUDA [57], CUDA-CHiLL [101]) for generating CUDA code from annotated C/C++ programs, either let the user choose, or make assumptions on the characteristics of the targeted hardware and on how the work is divided among the processors of that device. These choices and assumptions limit *code portability* as well as opportunities for *code optimization*. This latter fact will be illustrated with the following example for reversing a one-dimensional array.

Example 1 Consider the C program on Figure 1.1 for reversing a one-dimensional array, where each element of the input vector `In` is moved to the appropriate position of the output vector `Out`. Figure 1.2 gives a corresponding CUDA program, where the host code, shown on Figure 1.2(a), launches a kernel on the device, shown on Figure 1.2(b). Note that the kernel code makes a few assumptions:

- (1) The array length `N` is less than the maximum number of threads that the hardware architecture supports.
- (2) 32 divides `N`.

In order to launch a kernel function, one must specify the grid and thread-block formats (introduced in Section 2.2). In our example of Figure 1.2 32 threads per thread-block and `N/32` thread-blocks per grid are specified for `kernel0`. Observe also that the program explicitly handles the data placement on the device, for instance, on Figure 1.2, allocating shared memory for array `In` and using the global memory for array `Out`.

```
int N, In[N], Out[N];

// Initializing
for (int i = 0; i < N; i++)
    In[i] = i+1;

// Reversing the array In
for(int i = 0; i < N; i++)
    Out[N-1-i] = In[i];
```

Figure 1.1: A C program for reversing a one-dimensional array

The C-to-CUDA code generation introduced in [6] is known to be the first source-to-source polyhedral framework (see Section 2.5 for the term *polyhedral*) that translates serial C programs into CUDA programs. However, manual post-processing, such as placing synchronization statements in the kernel code, is required for generating a compilable CUDA kernel.

PPCG [115] automatically generates CUDA code from a given serial C program. In particular, Feautrier’s and PLUTO’s algorithms, respectively presented in [46] and [15], are adopted by the PPCG designers for exploring parallelization opportunities in serial programs.


```

int N, In[N], Out[N];

// Initializing
for (int i = 0; i < N; i++)
    In[i] = i+1;

int *dev_In, *dev_Out;

// Allocating memory spaces on the device
cudaMalloc((void **) &dev_In, (N)*sizeof(int));
cudaMalloc((void **) &dev_Out, (N)*sizeof(int));

// Copying the data from host to device
cudaMemcpy(dev_In, In, (N)*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_Out, Out, (N)*sizeof(int), cudaMemcpyHostToDevice);

// Launching the kernel
dim3 dimBlock(32);
dim3 dimGrid(N/32);
kernel0 <<<dimGrid, dimBlock>>> (dev_In, dev_Out, N);

// Copying the data from device to host
cudaMemcpy(Out, dev_Out, (N)*sizeof(int), cudaMemcpyDeviceToHost);

// Freeing the memory spaces on the device
cudaFree(dev_In);
cudaFree(dev_Out);

```

(a) The host code

```

__global__ void kernel0(int *In, int *Out, int N) {
    int idx = blockIdx.x * 32 + threadIdx.x;
    __shared__ int shared_In[32];

    if (idx < N) {
        shared_In[threadIdx.x] = In[idx];
        __syncthreads();
        Out[N-1-idx] = shared_In[threadIdx.x];
    }
}

```

(b) The device code

Figure 1.2: The CUDA program for reversing a one-dimensional array

However, by default, PPCG uses 32 (resp. 16×32) as the thread-block format in the generated one-dimensional (resp. two-dimensional) kernels; meanwhile, PPCG also allows the user to pass a numerical value as the thread-block format.

hiCUDA (acronym for high-level CUDA) [57] is defined as a directive-based language for generating CUDA programs from C programs. To be precise, hiCUDA relies on the `pragma` mechanism provided by the C/C++ standards to indicate how the work is distributed among

threads and how the data is mapped on the device. However, the grid and thread-block formats of a kernel must be specified as integer expressions.

CUDA-CHiLL [101] presents a script-based compiler framework for transforming annotated C programs to CUDA programs. Moreover, in [72], a transformation strategy generator is introduced with CUDA-CHiLL, such that several candidate CUDA kernels are generated, by varying data placement (shared memory or register) and numerical values of parameters (like thread-block sizes). Then, empirical evaluation of those candidate CUDA kernels is performed so as to select the CUDA kernel with the best performance by the end.

In summary, for those CUDA kernels generated by PPCG, C-to-CUDA, hiCUDA and CUDA-CHiLL, the program parameters (e.g. the number of threads per thread-block) and the machine parameters (e.g. the shared memory size) are numerical values instead of unknown symbols.

In this thesis, we propose an accelerator model for annotated C/C++ code, together with an implementation, that allows the automatic generation of CUDA code. One of the key features of this CUDA code generator is that it supports the generation of CUDA kernel code where program parameters (like number of threads per block) and machine parameters (like shared memory size) are treated as unknown symbols. Thus, machine parameters can be specialized to actual values when the generated CUDA code is compiled on the targeted hardware. The program parameters can be either optimized by techniques developed in this thesis or automatically tuned at run-time. As an illustration, following up on Example 1, a CUDA kernel code depending on one program parameter, along with a kernel function call from the host, is shown in Figure 1.3 where the variable `B` specifies the thread-block format. Observe that `kernel1` takes the program parameter `B` as an argument, whereas the `kernel0` in Figure 1.2 takes data parameters `In`, `Out` and `N` only.

```
__global__ void kernel1(int *In, int *Out, int N, int B) {
    int idx = blockIdx.x * B + threadIdx.x;
    // BLOCK_0 should be pre-defined as a constant
    // and be equal to B
    __shared__ int shared_In[BLOCK_0];

    if (idx < N) {
        shared_In[threadIdx.x] = In[idx];
        __syncthreads();
        Out[N-1-idx] = shared_In[threadIdx.x];
    }
}

// The kernel function call from the host
dim3 dimBlock(B);
dim3 dimGrid(N/B);
kernel1 <<<dimGrid, dimBlock>>> (dev_In, dev_Out, N, B);
```

Figure 1.3: Parametric CUDA kernel for reversing a one-dimensional array

Furthermore, given machine parameters represented as unknown symbols as well, we aim at optimizing CUDA kernels, such that for each possible value of the machine and program

parameters, we provide a correspondingly optimal CUDA kernel. Of course, the meaning of *optimal* has to be clearly defined, which will be done in Chapter 7.

To continue with our example, consider two machine parameters for the targeted GPGPU device:

Z: the maximum number of shared memory words per processor supported by the hardware architecture,

R: the maximum number of registers per thread supported by the hardware architecture.

For the C program in Example 1, we generate two CUDA kernels based on the possible values of the machine and program parameters, see Figure 1.4.

Observe that each thread in `kernel2` moves two elements of array `In` to the corresponding positions of array `Out`, whereas `kernel1` is identical to the kernel in Figure 1.3. By doing so, `kernel2` increases arithmetic intensity so as to hide the data transfer time between the global and shared memories for array `In`; however, this increases both register usage (from 6 to 8) and shared memory usage (from B to $2*B$) machine words. Therefore, `kernel2` works correctly and is optimal under the system of the constraints C_2 but not under the system of the constraints C_1 .

1.1 Contributions of this thesis

One of main objectives of this thesis is to generate CUDA kernels with program and machine parameters represented by unknown symbols. We call such kernels *parametric*. These parameters need not be known at code-generation time. Machine parameters (e.g. shared memory size) and program parameters (e.g. number of threads per thread-block) can be, respectively, determined when the generated CUDA code is installed on the targeted hardware. The challenge here is that any manipulation of non-linear expressions to be generated in the CUDA kernel requires specific code generation techniques relying on algebraic computation.

A second objective is to optimize parametric CUDA programs at compile-time, in the form of a case discussion, where cases depend on the possible values of machine parameters and program parameters. This leads us to the concept of *comprehensive parametric CUDA kernels*. To be more precise, given an input annotated C code., this is a decision tree, where each edge holds a Boolean expression (given by polynomial constraints) and each leaf is either a CUDA program such that for each leaf K we have:

1. K works correctly under the conjunction of the Boolean expressions located between the root node and the leaf, and
2. K is semantically equivalent to C .

In each Boolean expression, the unknown variables represent machine parameters and program parameters. This case discussion can be handled by techniques from symbolic computation. Automatic parametric kernel code generation can, then, be achieved by means of combining an optimizing compiler and a computer algebra system.

A third objective of this thesis is to measure the performance of *parametric* algorithms or programs targeting many-core devices like GPGPUs. To this end, a model of multithreaded computation targeting many-core architectures is introduced, such that one can either tune a program parameter to determine a value range minimizing parallelism overheads, or compare different multi-threaded algorithms solving the same problem.

$$C_1 : \begin{cases} B \leq Z < 2B \\ \cup \{ 6 \leq R < 8 \end{cases}$$

```

__global__ void kernel1(int *In, int *Out, int N, int B)
{
    int idx = blockIdx.x * B + threadIdx.x;
    // BLOCK_0 should be pre-defined as a constant
    // and be equal to B
    __shared__ int shared_In[BLOCK_0];
    if (idx < N) {
        shared_In[threadIdx.x] = In[idx];
        __syncthreads();
        Out[N-1-idx] = shared_In[threadIdx.x];
    }
}
dim3 dimBlock(B);
dim3 dimGrid(N/B);
kernel1 <<<dimGrid, dimBlock>>> (dev_In, dev_Out, N, B);

__global__ void kernel2(int *In, int *Out, int N, int B)
{
    int even_idx = blockIdx.x * 2 * B + 2 * threadIdx.x;
    int odd_idx = blockIdx.x * 2 * B + 2 * threadIdx.x + 1;
    // BLOCK_0 should be pre-defined as a constant
    // and be equal to B
    __shared__ int shared_In[2*BLOCK_0];
    if (even_idx < N && odd_idx < N) {
        shared_In[2*threadIdx.x] = In[even_idx];
        shared_In[2*threadIdx.x+1] = In[odd_idx];
        __syncthreads();
        Out[N-1-even_idx] = shared_In[2*threadIdx.x];
        Out[N-1-odd_idx] = shared_In[2*threadIdx.x+1];
    }
}
dim3 dimBlock(B);
dim3 dimGrid(N/(2*B));
kernel2 <<<dimGrid, dimBlock>>> (dev_In, dev_Out, N, B);

```

$$C_2 : \begin{cases} 2B \leq Z \\ 8 \leq R \end{cases}$$

Figure 1.4: Two CUDA kernels based on possible values of machine and program parameters for reversing a one-dimensional array, where Z is the maximum number of shared memory words per processor supported by the hardware architecture and R is the maximum number of registers per thread supported by the hardware architecture

We observe that classical models of parallel computation, namely the fork-join concurrency model [11] and the parallel random access machine (PRAM) model [110, 51], do not distinguish between the task-based and data-based parallelism. Thus, those models are too simplistic for analyzing algorithms targeting GPGPUs.

1.2 Outline of this thesis

In Chapter 2, various topics related to our work are reviewed. We first discuss briefly classical models of computation for concurrency platforms. In particular, we review the CUDA programming model and some important features of modern GPU architectures [36, 37]. We also

review the **MetaFork** language [29], which is a linguistic extension of C/C++ with high-level parallel programming constructs. Similarly to **OpenMP** and **OpenACC**, the **MetaFork** language offers a *high-level* model for accelerator programming. We stress the fact that this thesis does not deal with dependence analysis and the computation of schedules in automatic parallelization; this can be done via the polyhedral model [46, 15] that we briefly review in Chapter 2. Finally, in that same chapter, we give an overview of the algorithmic tools [4, 25, 23, 24, 112, 22] for dealing with systems of non-linear polynomial equations and inequalities.

In Chapter 3, we present the *Basic Polynomial Algebra Subprograms* (BPAS) library for arithmetic operations with univariate and multivariate polynomials, dense or sparse. The BPAS library is written in **CilkPlus** targeting multi-core architectures. The original goal of this thesis was to investigate how to integrate code targeting GPGPUs (for instance, from the **CUMODP** library) within the BPAS library. These two libraries, BPAS and **CUMODP**, are developed in our research group; moreover, this leads us to the idea of developing frameworks, such as **MetaFork**, for translating programs between different concurrency platforms. We also note that, the BPAS library, as a computer algebra library targeting high-performance, could also be used, in the future, for improving the efficiency of our algebraic tools for generating parametric CUDA kernels. The work reported in Chapter 3 is an extended version of [20] as well as a joint project with Changbo Chen, Svyatoslav Covanov, Farnam Mansouri, Robert Moir, Marc Moreno Maza and Yuzhen Xie. Appendix A lists sample codes from the BPAS library.

Chapter 4 presents a model of multithreaded computation with an emphasis on estimating the parallelism overheads of programs targeting many-core machines. To evaluate the performance of a given CUDA-like program, we consider three complexity measures, namely *work*, *span* and *parallelism overhead*. We also extend the Graham-Brent theorem so as to estimate the running time of a CUDA-like program on a given number of streaming multiprocessors. We evaluate the benefits of our model on six fundamental algorithms, including the Euclidean algorithm for univariate polynomial GCDs, two fast Fourier transform algorithms, the plain and FFT-based univariate polynomial multiplication algorithms, and radix sort [103]. Each studied algorithm is either implemented in the **CUMODP** library or reported in [103]. We observe that experimentation is coherent with the theoretical analysis based on our model. The work in Chapter 4 is an extended version of [61] as well as a joint project with Sardar Anisul Haque and Marc Moreno Maza.

Chapter 5 reports on a preliminary implementation of the C-to-CUDA code generator discussed above. Generating parametric CUDA kernels implies dealing with non-linear polynomial expressions, particularly during the tiling phase. To achieve these algebraic calculations, we take advantage of quantifier elimination (QE) and its implementation of the **RegularChains** library of **MAPLE** [27]. In order to illustrate the merits of parametric CUDA kernels, we use our code generator on eight test cases: array reversal, 1D Jacobi, 2D Jacobi, LU decomposition, matrix transposition, matrix addition, matrix vector multiplication and matrix matrix multiplication. The performance evaluation of the generated CUDA programs of each test case is provided as well. Chapter 5 is related to [19] and is joint work with Changbo Chen, Xiaohui Chen and Marc Moreno Maza. Appendix C provides documentation of our **MetaFork**-to-CUDA code generator, while Appendix D collects the generated CUDA programs by **PPCG** for those same eight test cases.

In Chapter 6, based on the experimental results conducted in Chapter 5, we study advanced optimization techniques, such as the controlling granularity of threads and loop unrolling, We

use six test cases so as to verify whether these techniques can, in general, further improve the performance of parametric CUDA kernels. For each test case, since we use the MetaFork language as the high-level accelerator programming model, the portion of the code that should be translated into CUDA kernels is annotated manually. Along with the previous chapter, experimentation shows that the generation of parametric CUDA kernels can lead to significant performance improvement w.r.t. approaches based on the generation of CUDA kernels that are not parametric. Moreover, for certain test cases, our experimental results show that the optimal choices for program parameters may depend on the problem size.

In Chapter 7, we propose an algorithm for *comprehensive optimization* of an annotated C program, depending on parameters treated as symbols at compile-time. We use this algorithm to generate *optimized parametric CUDA kernels* in the form of a case discussion based on the possible values of the machine and program parameters. In our preliminary implementation of the *comprehensive optimization* algorithm, we consider two machine parameters: register usage per thread and required shared memory per thread-block; meanwhile, we apply four code optimization strategies: caching data in local or shared memory, reducing register usage per thread, controlling thread granularity and eliminating common sub-expressions. This is a *proof-of-concept* implementation written in MAPLE and dedicated to the comprehensive generation of optimized MetaFork programs from an input MetaFork program. For each of the six test cases: array reversal, matrix vector multiplication, 1D Jacobi, matrix addition, matrix transpose and matrix matrix multiplication, three optimized MetaFork programs are generated with systems of constraints. Chapter 7 is joint work with Xiaohui Chen and Marc Moreno Maza. Appendix E provides the pseudo-codes of the implemented algorithms.

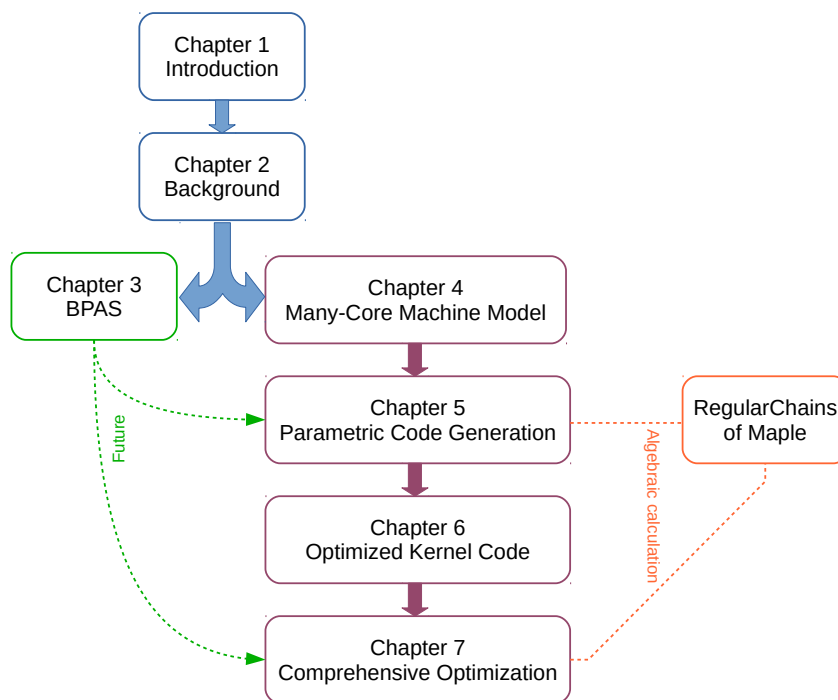


Figure 1.5: Overview of the thesis

To summarize, Figure 1.5 gives an overview of this thesis. The BPAS library is freely

available in source at <http://www.bpaslib.org>, and the MetaFork-to-CUDA code generator is freely available in source at <http://www.metafork.org>.

In the future, we plan to apply our MetaFork-to-CUDA code generator to efficiency-critical routines of the BPAS library code. Meanwhile, we plan to improve the run-time performance of our MetaFork-to-CUDA code generator by using BPAS instead of MAPLE for dealing with the non-linear expressions arising in the generation of parametric CUDA code.

Chapter 2

Background

In this chapter, we review background materials related to our work. Section 2.1 is dedicated to classical models of parallel computation. Section 2.2 discusses briefly the features of modern GPU architectures and the programming model CUDA. Section 2.3 gives an overview of the CUMODP library, which is used in the experimentation reported in Chapter 4. In Section 2.4, we present the MetaFork language, which serves as a high-level parallel programming model for automatic CUDA code generation in Chapters 5, 6 and 7. In Section 2.5, we study Feautrier’s and PLUTO’s algorithms in the so-called *polyhedral model*. Finally, Section 2.6 is an overview of the algorithmic tools used for dealing with systems of polynomial equations and inequalities.

2.1 Models of computation

Based on Flynn’s taxonomy [47], *single instruction multiple data* (SIMD) is defined as one of the categories of parallel computers, where multiple processing units execute the same instructions on multiple data sets. In this scheme, data is distributed across different processing units so as to achieve *data parallelism*. In contrast, *task parallelism* distributes tasks (or functions) to different processing units.

As mentioned before, with the pervasive ubiquity of many-core processors, in particular GPUs, models of computation must take into account both task-based and data-based parallelism. In fact, popular concurrency platforms (such as CilkPlus [10, 79], CUDA [95, 73] and OpenCL [111]) offer both forms of parallelism, with parallel constructs specific to each case.

Hereafter, we review two classical models of parallel computation, the fork-join model in Section 2.1.1 and the parallel RAM (PRAM) model in Section 2.1.2. Those models do not distinguish between task-based parallelism and data-based parallelism; thus, those models are too simplistic for analyzing algorithms targeting many-cores.

2.1.1 Fork-join model

In the fork-join model [11], one can consider a multithreaded program (see Chapter 27 in [34]) as a directed acyclic graph $G = (V, E)$, called *computation DAG*. Figure 2.1 shows the computation DAG for calculating the 5-th Fibonacci number, namely F_4 , where $F_n = F_{n-1} + F_{n-2}$ if

$n \geq 2$ and $F_0 = F_1 = 1$ otherwise. In a multithreaded program, we call a *strand* a sequence of consecutive instructions without parallel constructs.

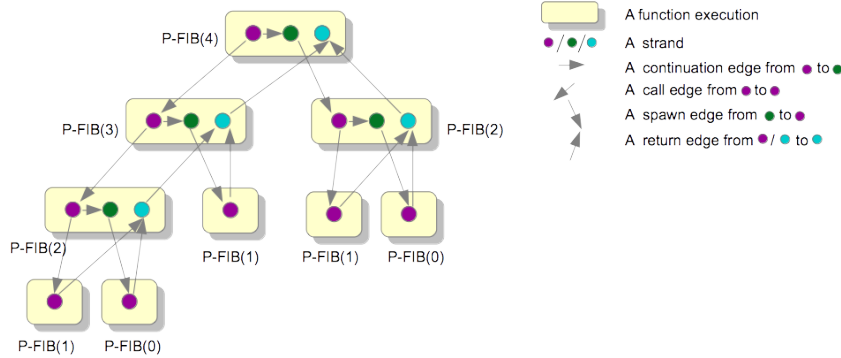


Figure 2.1: An example of computation DAG: Fourth Fibonacci

This model defines two complexity measures:

1. the *work* T_1 of a multithreaded computation is the total time to execute the entire computation on one processor, and
2. the *span* T_∞ is the longest time to execute the strands along any path in the DAG.

Assuming that each strand executes in unit time, the work is the number of vertices, and the span equals the number of vertices on a longest path or *critical path* in the computation DAG.

Those two measures, work and span, and one machine parameter, the number P of processors, can be combined in results like the Graham-Brent theorem ([11, 53]) or the Blumofe-Leiserson theorem (Theorems 13 & 14 in [12]) in order to give running time estimates. We recall that the Graham-Brent theorem states that the running time T_P on P processors satisfies $T_P \leq T_1/P + T_\infty$. A refinement of this latter theorem actually supports the implementation (on multi-core architectures) of the parallel performance analyzer, called *Cilkview* [63]. In this context, the running time T_P is bounded in expectation by $T_1/P + 2\delta\widehat{T}_\infty$, where δ is a constant (called the *span coefficient*) and \widehat{T}_∞ is the so-called *burdened span*.

The fork-join model has become popular with the development of the concurrency platform *CilkPlus* [10, 79] targeting multi-core architectures. An example of code written in *CilkPlus* is given in Figure 2.2 for computing matrix multiplication. This *CilkPlus* code is implemented with a divide & conquer method and the blocking strategy reported in [49] by Frigo, Leiserson, Prokop and Ramachandran.

2.1.2 PRAM model

The PRAM model [110, 51] is defined as a synchronous model of parallel computation. The PRAM machine consists of a number of processors, each of which is a RAM with a private local memory, and a shared memory that processors communicate with. Since the amount of shared memory is limited, it restricts the amount of data that can be communicated between processors in one step.

Moreover, the PRAM model defines four types of accesses to the same shared memory cell with respect to read and write operations. These four types are *exclusive read exclusive write*

```

/**
 * Square matrices A, B, C of order N
 * The base case size is set to X
 * Call to parallel_dandc(0, N, 0, N, 0, N, A, B, C, N, X);
 */
void parallel_dandc(int i0, int i1, int j0, int j1, int k0, int k1, int* A,
                   int* B, int* C, int N, int X) {

    int di = i1 - i0;
    int dj = j1 - j0;
    int dk = k1 - k0;
    if (di >= dj && di >= dk && di >= X) {
        int mi = i0 + di / 2;
        cilk_spawn parallel_dandc(i0, mi, j0, j1, k0, k1, A, B, C, N, X);
        parallel_dandc(mi, i1, j0, j1, k0, k1, A, B, C, N, X);
        cilk_sync;
    }
    else if (dj >= dk && dj >= X) {
        int mj = j0 + dj / 2;
        cilk_spawn parallel_dandc(i0, i1, j0, mj, k0, k1, A, B, C, N, X);
        parallel_dandc(i0, i1, mj, j1, k0, k1, A, B, C, N, X);
        cilk_sync;
    }
    else if (dk >= X) {
        int mk = k0 + dk / 2;
        parallel_dandc(i0, i1, j0, j1, k0, mk, A, B, C, N, X);
        parallel_dandc(i0, i1, j0, j1, mk, k1, A, B, C, N, X);
    }
    else {
        // The base case using the serial, naive matrix multiplication
        for (int i = i0; i < i1; ++i)
            for (int j = j0; j < j1; ++j)
                for (int k = k0; k < k1; ++k)
                    C[i * N + j] += A[i * N + k] * B[k * N + j];
    }
}

```

Figure 2.2: Matrix multiplication written in CilkPlus

(EREW), *concurrent read exclusive write* (CREW), *exclusive read concurrent write* (ERCW) and *concurrent read concurrent write* (CRCW). At one step, “exclusive” indicates that at most one processor can read from or write to the same memory cell, while “concurrent” indicates that processors can simultaneously read from or write to the same memory cell. The advantage of the PRAM model is that it focuses on designing efficient parallel algorithms in terms of arithmetic operations while ignoring communication issues. In addition, all memory accesses are assumed to take place in constant time. Unfortunately, concrete machines cannot currently scale to large numbers of processors while preserving uniformly fast access time to the shared memory.

An attempt to integrate memory contention into the PRAM model has been made with the *queue read queue write* (QRQW) PRAM, defined in [52]. This model enhances the Graham-Brent theorem with memory access time. However, both time spent in arithmetic operations

and time spent in read/write accesses are conflated in a single quantity. We believe that this unification is not appropriate for recent many-core processors, such as NVIDIA GPUs, for which the ratio between one global memory read/write access and one floating point operation can be in the 100's.

A more practical PRAM model is proposed in [51] called *asynchronous PRAM*. Unlike the PRAM model, the processors of an asynchronous PRAM run asynchronously, that is, each processor executing its instructions independently without interrupting others. It considers subset independent synchronization and fixed communication delays for global reads/writes. However, this family of PRAM is not suitable to modern GPUs, since it is trivial to consider a machine parameter for synchronization, which is not an issue for modern GPUs.

2.2 General-purpose GPU computing

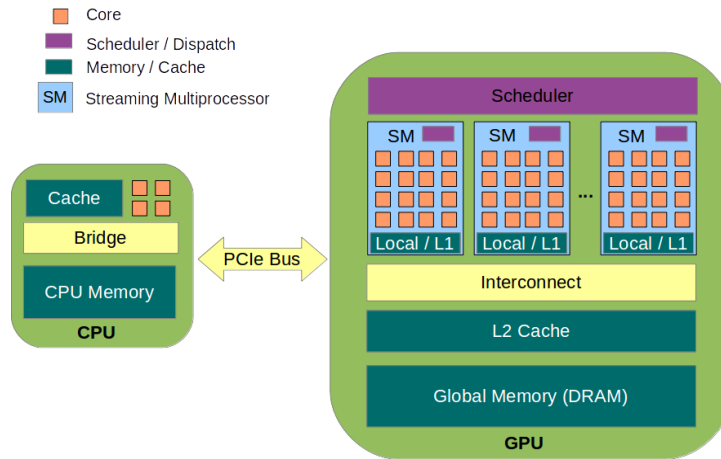


Figure 2.3: Overview of a hybrid CPU-GPU system

General-purpose GPU computing has evolved from graphics processors to massively parallel many-core multiprocessors with the adoption of co-processing between the CPU and the GPU. In such a hybrid CPU-GPU system, GPUs can be used to accelerate a portion of sequential code by employing task-parallelism and data-parallelism. Figure 2.3 shows an overview of a typical CPU-GPU system architecture. One of the main differences between the CPU and the GPU is the number of cores that are designed for different types of execution. Indeed, a GPU device consists of several *streaming multiprocessors* (SMs); meanwhile, each SM consists of a large number of processing cores. For this reason, these cores are particularly designed to take advantage of parallel execution in the SIMD (*single instruction multiple data*) manner. While memory coherence is maintained by the CPU, the programmer needs to explicitly deal with data management on the GPU. Section 2.2.1 introduces the programming model, the *Compute Unified Device Architecture* (CUDA). In Section 2.2.2, we summarize the characteristics of modern GPU architectures.

2.2.1 The Compute Unified Device Architecture (CUDA)

CUDA is the programming model introduced to implement parallel algorithms on NVIDIA GPU devices [95, 73]. A CUDA program consists of one or more blocks of code that are executed on either the host (CPU) or the device (GPU). On the host, the code is implemented with little or no parallelism, while on the device, the code is designed to exhibit a rich amount of task or data parallelism, in particular, in a SIMD fashion.

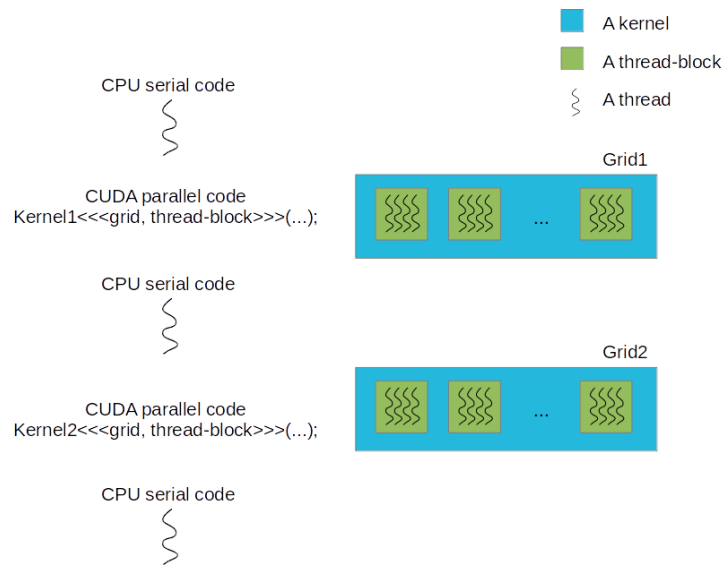


Figure 2.4: Execution of a CUDA program

The execution of a typical CUDA program is shown in Figure 2.4. It starts with the serial C/C++ code and then launches a kernel on the device to execute the parallel code. Upon the completion of the kernel, the process resumes to the execution of serial code on the host. When a kernel function is invoked, the number of thread-blocks and the number of threads per thread-block are specified, according to the problem size, by the programmer. The syntax for launching a kernel from the host code extends that of a C function call with kernel execution configuration parameters surrounded by `<<<` and `>>>`, where the execution configuration parameters define the dimensions of the grid and the dimensions of each thread-block. All the threads comprising a kernel during an invocation are collectively called a *grid*.

In addition, the programmer needs to allocate the problem size on the device memory using the keyword `cudaMalloc` and free the device memory after computation using the keyword `cudaFree`. To transfer pertinent data between the host and allocated device memories, one can use the keyword `cudaMemcpy`, with `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost` specifying where the data is copied from and to. The function type qualifier `__global__`¹, written prior to a function declaration, indicates that the function is a kernel; meanwhile, on the host, this kernel is called to generate a grid of threads on the device. Moreover, `threadIdx`,

¹See <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> for CUDA C programming guide.

`blockIdx` and `blockDim` are built-in variables to identify the thread index, the thread-block index and the thread-block dimension, respectively.

2.2.2 Modern GPU architectures

Unlike CPUs optimized for low latency access to memory systems, modern GPUs are designed for data-parallel, throughput computation and tolerance of memory latency. A GPU typically consists of a global memory (*dynamic random-access memory* (DRAM)), a number of *streaming multiprocessors* (SMs) and a global scheduler. Each SM contains a number of cores, a shared memory and a thread scheduler.

Apart from the evolution of graphic techniques on GPUs, we summarize key features of modern GPU architectures with respect to general-purpose GPU computing as below.

1. *Hardware thread scheduling.* Based on the documentation of NVIDIA Fermi [36] and Kepler [37] architectures, there is a two-level, distributed thread scheduler, such that thread-blocks from the same or different kernels are scheduled to various SMs and warps of 32 threads of a thread-block are distributed to execute concurrently.
2. *Memory hierarchy.* This hierarchy offers the benefits of the *on-chip* local memory for each thread, the *on-chip* shared memory shared among threads in an SM with low latency, and the global memory for sharing across the GPU with high throughput. In order to declare an array allocated in the shared memory of the SM, one shall use the `__shared__` qualifier prior to the declaration of the array (see Figure 1.3 for an example). Moreover, when a warp of threads accesses consecutive memory locations to the global memory, the hardware *coalesces* its memory accesses into a consolidated access.
3. *Data parallelism.* This is implemented as SIMD processors or in a *single instruction multiple thread* (SIMT) fashion.
4. *Task parallelism.* This type of parallelism, also called *dynamic parallelism*, can launch a new kernel from a thread, synchronize on the results and control the scheduling via the hardware paths; moreover, these operations are done independently of the CPU. This allows data-dependent and recursive code to execute in parallel on GPUs.
5. *Latency hidden.* This is done via computation by SIMT (instead of cache). Based on the concept of *fast context switching*, active warps of threads that wait for memory accesses are switched to those whose data are available. Typically, 128 threads per SM occupy an SM during the computation to hide the global memory access latency.
6. *Synchronization.* Threads are independent of each other, such that there are no synchronization issues among thread-blocks per grid, while threads with a thread-block can be synchronized by issuing `__syncthreads()`. Additionally, synchronization of task parallelism is explicitly managed prior to launching a (child) kernel to ensure that all data is ready. If thread divergence occurs, both sides of the branch execute within a warp, and idle threads in a warp wait for others to complete.

To be consistent with the programming framework, a thread-block is mapped to one SM, such that threads within a thread-block communicate via the shared memory with low latency and low throughput. Thread-blocks mapped to different SMs communicate via the global memory with high latency and high throughput.

2.3 Dense arithmetic over finite fields with the CUMODP library



CUMODP [59] is a CUDA library for exact computations with dense polynomials over finite fields. A variety of operations, like multiplication, division, computation of subresultants, multi-point evaluation, interpolation and many others, are provided. These routines are primarily designed to offer GPU support to polynomial system solvers and a bivariate system solver is part of the library. Algorithms combine FFT-based and plain arithmetic, while the implementation strategy emphasizes reducing parallelism overheads and optimizing hardware usage.

Polynomial multiplication and matrix multiplication are at the core of many algorithms in symbolic computation. Expressing, in terms of multiplication time, the algebraic complexity of an operation like univariate polynomial division or the computation of a characteristic polynomial is a standard practice, see for instance the landmark book [120]. At the software level, the motto “reducing everything to multiplication”² is also common, see for instance the computer algebra systems Magma [16], NTL [108] or FLINT [62].

With the advent of hardware accelerator technologies, multi-core processors and Graphics Processing Units (GPUs), this reduction to multiplication is, of course, still desirable, but becomes more complex since both algebraic complexity and parallelism need to be considered when selecting and implementing a multiplication algorithm. In fact, other performance factors, such as cache usage or CPU pipeline optimization, should be taken into account on modern computers, even on single-core processors. These observations guide the developers of projects like SPIRAL [98] or FFTW [48].

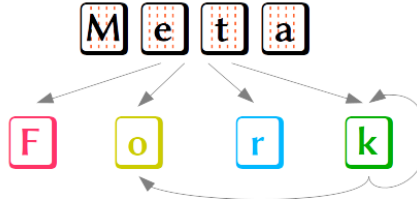
The CUMODP library provides arithmetic operations for matrices and polynomials (in a dense representation) primarily with modular integer coefficients, targeting many-core GPUs. Some operations are available for integer or floating point coefficients as well. A large portion of the CUMODP library code is devoted to polynomial multiplication and the integration of that operation into higher-level algorithms.

Typical CUMODP operations are matrix determinant computation, polynomial multiplication (both plain and FFT-based), univariate polynomial division, the Euclidean algorithm for univariate polynomial GCDs, subproduct tree techniques for multi-point evaluation and interpolation, subresultant chain computation for multivariate polynomials and bivariate system solving. The CUMODP library is written in CUDA [95, 73] and its source code is publicly available at www.cumodp.org.

This work is reported in [59] and is a joint project with Sardar Anisul Haque, Xin Li, Farnam Mansouri, Marc Moreno Maza and Wei Pan.

²Quoting a talk title by Allan Steel, from the Magma Project.

2.4 The MetaFork language



MetaFork [29] is a high-level programming language extending C/C++, which combines several models of concurrency, including fork-join and pipelining parallelisms. MetaFork is also a compilation framework, which aims at facilitating the design and implementation of concurrent programs through three key features:

1. Perform automatic code translation between concurrency platforms targeting both multi-core and many-core GPU architectures.
2. Provide a high-level language for expressing concurrency as in the fork-join model, the SIMD (*single instruction multiple data*) paradigm and the pipelining parallelism.
3. Generate parallel code from serial code with an emphasis on code depending on machine or program parameters (e.g. cache size, number of processors, number of threads per thread-block).

As of today, the publicly available and latest release of MetaFork, see www.metafork.org, offers the second feature stated above, a preliminary implementation of the third feature as well as the multi-core and many-core portions of the first one. To be more specific, MetaFork is a meta-language for concurrency platforms based on the fork-join model, pipelining parallelism and the SIMD paradigm. This meta-language forms a bridge between actual multi-threaded programming languages, and we use it to perform automatic code translation between those languages.

In an earlier work [29], MetaFork was introduced as an extension of both the C and C++ languages into a multithreaded language based on the fork-join concurrency model [11]. Thus, concurrent execution is obtained by a parent thread creating and launching one or more children threads, so that the parent and its children execute a so-called *parallel region*. An important example of parallel regions are for-loop bodies. MetaFork has four parallel constructs dedicated to the fork-join model: function call spawn, block spawn, parallel for-loop and synchronization barrier. The first two use the keyword `meta_fork`, while the other two use, respectively, the keywords `meta_for` and `meta_join`. Similar to the CilkPlus specifications, the parallel constructs of MetaFork grant permission for concurrent execution but do not command it. Hence, a MetaFork program can execute on a single core machine. We emphasize the fact that `meta_fork` allows the programmer to spawn a function call (like in CilkPlus [10, 79, 35]) as well as a block (like in OpenMP [41, 13, 9]). Using the same examples from [29], Figures 2.5, 2.6 and 2.7 illustrate automatic code translation between the OpenMP program and the CilkPlus program via the MetaFork language.

On the other hand, stencil computations are a major pattern in scientific computing. Stencil codes perform a sequence of sweeps (called time-steps) through a given array, and each sweep can be seen as the execution of a pipeline. When expressed with concurrency platforms based on, and limited by, the fork-join model, parallel stencil computations incur excessive

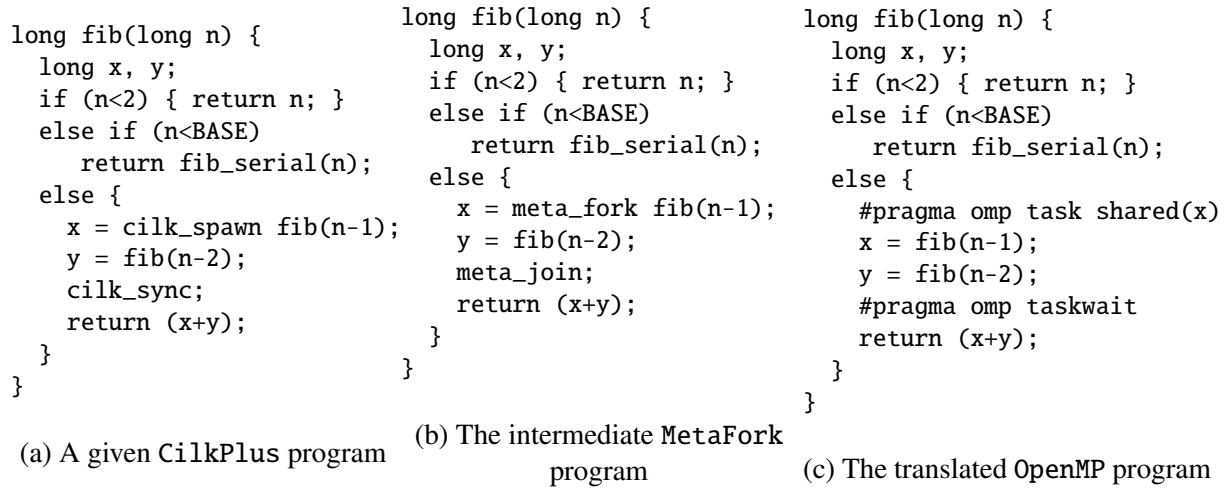


Figure 2.5: Using MetaFork to translate a given CilkPlus program into a OpenMP program

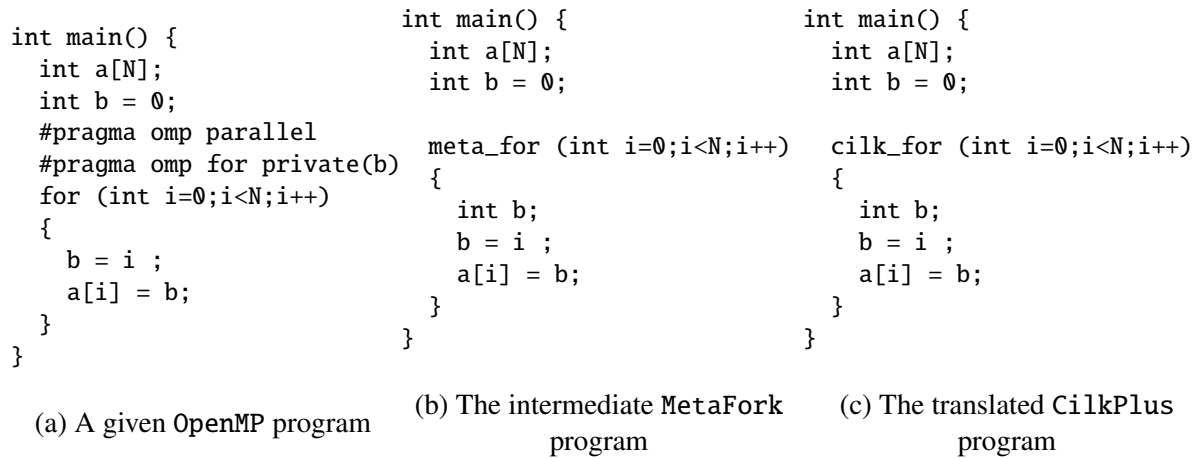


Figure 2.6: Using MetaFork to translate a given OpenMP program into a CilkPlus program

parallelism overheads. This problem is studied by Shirako, Unnikrishnan, Chatterjee, Li and Sarkar [107] together with a solution in the context of OpenMP by proposing new synchronization constructs to enable *do-across parallelism*. These observations have motivated a first extension of the MetaFork language with three constructs to express *pipelining parallelism*: `meta_pipe`, `meta_wait` and `meta_continue`. Recall that a pipeline is a linear sequence of processing stages through which data items flow from the first stage to the last stage. If each stage can process only one data item at a time, then the pipeline is said to be *serial* and can be depicted by a (directed) path in the sense of graph theory. If a stage can process more than one data item at a time, then the pipeline is said to be *parallel* and can be depicted by a directed acyclic graph (DAG), where each parallel stage is represented by an independent set, that is, a set of vertices of which no pair is adjacent.

In order to generate efficient CUDA code from an input MetaFork program, we introduced a tenth keyword, namely `meta_schedule`, in [19]. This keyword allows its body to be scheduled on a device, such as the NVIDIA GPU, to execute in a SIMD fashion. In Chapter 5 as well as Appendix C, we depict the MetaFork-to-CUDA code generator, which is capable of


```

int main() {
    int sum_a = 0, sum_b = 0;
    int a[ 5 ] = {0,1,2,3,4};
    int b[ 5 ] = {0,1,2,3,4};
    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                for(int i=0; i<5; i++)
                    sum_a += a[ i ];
            }
            #pragma omp section
            {
                for(int i=0; i<5; i++)
                    sum_b += b[ i ];
            }
        }
    }
}

```

(a) A given OpenMP program

```

int main() {
    int sum_a = 0, sum_b = 0;
    int a[ 5 ] = {0,1,2,3,4};
    int b[ 5 ] = {0,1,2,3,4};

    meta_fork shared(sum_a) {
        for(int i=0; i<5; i++)
            sum_a += a[ i ];
    }
    meta_fork shared(sum_b) {
        for(int i=0; i<5; i++)
            sum_b += b[ i ];
    }

    meta_join;
}

```

(b) The intermediate MetaFork program

```

void fork_func0(int* sum_a,int* a) {
    for(int i=0; i<5; i++)
        (*sum_a) += a[ i ];
}
void fork_func1(int* sum_b,int* b) {
    for(int i=0; i<5; i++)
        (*sum_b) += b[ i ];
}

int main() {
    int sum_a = 0, sum_b = 0;
    int a[ 5 ] = {0,1,2,3,4};
    int b[ 5 ] = {0,1,2,3,4};
    cilk_spawn fork_func0(&sum_a,a);
    cilk_spawn fork_func1(&sum_b,b);
    cilk_sync;
}

```

(c) The translated CilkPlus program

Figure 2.7: Using MetaFork to translate a given OpenMP program into a CilkPlus program

automatically generating compilable CUDA programs with program parameters (like number of threads per thread-block) and machine parameters (like shared memory size) allowed at code-generation-time.

This work is summarized in [19].

2.5 Automatic parallelization in the polyhedral model

Automatic parallelization deals mainly with the transformation of for-loop nests so as to expose parallelism, that is, being able to execute one or more for-loops in a parallel fashion. By executing a for-loop in a parallel fashion, we mean that all iterations of that loop can be executed concurrently without changing the semantics of that loop. For the sake of simplicity, we shall make the following assumption in the sequel of this section.

Hypothesis 1 *We consider for-loop nests having the format shown on Figure 2.8. Note that*

```

for (i_1 = 0; i_1 < N_1; i_1++)
.
.
.
for (i_e = 0; i_e < N_e; i_e++) {

//      for-loop nest body

}

```

Figure 2.8: For-loop nest in the polyhedral model

Figure 2.8 suggests that the for-loop nest is perfect, that is, only the innermost loop may have statements other than a for-loop.

In fact, imperfect for-loop nests can be reduced to perfect for-loop nests [71] without change of semantics but with possible loss of efficiency in terms of consumption of computer resources (time and space).

Paul Feautrier's [46] and PLUTO's [15] algorithms are two procedures for generating parallel code from serial C code automatically. The two procedures focus on parallelizing for-loop nests (including imperfect ones) under the following assumption.

Hypothesis 2 *Referring to the notations introduced in Figure 2.8, in every array reference (e.g. $i_1 * N_1 + i_2$ in $a[i_1 * N_1 + i_2]$) in the for-loop nest body,*

- (i) *the index is an arithmetic expression linear in the vector $(i_1, \dots, i_e, N_1, \dots, N_e)$, where*
- (ii) *each of N_1, \dots, N_e is an arithmetic expression linear in all the variables occurring in that expression.*

Note that $i_1 * N_1 + i_2$ is linear in each of the variables i_1, N_1, i_2 , but not in the vector (i_1, i_2, N_1) .

Hypothesis 2 implies that the iterations of the for-loop nest can be represented by the integer points of a polyhedron. For that reason, Feautrier's and PLUTO's algorithms are said to be based on the *polyhedral model*. Before giving a sketch of those algorithms in Sections 2.5.7 and 2.5.8, we review the concepts of a \mathbb{Z} -polyhedron in Section 2.5.1, a polyhedral iteration domain in Section 2.5.2, a data dependence graph in Section 2.5.3, the dependence polyhedron in Section 2.5.4, an affine transformation in Section 2.5.5 and Farkas multipliers in Section 2.5.6.

2.5.1 \mathbb{Z} -polyhedron

Let \mathbb{Z} be the ring of the integer numbers and m be a positive integer. The set of all vectors $\vec{x} \in \mathbb{Z}^m$ where $\mathbf{h} \cdot \vec{x} = k$ with $\mathbf{h} \in \mathbb{Z}^m$ and $k \in \mathbb{Z}$ is an *affine \mathbb{Z} -hyperplane*. The set of all vectors $\vec{x} \in \mathbb{Z}^m$ where $A \vec{x} + \vec{c} \geq 0$ with an $p \times m$ integer matrix A (for $p > 0$) and $\vec{c} \in \mathbb{Z}^m$ defines a *\mathbb{Z} -polyhedron*. A *\mathbb{Z} -polytope* is a bounded \mathbb{Z} -polyhedron.

2.5.2 Polyhedral iteration domain

We observe that every `for`-loop nest in the format shown on Figure 2.8 naturally defines a \mathbb{Z} -polytope \mathcal{D} in \mathbb{Z}^e , called *iteration domain* of the `for`-loop nest; moreover, the points of \mathcal{D} are called *iteration vectors*. Let $\vec{I} = (I_1, \dots, I_e)$ and $\vec{J} = (J_1, \dots, J_e)$ be two iteration vectors in \mathcal{D} . We write $\vec{J} <_{\text{lex}} \vec{I}$ whenever \vec{I} is lexicographically greater than \vec{J} . We write $\vec{J} \leq_{\text{lex}} \vec{I}$ if either $\vec{J} <_{\text{lex}} \vec{I}$ or $\vec{J} = \vec{I}$ holds. We call *defining system* of \mathcal{D} the system of linear inequalities:

$$\begin{cases} 0 \leq i_1 < N_1 \\ \vdots \\ 0 \leq i_e < N_e \end{cases}$$

Let $\vec{I} = (I_1, \dots, I_e)$ and S be respectively an iteration vector of \mathcal{D} and a statement of the body of our perfect loop nest. We denote by $S(\vec{I})$ the execution of S during the \vec{I} -th iteration of the `for`-loop nest.

2.5.3 Data dependence graph

The following definition is restricted to our context of a perfect `for`-loop nest, while it could be stated for an arbitrary program. Let \vec{I} and \vec{J} be two iteration vectors in \mathcal{D} . Let S_s and S_t be two statements of the `for`-loop nest body. We say that there is a *data dependence* from $S_s(\vec{I})$ to $S_t(\vec{J})$ (alternatively, we say that $S_t(\vec{J})$ *depends on* $S_s(\vec{I})$), and we write $S_s(\vec{I}) \implies S_t(\vec{J})$, whenever the following conditions hold simultaneously:

- (1) both $S_s(\vec{I})$ and $S_t(\vec{J})$ access the same memory location and at least one of them stores data into this memory location, and
- (2) we have $\vec{I} <_{\text{lex}} \vec{J}$.

The *data dependence graph* (DDG) $G = (V, E)$ is a labelled directed-graph, where

- (1) the set V of the vertices is the set of the statements of the `for`-loop nest body,
- (2) (S_s, S_t) is an edge if there exist two iteration vectors \vec{I} and \vec{J} such that we have $S_s(\vec{I}) \implies S_t(\vec{J})$,
- (3) each edge (S_s, S_t) is labelled with the set of the pairs (\vec{I}, \vec{J}) of iteration vectors such that we have $S_s(\vec{I}) \implies S_t(\vec{J})$.

We classify the data dependence from statement $S_s(\vec{I})$ to statement $S_t(\vec{J})$ into three categories, based on the sequences of read and write operations to the same memory location X :

- *True dependence* (also known as a *flow dependence*): X is written in S_s before it is read in S_t .

$$\begin{array}{lcl} S_s : & X & = \dots \\ S_t : & \dots & = X \end{array}$$

- *Anti-dependence*: X is read in S_s before it is written in S_t .

$$\begin{array}{lcl} S_s : & \dots & = X \\ S_t : & X & = \dots \end{array}$$

- *Output dependence*: X is written in S_s before it is written in S_t .

$$\begin{array}{lcl} S_s : & X & = \dots \\ S_t : & X & = \dots \end{array}$$

Example 2 Consider the source program given below:

```

for (int i = 0; i < N; ++i)
  for (int j = 1; j < N; ++j)
S:  A[i][j] = A[j][i] + A[i][j-1];

```

Figure 2.9 shows the data dependence graph of the source program in Example 2. Observe that data dependence occurs for each of the following:

- from $A[i][j-1]$ in statement S to $A[i'][j']$ in statement S when $i' = i$, $j' = j - 1$ (flow dependence),
- from $A[j][i]$ in statement S to $A[i'][j']$ in statement S when $i' = j$, $j' = i$, $i < j$ (flow dependence),
- from $A[i][j]$ in statement S to $A[j'][i']$ in statement S when $j' = i$, $i' = j$, $j < i$ (anti-dependence),

where (i, j) and (i', j') are two iteration vectors of the `for`-loop nest; thus, i, j, i', j' are non-negative integers satisfying $i < N$, $0 < j, j < N$, $i' < N$, $0 < j', j' < N$.

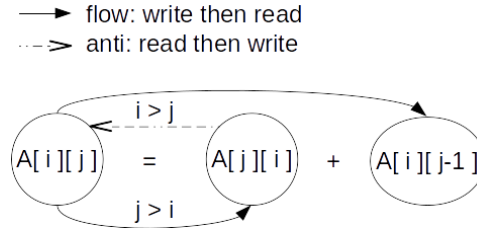


Figure 2.9: An example of the data dependence graph of the source program

2.5.4 Dependence polyhedron

Let $e := (S_s, S_t)$ be an edge of the data dependence graph. Let \vec{I} and \vec{J} be two iteration vectors such that we have $S_s(\vec{I}) \implies S_t(\vec{J})$. We call *dependence polyhedron* associated with e the \mathbb{Z} -polyhedron defined by the conjunction of the inequalities on \vec{I}, \vec{J} expressing the fact that the following conditions hold simultaneously:

- (i) $\vec{I} \in \mathcal{D}$,
- (ii) $\vec{J} \in \mathcal{D}$,
- (iii) $S_s(\vec{I}) \implies S_t(\vec{J})$.

In Example 2, for the flow dependence from $A[i][j-1]$ to $A[i'] [j']$ in Figure 2.9, we observe that the dependence polyhedron is

$$\left(\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 1 & -1 \\ \hline 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & -1 \end{array} \right) \begin{bmatrix} i \\ j \\ i' \\ j' \\ N \\ 1 \end{bmatrix} \geq 0 \quad (2.1)$$

$$\begin{array}{c} \hline \\ \hline \end{array} = 0$$

2.5.5 Affine transformation

Consider a for-loop nest and a statement S in the body of this loop nest. Let x_1, \dots, x_{m_s} be the loop iteration counters occurring in the indices of the array references in the statement S ; we write $\vec{x} = [x_1 \ \dots \ x_{m_s}]$. By definition, a *one-dimensional affine transformation* for the statement S maps \vec{x} to $\phi_S(\vec{x})$ defined by

$$\phi_S(\vec{x}) = [t_1 \ \dots \ t_{m_s}](\vec{x}) + t_0,$$

where $t_0 \in \mathbb{Z}$ and $[t_1 \ \dots \ t_{m_s}]$ is a row-vector over \mathbb{Z} . In the literature, the map ϕ_S is sometimes referred as an affine hyperplane. An *n-dimensional affine transformation* for statement S is given by

$$\phi_S(\vec{x}) = T(\vec{x}) + \vec{t},$$

where T is a constant matrix with n rows (and m_s columns) and \vec{t} is a constant vector of size n , the coefficients of both T and \vec{t} belonging to \mathbb{Z} . In the literature, the map ϕ_S is also called a *scattering function*.

In the polyhedral model, the notions of one-dimensional affine transformations and n -dimensional affine transformation are used to perform an affine change of coordinate system for the loop iteration counters so as to exhibit for-loop nests whose iterations can be executed concurrently. Note that, because of the inequality constraints that appear in the polyhedron, this cannot simply be done via linear algebra and requires polyhedral operations, like projection, which can be done by the Fourier-Motzkin elimination algorithm³ [122].

To be more precise, one-dimensional affine transformations and n -dimensional affine transformations are meant to be used as *time* coordinates and any other other coordinates are regarded as *space* coordinates. A tuple of space coordinates can be interpreted as a processor and the transformed for-loop can be interpreted as a schedule for each processor. Therefore, if such one-dimensional affine transformations exist, it can be understood as a way of executing the original for-loop in a parallel manner. In the sequel, we will call *parallelization* such a one-dimensional affine transformation.

³https://en.wikipedia.org/wiki/Fourier-Motzkin_elimination

For the source program of Example 2, one can obtain such new coordinates: $t = i + j$ and $p = i$, where t and p represent time and processor, respectively. The transformed code is shown in Figure 2.10 where each iteration of the p -loop can be executed concurrently.

```
// 1 <= t = i + j < 2N
for (int t = 1; t < 2*N; t++)
  // 0 <= p = i < N & 1 <= t - p = j < N
  parallel for (int p = max(0, t-N-1), p < min(t-2, N); p++)
    S': A[p][t-p] = A[t-p][p] + A[p][t-p-1];
```

Figure 2.10: The transformed code based on time and processor coordinates

2.5.6 Farkas multipliers

Let $\mathcal{D} \subset \mathbb{Z}^m$ be a nonempty \mathbb{Z} -polyhedron defined by p affine inequalities:

$$A_k \vec{x} + b_k \geq 0, \quad 1 \leq k \leq p,$$

where $A_k \in \mathbb{Z}^m$ and $b_k \in \mathbb{Z}$. It follows from Farkas' Lemma [46, 105] that an affine form $\phi : \vec{x} \in \mathbb{Z}^m \mapsto \phi(\vec{x}) \in \mathbb{Z}$ is non-negative in \mathcal{D} iff there exist non-negative integers $\mu_0, \mu_1, \dots, \mu_p$, called *Farkas multipliers*, such that for all $\vec{x} \in \mathbb{Z}^m$ we have:

$$\phi(\vec{x}) = \mu_0 + \sum_{k=1}^p \mu_k (A_k \vec{x} + b_k).$$

2.5.7 Feautrier's algorithm

In [46], Paul Feautrier introduced an algorithm, based on Farkas' Lemma, that, given a `for`-loop nest, determines a parallelization of that `for`-loop nest, if such affine transformation exists. We give a brief sketch of Feautrier's algorithm.

Consider a `for`-loop nest. Let us call *program parameter* any scalar variable read but not written in that `for`-loop nest. Write $\vec{n} = (n_1, \dots, n_d)$ a vector of all those program parameters. Typically, \vec{n} includes `for`-loop upper bounds, dimension sizes of arrays, etc. Consider a statement S_i in this loop nest and denote by \vec{x} a vector of the loop iteration counters occurring in the indices of the array references in S_i . Then, the iteration domain \mathcal{D}_{S_i} for S_i is given by a system of p linear inequalities of the following form:

$$A_{S_{i,k}} \begin{pmatrix} \vec{x} \\ \vec{n} \end{pmatrix} + b_{S_{i,k}} \geq 0, \quad 1 \leq k \leq p,$$

where $A_{S_{i,k}}$ is a row-vector over \mathbb{Z} and $b_{S_{i,k}} \in \mathbb{Z}$ holds. Applying Farkas' lemma, an one-dimensional affine transformation $\vec{x} \mapsto \phi_{S_i}(\vec{x})$ is non-negative in the domain \mathcal{D}_{S_i} iff there exist non-negative integers $\mu_{S_{i,k}}$ with $0 \leq k \leq p$, such that

$$\phi_{S_i}(\vec{x}) = \mu_{S_{i,0}} + \sum_k \mu_{S_{i,k}} \left(A_{S_{i,k}} \begin{pmatrix} \vec{x} \\ \vec{n} \end{pmatrix} + b_{S_{i,k}} \right). \quad (2.2)$$

In the sequel, we view ϕ as a function that, given a statement S_i and an iteration vector \vec{x} (of all the loop iteration counters occurring in the indices of the array references in S_j), maps (S_i, \vec{x}) to a $\phi_{S_i}(\vec{x})$, where ϕ_{S_i} is a one-dimensional affine transformation.

We say that ϕ is a *schedule* whenever for all statements S_i, S_j and all iteration vectors \vec{x} (resp. \vec{y}) of S_i (resp. S_j) such that $S_i(\vec{x}) \Rightarrow S_j(\vec{y})$ holds, we have:

$$\phi_{S_j}(\vec{y}) \geq \phi_{S_i}(\vec{x}) + 1.$$

One can interpret the number 1 as the time unit for executing a statement.

We associate each edge $e := (S_i, S_j)$ of the DDG with a *delay* denoted by Δ_e and defined by

$$\Delta_e = \phi_{S_j}(\vec{y}) - \phi_{S_i}(\vec{x}) - 1 \geq 0.$$

Recall that the dependence polyhedron associated with e (see Section 2.5.4) is a system of q linear inequalities of the form:

$$C_{e,k} \begin{pmatrix} \vec{x} \\ \vec{y} \\ \vec{n} \end{pmatrix} + d_{e,k} \geq 0, \quad 1 \leq k \leq q,$$

where $C_{e,k}$ is a row-vector over \mathbb{Z} and $d_{e,k} \in \mathbb{Z}$.

Due to the fact that a schedule does exist, we can find Farkas multipliers λ_{ek} for the delay Δ_e such that we have

$$\phi_{S_j}(\vec{y}) - \phi_{S_i}(\vec{x}) - 1 = \lambda_{e0} + \sum_k \lambda_{ek} \left(C_{e,k} \begin{pmatrix} \vec{x} \\ \vec{y} \\ \vec{n} \end{pmatrix} + d_{e,k} \right). \quad (2.3)$$

Equations 2.2 and 2.3 yield a system of linear equations and inequalities including the fact all μ 's and λ 's are non-negative integers. Using techniques from integer programming, in particular from the landmark paper [45] of P. Feautrier, one can compute values for the μ 's and finally deduce $\phi_{S_i}(\vec{x})$.

2.5.8 PLUTO's algorithm

In [15], Bondhugula, Hartono, Ramanujam and Sadayappan proposed PLUTO's algorithm for automatically generating the OpenMP [41, 13, 9] code from a given serial C program targeting multi-cores. We describe PLUTO's algorithm as the following. Consider a loop nest with statements S_1, \dots, S_ℓ . Let ϕ_{S_i} be a one-dimensional affine transformation for statement S_i , for $i = 1 \dots \ell$. The sequence $\{\phi_{S_1}, \phi_{S_2}, \dots, \phi_{S_\ell}\}$ is called a *tiling hyperplane*. We say that $\{\phi_{S_1}, \phi_{S_2}, \dots, \phi_{S_\ell}\}$ is *legal* (or *statement-wise*) if the following holds for each dependence edge e from statement S_i to statement S_j :

$$\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) \geq 0, \quad (2.4)$$

where \vec{s} and \vec{t} are the vectors of loop iteration counters occurring in the indices of array references in S_i and S_j , respectively. A *cost function* for the dependence edge e is defined as the following affine form:

$$\delta_e(\vec{s}, \vec{t}) = \phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}).$$

Since the loop iteration counters themselves can be bounded by the affine functions of the parameters, one can always find an affine form $v(\vec{n})$ in the program parameters \vec{n} , that is, $v(\vec{n}) = \mathbf{u} \cdot \vec{n} + w$, where \mathbf{u} is a row-vector over \mathbb{Z} and $w \in \mathbb{Z}$. Thus, for all \vec{s}, \vec{t} , we have

$$\delta_e(\vec{s}, \vec{t}) \leq v(\vec{n}). \quad (2.5)$$

Given a statement S and its iteration counters \vec{x} , we define the affine transformation $\phi_S(\vec{x}) = \mathbf{c} \cdot \vec{x} + c_0$, where \mathbf{c} is a row-vector over \mathbb{Z} and $c_0 \in \mathbb{Z}$. Our unknowns are $\mathbf{u}, w, c_0, \mathbf{c}$. We determine them as the solution of a minimization problem defined as follows.

Assume that the polyhedron associated with the dependence edge e having p linear inequalities is given by $A_{e,k} \begin{pmatrix} \vec{s} \\ \vec{t} \\ \vec{n} \end{pmatrix} + b_{e,k} \geq 0$, where $A_{e,k}$ is a row-vector over \mathbb{Z} and $b_{e,k} \in \mathbb{Z}$ for $1 \leq k \leq p$. Then, applying the Farkas Lemma, we rewrite the legality in Equation 2.4 as

$$\phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}) = \mu_{e0} + \sum_k \mu_{ek} \left(A_{e,k} \begin{pmatrix} \vec{s} \\ \vec{t} \\ \vec{n} \end{pmatrix} + b_{e,k} \right), \quad 1 \leq k \leq p,$$

where $\mu_{e0}, \dots, \mu_{ep}$ are the Farkas multipliers, and the cost function in Equation 2.5 as

$$v(\vec{n}) - \delta_e(\vec{s}, \vec{t}) = v\lambda_{e0} + \sum_k \lambda_{ek} \left(A_{e,k} \begin{pmatrix} \vec{s} \\ \vec{t} \\ \vec{n} \end{pmatrix} + b_{e,k} \right), \quad 1 \leq k \leq p,$$

where $\lambda_{e0}, \dots, \lambda_{ep}$ are the Farkas multipliers.

We obtain constraints on $\mathbf{u}, w, c_0, \mathbf{c}$, from a process of identification between the multipliers μ and λ . Then, the best solution for the dependence edge e is the *lexicographic minimal solution*:

$$\text{minimize}_{<} \{\mathbf{u}, w, c_0, \mathbf{c}\}.$$

Recall that given two posets A and B , the *lexicographical order* on the Cartesian product $A \times B$ is defined as $(a, b) \leq (a', b')$ iff $a < a'$ or $(a = a' \text{ and } b \leq b')$. Solving this minimization problem can be handled by the parametric integer programming (PIP) [84] software.

Due to the fact that $v(\vec{n})$ gives a maximum delay to schedule the original program, if we could find an affine form with $\mathbf{u} = 0$ and $w > 0$, then we have a small delay to execute the statement. In the case $\mathbf{u} = 0$ and $w = 0$, there is no dependence in the transformed loop nest; thus the transformed loop nest can be executed in parallel. Once solutions of \mathbf{c} and c_0 to the affine transformation ϕ are obtained by PLUTO's algorithm, one can parallelize the `for`-loops iff $\mathbf{u} = 0$ and $w \geq 0$.

2.6 Solving systems of polynomial equations and inequalities

This section is an overview of the algorithmic tools used in this thesis for dealing with systems of polynomial equations and inequalities. We rely on the theory of *regular chains* and its implementation in the RegularChains library.

The notion of a *regular chain* (introduced independently in [70] by Kalkbrener, and in [82] by Yang & Zhang) is closely related to that of a triangular decomposition of a polynomial system. Broadly speaking, a *triangular decomposition*⁴ [22] of a polynomial system S is a set of simpler (in a precise sense) polynomial systems S_1, \dots, S_e such that

$$p \text{ is a solution of } S \Leftrightarrow \exists i : p \text{ is a solution of } S_i. \quad (2.6)$$

When the purpose is to describe all the solutions of S , whether their coordinates are real numbers or not, in which case S is said to be *algebraic*, those simpler systems are required to be regular chains⁵. If the coefficients of S are real numbers and if only the real solutions are required, in which case S is said to be *semi-algebraic*, then those real solutions can be obtained by a triangular decomposition into so-called *regular semi-algebraic systems*, a notion introduced in [22]. In both cases, each of these simpler systems has a triangular shape and remarkable properties, which justifies the terminology. We refer to [4, 25] for a formal presentation on the concepts of a regular chain and a triangular decomposition of a polynomial system. We recall, however, the concept of a regular semi-algebraic system since it is at the core of the present paper. A regular semi-algebraic system is a triple $[T, Q, P]$ where T is a regular chain, Q is a quantifier-free formula involving only the free variables of T and P is a set of polynomial inequalities; moreover $[T, Q, P]$ must satisfy the following properties.

- (i) Q defines a non-empty open set in the space of the free variables of T ,
- (ii) $[T, P]$ specializes well at any point⁶ defined by Q ,
- (iii) At any point α defined by Q , the specialized system $[T_\alpha, P_\alpha]$ admits at least one real solution β , in the sense that every polynomial in T_α is zero at β , and every polynomial in P_α is positive at β .

Consider semi-algebraic systems $[T_1, Q_1, P_1], \dots, [T_e, Q_e, P_e]$ forming a triangular decomposition of the polynomial system S . A consequence of the above relation (2.6) is that, if all the T_i have the same free variables, i.e. parameters, then the disjunction $Q_1 \vee \dots \vee Q_e$ defines the set of the parameter values for which the input system possesses real solutions. However, it is not necessary for all the T_i to have the same free variables and the example below given by (2.7) illustrates this fact.

An important property of any regular semi-algebraic system $[T, Q, P]$ is the fact that it is a parametrization of its zero set. Therefore, a triangular decomposition of a semi-algebraic system S decomposes the zero set of S into components, with each of them given by a parametric representation. This type of representation of the solutions of S is very useful to compute geometrical quantities such as dimension. As a first illustration let us consider the following semi-algebraic system

$$\begin{cases} (x-1)(y^2 + t^2) + (x-2)(y^2 - t) = 0 \\ (x-1)(x-2) = 0, \end{cases} \quad (2.7)$$

and solve it with `RealTriangularize` command of the `RegularChains` library, leading to the computations on Figure 2.11.

⁴http://en.wikipedia.org/wiki/Triangular_decomposition

⁵ More generally, a triangular decomposition into regular chains of a polynomial system S with coefficients in an arbitrary field K describes the solutions of S whose coordinates are in the algebraic closure of K .

⁶This means that at any point u defined by Q , the specialized set $T(u)$ is a squarefree regular chain with the same rank as T and each specialized polynomial $P_i(u)$ is invertible modulo $\langle T(u) \rangle$.

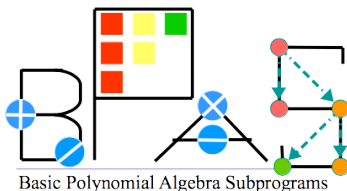
The solutions of the above are all (x, y, z) where x, y, z are complex numbers satisfying this equation. The output of `RealTriangularize` consists of 9 regular semi-algebraic systems for which the variables are ordered as $x > y > z$. The first regular semi-algebraic system represents a two-dimensional component. Indeed, it defines x as the solution of a parametric equation of degree 2, where y, z are regarded as parameters subject to an inequality (defined by the discriminant of the equation) which ensures the existence of two x -values for each valid (y, z) -value. The second regular semi-algebraic system represents a one-dimensional component: the two equations define (x, y) as functions of z , which is subject to various inequalities. Each of the other 7 regular semi-algebraic systems encodes a zero-dimensional component, that is, a finite set of points.

Triangular decompositions into regular semi-algebraic systems are an interesting representation of semi-algebraic sets for the following reasons. First, triangular decompositions into regular chains are a space-efficient encoding of algebraic sets. This fact is formally established in [42] and experimentally verified with the `RegularChains` library in [25]. Secondly, triangular decompositions into regular chains (or regular semi-algebraic systems) reveal important geometrical properties (dimensions of the irreducible components, fibration structure, etc.) of the input algebraic sets (or semi-algebraic sets), as illustrated by the previous examples. These can be used to design efficient algorithms for the operations manipulating (semi-)algebraic sets. For instance, performing set-theoretic operations (in particular, set theoretic difference) can be done very efficiently on both constructible sets and semi-algebraic sets as reported in [24] and [23], respectively. Last but not least, triangular decompositions and related techniques, such as dynamic evaluation, are well suited for supporting weaker solving specifications. Two examples of that are triangular decomposition of algebraic systems in the sense of Kalkbrener [70] and lazy triangular decomposition of semi-algebraic systems [22]. These types of decompositions provide a description of the “generic solutions” plus a continuation mechanism for obtaining the other solutions, if necessary. One major supporting argument for those decompositions is the existence of favorable algebraic complexity estimates. Both of them can indeed be computed in singly exponential time with respect to the number of variables, as established in [112] and [22], respectively.

Chapter 3

The Basic Polynomial Algebra Subprograms

The Basic Polynomial Algebra Subprograms (BPAS) provide arithmetic operations (multiplication, division, root isolation, etc.) for univariate and multivariate polynomials over prime fields or with integer, rational number or complex rational number coefficients. The code is mainly written in CilkPlus [10, 79] targeting multi-core processors. The current distribution focuses on dense polynomials and the sparse case is work in progress. A strong emphasis is put on adaptive algorithms as the library aims at supporting a wide variety of situations in terms of problem sizes and available computing resources. One of the purposes of the BPAS project is to take advantage of hardware accelerators in the development of polynomial system solvers. The BPAS library is publicly available in source at www.bpaslib.org.



We describe the design and specification in Section 3.1. In Section 3.2, we introduce the user interface of our library. In Sections 3.3 and 3.4, we demonstrate the core algorithms, like FFT and polynomial multiplication, and the experimental results, respectively. Applications to solve real root isolation and symbolic-numeric integration are summarized in Section 3.5. Appendix A shows sample code in the BPAS library.

This chapter is an extended version of [20] and contains joint work with Changbo Chen, Svyatoslav Covanov, Farnam Mansouri, Robert Moir, Marc Moreno Maza and Yuzhen Xie.

3.1 Design and specification

Inspired by the Basic Linear Algebra Subprograms (BLAS) [77], BPAS functionalities are organized into three levels. At Level 1, one finds basic arithmetic operations that are specific to a polynomial representation or specific to a coefficient ring. Examples of Level-1 operations are

multi-dimensional FFTs/TFTs and univariate real root isolation. At Level 2, arithmetic operations are implemented for all types of coefficients rings that BPAS supports (prime fields, ring of integers, field of rational numbers). Level 3 gathers advanced arithmetic operations taking as input a zero-dimensional regular chain, e.g. normal form of a polynomial, multivariate real root isolation.

Level 1 functions are highly optimized in terms of data locality and parallelism. In particular, the underlying algorithms are nearly optimal in terms of cache complexity [49]. This is the case, for instance, for our modular multi-dimensional FFTs/TFTs [94], modular dense polynomial arithmetic [93] and Taylor shift [28] algorithms.

At Level 2, users can choose between algorithms that either minimize work (at the possible expense of decreasing parallelism) or maximizes parallelism (at the possible expense of increasing work). For instance, five different integer polynomial multiplication algorithms are available, namely: Schönhage-Strassen [104], 8-way Toom-Cook [14], 4-way Toom-Cook [14], divide-and-conquer plain multiplication and the two-convolution method [21]. The first one has optimal work (i.e. algebraic complexity) but is purely serial due to the difficulties of parallelizing 1D FFTs on multi-core processors. The next three algorithms are parallelized but their parallelism is static, that is, independent of the input data size; these algorithms are practically efficient when both the input data size and the number of available cores are small, see [87] for details. The fifth algorithm relies on modular 2D FFTs which are computed by means of the row-column scheme; this algorithm delivers high scalability and can fully utilize the hardware on fat multi-core nodes.

Another example of Level 2 functionality is parallel Taylor shift computation for which four different algorithms are available: the two plain algorithms presented in [28], Algorithm (E) of [119] and an optimized version of Algorithm (F) of [119]. The first two are highly effective when both the input data size and the number of available cores are small. The third algorithm creates parallelism by means of a divide-and-conquer procedure and relies on polynomial multiplication; this approach is effective when 8-way Toom-Cook multiplication is selected. The fourth algorithm reduces a Taylor shift computation to a single polynomial multiplication; this latter approach outperforms the other three, as soon as the two-convolution multiplication dominates its counterparts, that is, when either input data size and the number of available cores become large.

This variety of parallel solutions leads, at Level 3, to adaptive algorithms which select appropriate Level 2 functions depending on available resources (number of cores, input data size). An example is parallel real root isolation. Many procedures for this purpose are based on a *subdivision scheme*. However, on many examples, this scheme exposes only a limited amount of opportunities for concurrent execution, see [28]. It is, therefore, essential to extract as much as parallelism from the underlying routines, such as Taylor shift computations.

3.2 User interface

Inspired by computer algebra systems like AXIOM [69] and Magma [16], the BPAS library makes use of type constructors [96] so as to provide genericity. For instance, `SparseUnivariatePolynomial` (SUP) can be instantiated over any BPAS ring. On the other hand, for efficiency consideration, certain polynomial type constructors, like `DistributedDenseMultivariateMod-`

ularPolynomial (DDMMP), are only available over finite fields in order to ensure that the data encoding a DDMMP polynomial consists only of consecutive memory cells. For the same efficiency consideration, the most frequently used polynomial rings, like DenseUnivariateIntegerPolynomial (DUZP) and DenseUnivariateRationalNumberPolynomial (DUQP) are primitive types. Consequently, DUZP and SUP<Integer> implement the same functionalities; however, the implementation of the former is further optimized. Figure 3.1 shows a subset of BPAS's tree of algebraic data structures. Each class with its name starting with "BPAS" is defined as an abstract class, while others are defined as concrete classes. Abstract classes are used to indicate the public members that their children must implement. Each concrete derived class overrides the pure virtual member functions of its parent(s).

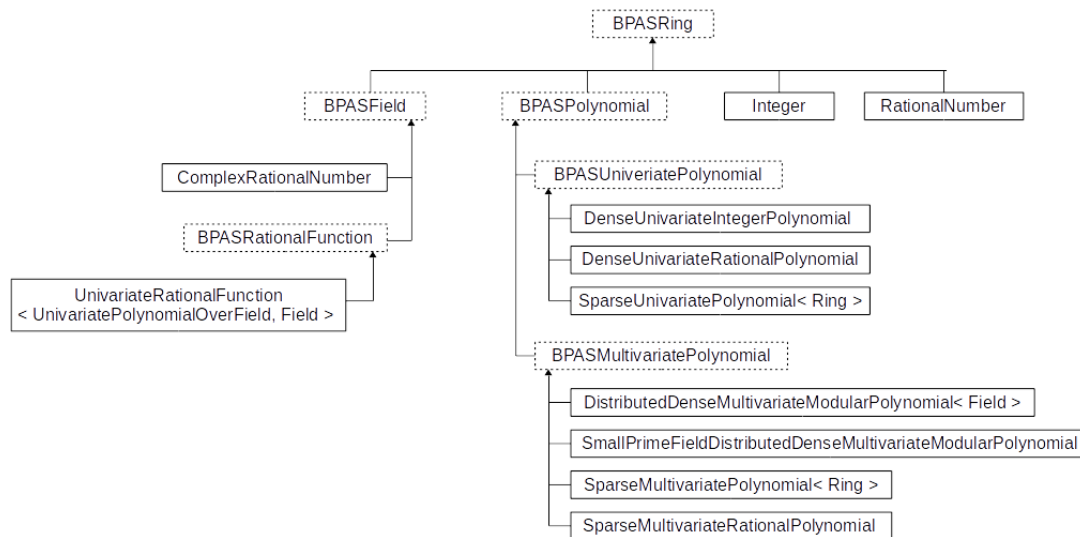


Figure 3.1: A subset of BPAS algebraic data structures

With the support of C++ operator overloading, we define the basic arithmetic operations '+', '-', '*' and '/' for each BPAS polynomial to execute addition, subtraction, multiplication and exact division, respectively. In addition, the insertion ('<<') operator for output streams (to the screen and files) is overloaded for each BPAS polynomial as well as each BPAS field. Particularly for BPAS univariate polynomials, we redefine shift operators '<<' and '>>' to represent multiplying and dividing, respectively, by a power of the variable. Several member functions, such as monicDivide, pseudoDivide, squareFree and gcd, are extended for BPAS univariate polynomials as well.

While the concrete class Integer (resp. RationalNumber) inherits from BPASRing and mpz_class (resp. mpq_class) (from the GMP library), arithmetic operations (addition, subtraction, multiplication and division) rely on the GMP library. Since the sparse univariate or multivariate polynomial can take an arbitrary ring as the coefficient ring, the programmer shall realize the coefficient ring with the pure virtual functions defined by BPAS ring class, such as isZero() and zero(). In order to identify the type of the coefficient ring in the sparse univariate polynomial, we add static attributes, such as characteristic and isPrimeField, to each BPAS ring, so that SUP<Ring> (and later SMP<Ring>) can invoke an efficient implementation of the functionality from those polynomial classes in a dense representation.

Moreover, we implement a `ComplexRationalNumber` class based on two rational numbers from the GMP library to represent the real part and the imaginary part, respectively. With this support as well as the definition of the BPAS field, we expand the BPAS ring with a univariate rational function so as to solve symbolic numeric integration. Moreover, the conversion between BPAS univariate polynomials, `Integer`, `RationalNumber` and `ComplexRationalNumber` can be done via the construction of each concrete class.

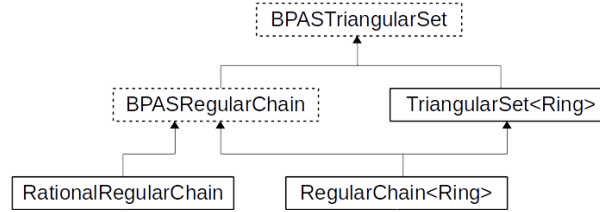


Figure 3.2: Another subset of BPAS algebraic data structures

Last but not least, BPAS is capable of isolating real roots given a regular chain over \mathbb{Q} . Figure 3.2 shows another subset of BPAS data structures to support triangular sets and regular chains of polynomial systems. BPAS also counts other classes, for instance, `Intervals`.

```

#include <bpas.h>

int main(int argc, char *argv[]) {
    int d = 4095;
    /* Univariate Integer Polynomial Multiplication */
    DUZP a(d+1), b(d+1);
    a.read("a_input.dat"); b.read("b_input.dat");
    DUZP c = a * b;
    std::cout << "c = " << c << std::endl;
    /* Real Root Isolation */
    DUQP p;
    p = (p + mpq_class(1) << d) + mpq_class(d); // Cn,d(x) = x^d + d
    Intervals boxes = p.realRootIsolate(mpq_class(1, 20));
    std::cout << "boxes = " << boxes << std::endl;
    /* Symbolic Numeric Integration */
    SparseUnivariatePolynomial<RationalNumber> f, g;
    f.one(); f.setVariableName("x");
    g = (polynomialParser("1+2*x+2*x^2"))^4;
    UnivariateRationalFunction<SparseUnivariatePolynomial<RationalNumber>,
                                RationalNumber> h(f, g);
    h.realSymbolicNumericIntegrate(53);

    return 0;
}

```

Figure 3.3: A snapshot of BPAS code

The snapshot of the BPAS code in Figure 3.3 first shows how two dense univariate polynomials over \mathbb{Z} are read from files and how their product is computed. Then, on the same code

fragment, a dense univariate polynomial over \mathbb{Q} is assigned by operators ‘+’ and ‘<<’ and its real roots are isolated. Finally, it has a polynomial parser of `SUP<RationalNumber>` and symbolic numeric integration of a univariate rational function.

3.3 Implementation techniques

Modular FFTs are at the core of asymptotically fast algorithms for dense polynomial arithmetic operations. A substantial body of code of the BPAS library is, therefore, devoted to the computation of one-dimensional and multi-dimensional FFTs over finite fields. In the current release, the characteristic of those fields is of machine word size, while larger characteristics are a work in progress.

The techniques used for the multi-dimensional FFTs are described in [94, 93] while those for one-dimensional FFTs are inspired by the design of the FFTW [48].

BPAS’ one-dimensional FFTs code is optimized in terms of cache complexity and register usage. To achieve this, the FFT of a vector of size n is computed in a divide-and-conquer manner until the vector size is smaller than a threshold, at which point FFTs are computed using a tiling strategy [124]. This threshold can be specified by the user through an environment variable `HTHRESHOLD` or determined automatically when installing the library. At compile time, this threshold is used to generate and optimize the code. For instance, the code of all FFTs of size less than or equal to `HTHRESHOLD` is decomposed into blocks (typically performing FFTs on 8 or 16 points) for which straight-line program (SLP) [5] machine code is generated. Instruction level parallelism (ILP) is carefully considered: vectorized instructions are explicitly used (SSE2, SSE4) and instruction pipeline usage is highly optimized. Other environment variables are available for the user to control different parameters in the code generation.

Table 3.1: One-dimensional modular FFTs: `Modpn` vs BPAS

Input size	<code>Modpn</code>	BPAS	Speedup
16777216	6.232	1.391	4.48
33554432	12.987	2.957	4.392
67108864	26.783	6.266	4.274
134217728	55.329	13.235	4.181
268435456	113.8	27.901	4.079

Table 3.1 compares running times (in sec. on Intel Xeon 5650) of one-dimensional modular FFTs computed by the `Modpn` library [80] and BPAS, both using serial C code in this case. The first column of Table 3.1 gives the size of the input vector; coefficients are in a prime field whose characteristic is a 57-bit prime.

Modular FFTs support the implementation of several algorithms performing dense polynomial arithmetic. As an example, we consider parallel multiplication of dense polynomials with integer coefficients by means of the *two-convolution method* [21] and which is illustrated on Figure 3.4. Given two univariate polynomials $a(y)$, $b(y)$ with integer coefficients, their product $c(y)$ is computed as follows.

(S1) Convert $a(y)$, $b(y)$ to bivariate integer polynomials $A(x, y)$, $B(x, y)$ s.t. $a(y) = A(\beta, y)$ and

- $b(y) = B(\beta, y)$ hold at $\beta = 2^M$, $K = \deg(A, x) = \deg(B, x)$, where M is essentially the maximum bit size of a coefficient in a and b .
- (S2) Consider $C^+(x, y) \equiv A(x, y) B(x, y) \bmod \langle x^K + 1 \rangle$ and $C^-(x, y) \equiv A(x, y) B(x, y) \bmod \langle x^K - 1 \rangle$. Compute $C^+(x, y)$ and $C^-(x, y)$ modulo machine-word primes so as to use modular 2D FFTs.
- (S3) Consider $C(x, y) = \frac{C^+(x, y)}{2} (x^K - 1) + \frac{C^-(x, y)}{2} (x^K + 1)$ and evaluate $C(x, y)$ at $x = \beta$, which finally gives $c(y) = a(y) b(y)$.

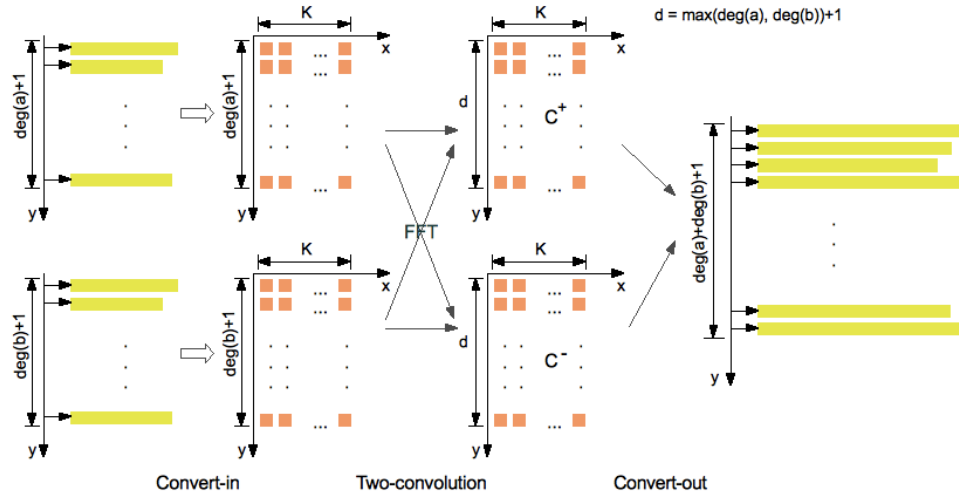


Figure 3.4: Multiplication scheme for dense univariate integer polynomials

The conversions from the univariate polynomials $a(y)$, $b(y)$ to the bivariate polynomials $A(x, y)$, $B(x, y)$ in Step (S1) as well as the conversions from the bivariate polynomials $C^+(x, y)$ and $C^-(x, y)$ in Step (S3) require only additions and shift operations on machine words. Moreover, the polynomials $C^+(x, y)$ and $C^-(x, y)$ are reconstructed from their modular images (in practice two modular images are sufficient) within Step (S3). Consequently, the data produced by 2D FFT computations is converted in a *single pass* into the final result $c(y)$. Similarly the bivariate polynomials $A(x, y)$, $B(x, y)$ are obtained from $a(y)$, $b(y)$ (here again by means of additions and shift operations on machine words) in a single pass. Since BPAS' 2D FFT computations are optimal in terms of cache complexity [93], the whole multiplication procedure is optimal for that same complexity measure. Last but not least, BPAS' 2D FFTs are computed by the row-column scheme which provides lots of parallelism with limited overheads on multi-core architectures. As a result, our multiplication code, based on this two-convolution method scales well on multi-cores as illustrated hereafter.

3.4 Experimental evaluation

As mentioned above, one of the main purposes of the BPAS library is to take advantage of hardware accelerators and support the implementation of polynomial system solvers. With this

goal, polynomial multiplication plays a central role. Moreover, both sparse and dense representations are important. Indeed, input polynomial systems are often sparse while many algebraic transactions, like substitution, tend to densify data. Parallel sparse polynomial arithmetic has been studied by Gastineau and Laskar in [50] and by Monagan and Pearce in [91].

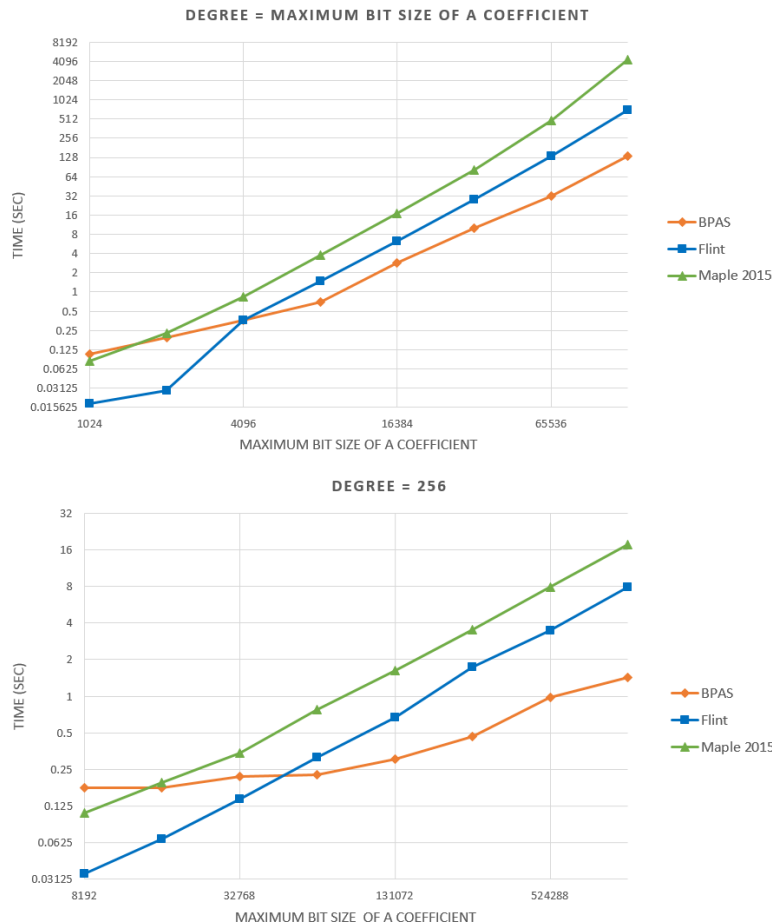


Figure 3.5: Dense integer polynomial multiplication: BPAS vs FLINT vs MAPLE

To the best of our knowledge, BPAS is the first publicly available library for parallel dense integer polynomial arithmetic. For this reason, we compare BPAS' parallel dense polynomial multiplication against its state-of-the-art counterpart implementation in FLINT 2.5.2 and MAPLE 2015. In Figure 3.5, the input of each test case is a pair of polynomials of degree d where each coefficient has bit size N . Two plots are provided: one for which $d = N$ holds and one for which d is much smaller than N .

The BPAS library is implemented with the multithreaded language CilkPlus [79] and we compile our code with the CilkPlus branch of GCC¹. Our experimental results were obtained on an 48-core AMD Opteron 6168, running at 900Mhz with 256 GB of RAM and 512KB of L2 cache. Figure 3.6 shows that using an interactive process viewer for Unix systems, namely `htop`, BPAS can use all available cores when multiplying two large integer polynomials.

¹<http://gcc.gnu.org/svn/gcc/branches/cilkplus/>

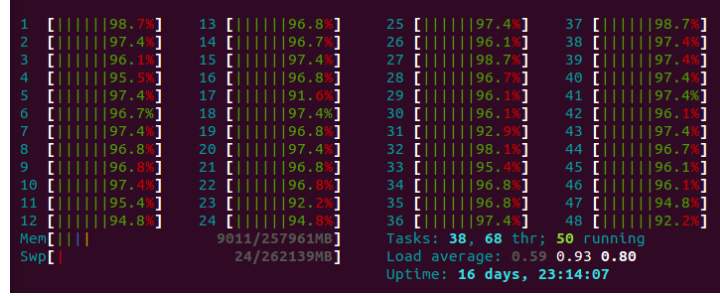


Figure 3.6: The htop screenshot of multiplying two large integer polynomials in BPAS

Table 3.2 shows that the work overhead (measured by Cilkview, the performance analysis tool of CilkPlus) of the BPAS method w.r.t. to a method based on Schönage & Strassen algorithm (KS) is only around 2, whereas BPAS provides a large amount of parallelism, see the last column.

Table 3.2: Cilkview analysis of BPAS and KS (* shows the number of instructions)

Size	Work(KS)*	Work(BPAS)*	Span(BPAS)*	$\frac{\text{Work(BPAS)}}{\text{Span(BPAS)}}$	$\frac{\text{Work(BPAS)}}{\text{Work(KS)}}$
2048	795,549,545	1,364,160,088	41,143,119	33.16	1.715
4096	4,302,927,423	5,663,423,709	96,032,325	58.97	1.316
8192	16,782,031,611	23,827,123,688	292,735,521	81.39	1.420
16384	63,573,232,166	100,688,072,711	1,017,726,160	98.93	1.584
32768	269,887,534,779	425,149,529,176	3,804,178,563	111.76	1.575

3.5 Application

Turning to parallel univariate real root isolation, given a *squarefree* univariate polynomial $f(x) = a_d x^d + \dots + a_1 x + a_0 \in \mathbb{Q}[x]$, we compute the real roots of $f(x) = 0$. We represent real roots as a list of *pairwise disjoint intervals* $[a_1, b_1], \dots, [a_e, b_e]$ with rational number endpoints, such that (1) each $[a_i, b_i]$ contains one and only one real root of $f(x)$; (2) if $a_i = b_i$, the real root $x_i = a_i (b_i)$; (3) otherwise, the real root $a_i < x_i < b_i$ ($f(x)$ does not vanish at either endpoint).

In [28], Chen, Moreno Maza and Xie presents a parallel algorithm (CMY) directly using the Vincent-Collins-Akritas (VCA) [32] algorithm for univariate real root isolation targeting multi-cores. Authors observe that the most consuming time is the operation of so-called *Taylor shift* [119], that is, the map $f(x) \mapsto f(x + 1)$. Among six classic Taylor shift algorithms reported in [119], the authors in [28] parallelize the Horner's method, which represents the computation in a Pascal's triangle. This method has an optimal cache complexity [49] but runs in quadratic time.

BPAS performs the Taylor shift operation by means of a *divide and conquer* method, that is, the Algorithm (E) in [119]. It reduces calculations to integer polynomial multiplications in large degrees and continues to use the algorithm of [28] in small degrees. This adaptive

Taylor shift algorithm combines FFT-based arithmetic (via Algorithm (E)) and plain arithmetic (via [28]).

We run two parallel univariate real root algorithms, BPAS and CMY [28], which are both implemented in CilkPlus, against MAPLE 18 serial `realroot` command. Table 3.3 shows the running times (in sec.) of four well-known test problems, including Cnd, Chebycheff, Laguerre and Wilkinson. Moreover, for each test problem, the degree of the input polynomial varies in a range. The results reported in Table 3.3 show that integrating parallel integer polynomial multiplication into our real root isolation code has substantially improved the performance of the latter.

Table 3.3: Univariate real root isolation running times (in secs.) for four examples

	Size	BPAS	CMY [28]	<code>realroot</code>	#Roots
Cnd	32768	18.141	125.902	816.134	1
	65536	66.436	664.438	7,526.428	1
Chebycheff	2048	608.738	594.82	1,378.444	2047
	4096	8,194.06	10,014	35,880.069	4095
Laguerre	2048	1,336.14	1,324.33	3,706.749	2047
	4096	20,727.9	23,605.7	91,668.577	4095
Wilkinson	2048	630.481	614.94	1,031.36	2047
	4096	9,359.25	10,733.3	26,496.979	4095

Consider a system of n -variable polynomials over \mathbb{Q} with n equations:

$$\begin{cases} f_1(x_1, x_2, \dots, x_n) = 0, \\ f_2(x_1, x_2, \dots, x_n) = 0, \\ \vdots \\ f_n(x_1, x_2, \dots, x_n) = 0. \end{cases}$$

We compute the common real roots of $f_i(x_1, x_2, \dots, x_n) = 0$ for each $1 \leq i \leq n$. In this case, each real root is the Cartesian product of n intervals (one for each of the coordinates x_1, \dots, x_n) with two endpoints $[a, b] \in \mathbb{Q}$.

After applying the `Triangularize` algorithm [25] to the above system, one obtains a description of the solution set in the form of a list of *regular chains*. A regular chain is a particular type of polynomial systems with a triangular shape:

$$\begin{cases} g_n(x_1, x_2, \dots, x_n) = 0, \\ \vdots \\ g_2(x_1, x_2) = 0, \\ g_1(x_1) = 0. \end{cases} \quad (3.1)$$

In addition, for all $2 \leq i \leq n$, the leading coefficient of g_i does not vanish at any root (x_1, \dots, x_{i-1}) of $g_1 = \dots = g_{i-1} = 0$.

One can write each multivariate polynomial as

$$g_i = h_\ell(x_1, \dots, x_{i-1}) x_i^\ell + \dots + h_1(x_1, \dots, x_{i-1}) x_i + h_0(x_1, \dots, x_{i-1}),$$

where $h_\ell, h_{\ell-1}, \dots, h_1, h_0 \in \mathbb{Q}[x_1, \dots, x_{i-1}]$ hold and ℓ is the degree of g_i w.r.t. its leading variable x_i .

For each isolation box $X = ([a_1, b_1], \dots, [a_{i-1}, b_{i-1}])$ of a real root of $g_1 = \dots = g_{i-1} = 0$, where $[a_j, b_j] \in \mathbb{Q}$ for $1 \leq j \leq i-1$, we evaluate g_i by $h_\ell(X)x_i^\ell + \dots + h_0(X)$, using the interval arithmetic [126]: $[\alpha, \beta] + [\gamma, \delta] = [\alpha + \gamma, \beta + \delta]$ and $[\alpha, \beta] \times [\gamma, \delta] = [\min(\theta), \max(\theta)]$, where $\theta = \{\alpha\gamma, \alpha\delta, \beta\gamma, \beta\delta\}$ and $[\alpha, \beta], [\gamma, \delta]$ are intervals.

Then we obtain

$$g_i(X, x_i) = [c_\ell, d_\ell]x_i^\ell + \dots + [c_0, d_0], \text{ where } [c_j, d_j] \in \mathbb{Q}, \text{ for } 0 \leq j \leq \ell.$$

Univariate polynomials $\underline{g}_i = c_\ell x_i^\ell + \dots + c_0$, and $\bar{g}_i = d_\ell x_i^\ell + \dots + d_0$ are defined as the *sleeve bound polynomials* [83] of g_i in X . For $x_i > 0$, we have

$$\underline{g}_i(x_i) \leq g_i(X, x_i) \leq \bar{g}_i(x_i).$$

Now, we can apply the univariate real root isolation routine to compute the real roots of $\underline{g}_i(x_i)$ and $\bar{g}_i(x_i)$, so as to obtain two lists of real root intervals, namely $List_1$ and $List_2$, respectively. The sequence of the intervals from $List_1$ and $List_2$ is *matchable*, with respect to a range value defined by the user, whenever both the following conditions are satisfied (see Figure 3.7 for an example):

1. The sequence of the real root intervals from $List_1$ and $List_2$ takes one of the following forms ('1' denotes an interval belonging to $List_1$ and '2' denotes an interval of $List_2$):
 - (a) 1, 2, 2, 1, 1, 2, 2, 1, \dots , 1, 2, 2, 1;
 - (b) 1, 2, 2, 1, 1, 2, 2, 1, \dots , 1, 2, 2, 1, 1, 2;
 - (c) 2, 1, 1, 2, 2, 1, 1, 2, \dots , 2, 1, 1, 2;
 - (d) 2, 1, 1, 2, 2, 1, 1, 2, \dots , 2, 1, 1, 2, 2, 1.
 2. In each interval from $List_1$ and $List_2$, the considered polynomial $g_i(X, x_i)$ is *monotonic*.
- Once such a matchable sequence of intervals is obtained, we say the real roots of g_i are found.

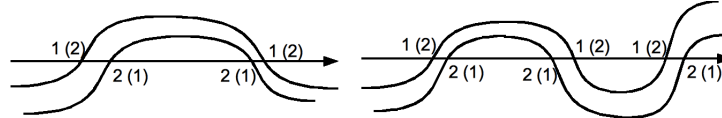


Figure 3.7: An example of matchable interval lists

For each real root isolation box $X = x_1, \dots, x_{i-1}$, we parallelize the computation of isolating x_i of $g_i(X, x_i)$ by using the algorithm described above. In general, after `Triangularize` an n -variable polynomial system, a list of regular chains is obtained. Thus, we can parallelize the real root isolation for each regular chain. Given a regular chain following the format of Equation 3.1, one can use `realRootIsolate` from the `RationalRegularChain` class in the BPAS library to isolate real roots of polynomial systems.

We compare our implementation with MAPLE 17 `RealRootIsolate` and `Isolate` written in C. In Table 3.4, we collect the running times (in secs.) and speedup factors. We observe that for 12 of 17 examples, BPAS outperforms the others and that the speedup factor varies from 1 to 16. To the best of our knowledge, this is the first parallel implementation of multivariate real root isolation.

Table 3.4: Running times (in secs.) of multivariate real root isolation: BPAS vs MAPLE 17 RealRootIsolate vs C (with MAPLE 17 interface) Isolate

Example	BPAS (CilkPlus)	RealRootIsolate (RegularChains, MAPLE)	Isolate (Gröbner bases, C)	Speedup factors
4-Body-Homog	0.402	0.608	0.382	
Arnborg-Lazard	0.146	0.299	0.066	
Caprasse	0.018	0.14	0.154	7.778
Circles	0.051	0.894	0.814	15.961
Cyclic-5	0.021	0.147	0.206	9.810
Czapor-Geddes-Wang	0.2	0.135	0.184	
D2v10	0.029	0.075	177.999	2.586
D4v5	0.037	0.044	49.09	1.189
Fabfaux	0.192	0.231	0.071	
Katsura-4	0.171	0.416	0.044	
L-3	0.02	0.252	0.12	6.0
Neural-Network	0.029	0.332	0.131	4.517
R-6	0.014	0.048	20.612	3.429
Rose	0.026	0.336	0.599	12.923
Takeuchi-Lu	0.027	0.16	0.031	1.148
Wilkinsonxy	0.023	0.165	0.046	2.0
Nld-10-3	1.249	8.993	707.334	7.20

Furthermore, for symbolic-numeric integration of symbolic functions focusing on rational functions, we rely on BPAS polynomial arithmetic operations (multiplication, division, root isolation, etc.) for univariate polynomials with integer, rational number or complex rational number coefficients. Our method proceeds by using the Lazard-Rioboo-Trager algorithm [78], while it relies on efficient numerical computation of isolating polynomial roots followed by symbolic post-processing. The symbolic computation is done within BPAS, while the numerical computation is done via using a highly optimized multiprecision rootfinding package, namely MPSOLVE.

The implementation of our algorithm is integrated in the BPAS library; it can be called through the `realSymbolicNumericIntegrate` method of the `UnivariateRationalFunction` template class. The following output formats are available: approximate (either floating point number or rational number) and symbolic expression (in either MAPLE or MATLAB syntax); see Figure 3.8 for a combination of floating point and MAPLE output formats.

```

-
/
|          1/16
| ----- dx =
| 1/16+1/2*x+2*x^2+5*x^3+17/2*x^4+10*x^5+8*x^6+4*x^7+x^8
-
(0.0208333+0.0416667*x)/(0.125+0.75*x+2.25*x^2+4*x^3+4.5*x^4+3*x^5+x^6) +
(0.104167+0.208333*x)/(0.25+x+2*x^2+2*x^3+x^4) + (0.625+1.25*x)/(0.5+x+x^2)
+ 2.5*arctan(1+2*x)
-----
realSymbolicNumericIntegrate runtime: 0.008 s

```

Figure 3.8: A sample output of `realSymbolicNumericIntegrate`

Chapter 4

A Many-Core Machine Model for Designing Algorithms with Minimum Parallelism Overheads

We present a model of multithreaded computation with an emphasis on estimating the parallelism overheads of programs written for modern many-core architectures. We establish a Graham-Brent theorem so as to estimate execution time of programs running on a given number of streaming multiprocessors. We evaluate the benefits of our model on fundamental algorithms from scientific computing. For three case studies, our model is used to minimize parallelism overheads by determining an appropriate value range for a given program parameter. For other cases, our model is used to compare different algorithms solving the same problem. In each case, the studied algorithms were implemented and the results of their experimental comparison are coherent with the theoretical analysis based on our model.

This chapter is an extended version of [61] and contains joint work with Sardar Anisul Haque and Marc Moreno Maza.

4.1 Introduction

Designing efficient algorithms targeting hardware accelerators (multi-core processors, graphics processing units (GPUs) and field-programmable gate arrays) creates major challenges for computer scientists. A first difficulty is to define models of computation retaining the computer hardware characteristics that have a dominant impact on program performance. In addition to specifying the appropriate complexity measures, those models must consider the relevant parameters characterizing the abstract machine executing the algorithms to be analyzed. A second difficulty is, for a given model of computation, to combine its complexity measures so as to determine the “best” algorithm among different possible solutions to a given algorithmic problem.

In the fork-join concurrency model [11], two complexity measures, the work T_1 and the span T_∞ , and one machine parameter, the number P of processors, are combined into a running time estimate, namely the Graham-Brent theorem [11, 53], which states that the running time T_P on P processors satisfies $T_P \leq T_1/P + T_\infty$. A refinement of this theorem supports the imple-

mentation (on multi-core architectures) of the parallel performance analyzer Cilkview [63]. In this context, the running time T_P is bounded in expectation by $T_1/P + 2\delta\widehat{T}_\infty$, where δ is a constant (called the *span coefficient*) and \widehat{T}_∞ is the burdened span, which captures parallelism overheads due to scheduling and synchronization.

The well-known PRAM (parallel random-access machine) model [110, 51] has also been enhanced [1] so as to integrate communication delay into the computation time. However, a PRAM abstract machine consists of an unbounded collection of RAM processors, whereas a many-core GPU holds a collection of streaming multiprocessors (SMs). Hence, applying the PRAM model to GPU programs fails to capture all the features (and thus the impact) of data transfer between the SMs and the global memory of the device.

Ma, Agrawal and Chamberlain [85] introduce the TMM (Threaded Many-core Memory) model, which retains many important characteristics of GPU-type architectures as machine parameters, like memory access bandwidth and hardware limit on the number of threads per core. In TMM analysis, the running time of an algorithm is estimated by choosing the maximum quantity among work, span and the number of memory accesses. Such running time estimates depend on the machine parameters. Hong and Kim [67] present an analytical model to predict the execution time of an actual GPU program. No abstract machine is defined in this case. Instead, a few metrics are used to estimate the CPI (cycles per instruction) of the considered program.

Many works, such as [86, 81], targeting code optimization and performance prediction of GPU programs are related to our work. However, these papers do not define an abstract model in support of the analysis of algorithms.

In this chapter, we propose a many-core machine (MCM) model with two objectives: (1) tuning program parameters to minimize parallelism overheads of algorithms targeting GPU-like architectures, and (2) comparing different algorithms independently of the targeted hardware device. In the design of this model, we insist on the following features:

1. *Two-level DAG programs.* Defined in Section 4.2, they capture the two levels of parallelism (fork-join and single instruction, multiple data) of heterogeneous programs (like a CilkPlus program using `#pragma simd` [100] or a CUDA program with the so-called dynamic parallelism [37]).
2. *Parallelism overhead.* We introduce this complexity measure in Section 4.2.3 with the objective of capturing communication and synchronization costs.
3. *A Graham-Brent theorem.* We combine three complexity measures (work, span and parallelism overhead) and one machine parameter (data transfer throughput) in order to estimate the running time of an MCM program on P streaming multiprocessors, see Theorem 4.2.1. However, as we shall see through a case study series, this machine parameter has no influence on the comparison of algorithms.

Our model extends both the fork-join concurrency and PRAM models, with an emphasis on parallelism overheads resulting from communication and synchronization.

We sketch below how, in practice, we use this model to tune a program parameter so as to minimize parallelism overheads of programs targeting many-core GPUs. Consider an MCM program \mathcal{P} , that is, an algorithm expressed in the MCM model. Assume that a program parameter s (like the number of threads per thread-block running on an SM) can be arbitrarily chosen within some range \mathcal{S} while preserving the specifications of \mathcal{P} . Let s_0 be a particular value of s

that corresponds to an instance \mathcal{P}_0 of \mathcal{P} , which, in practice, is seen as an initial version of the algorithm to be optimized.

We consider the ratios of work, span and parallelism overhead given by $W_{\mathcal{P}_0}/W_{\mathcal{P}}$, $S_{\mathcal{P}_0}/S_{\mathcal{P}}$ and $O_{\mathcal{P}_0}/O_{\mathcal{P}}$. Assume that, when s varies within \mathcal{S} , the work and span ratios stay within $O(s)$ (in fact, $\Theta(1)$ is often the case), but the ratio of the parallelism overhead reduces by a factor in $\Theta(s)$. Thereby, we determine a value $s_{\min} \in \mathcal{S}$ maximizing the parallelism overhead ratio. Next, we use our version of the Graham-Brent theorem (more precisely, Corollary 4.2.2) to check whether the upper bound for the running time of $\mathcal{P}(s_{\min})$ is less than that of $\mathcal{P}(s_o)$. If this holds, we view $\mathcal{P}(s_{\min})$ as a solution of our problem of algorithm optimization (in terms of parallelism overheads).

To evaluate the benefits of our model, we applied it successfully to six fundamental algorithms in scientific computing, see Sections 4.3 to 4.6. These six algorithms are the Euclidean algorithm, the Cooley & Tukey and Stockham fast Fourier transform algorithms, the plain and FFT-based univariate polynomial multiplication algorithms, and radix sort [103]. The former five algorithms are implemented in CUDA and publicly available with benchmarking scripts from <http://www.cumodp.org/>.

Following the strategy described above for algorithm optimization, our model is used to tune a program parameter in the case of the Euclidean algorithm, the plain multiplication algorithm and radix sort. Next, our model is used to compare the two fast Fourier transform algorithms and then the two univariate polynomial multiplication algorithms. In each case, work, span and parallelism overhead are evaluated so as to obtain running time estimates via our version of the Graham-Brent theorem and then select a proper algorithm.

4.2 A many-core machine model

The model of parallel computations presented in this Chapter aims at capturing communication and synchronization overheads of programs written for modern many-core architectures. One of our objectives is to optimize algorithms by techniques like reducing redundant memory accesses. The reason for this optimization is that, on actual GPUs, global memory latency is approximately 400 to 800 clock cycles. This memory latency, when not properly taken into account, may have a dramatically negative impact on program performance. Another objective of our model is to compare different algorithms targeting implementation on GPUs without taking hardware parameters into account.

As specified in Sections 4.2.1 and 4.2.2, our many-core machine (MCM) model retains many of the characteristics of modern GPU architectures and programming models, like CUDA or OpenCL. However, in order to support algorithm analysis with an emphasis on parallelism overheads, as defined in Section 4.2.3 and 4.2.4, the MCM abstract machines admit a few simplifications and limitations with respect to actual many-core devices.

4.2.1 Characteristics of the abstract many-core machines

Architecture. An MCM abstract machine shown in Figure 4.1 possesses an unbounded number of *streaming multiprocessors* (SMs), which are all identical. Each SM has a finite number of processing cores and a fixed-size private memory. An MCM machine has a two-level memory

hierarchy, comprising an unbounded global memory with high latency and low throughput and fixed size private memories with low latency and high throughput.

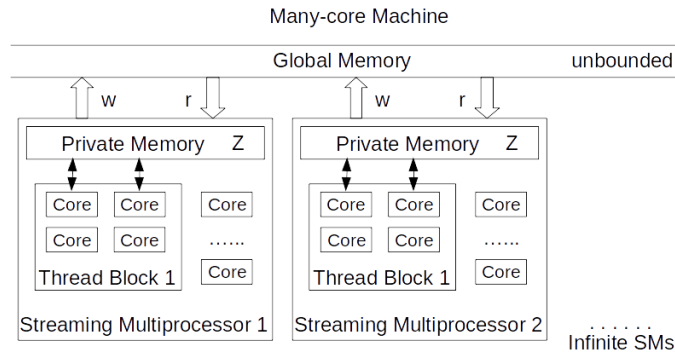


Figure 4.1: Overview of an abstract many-core machine

Programs. An MCM *program* is a directed acyclic graph (DAG), whose vertices are kernels (defined hereafter) and edges indicate serial dependencies; moreover, this DAG is similar to the instruction stream DAGs of the fork-join concurrency model. A *kernel* is an SIMD (single instruction, multiple data) program capable of branches and decomposed into a number of thread-blocks. Each *thread-block* is executed by a single SM, and each SM executes a single thread-block at a time. Similar to a CUDA program, an MCM program specifies for each kernel the number of thread-blocks and the number of threads per thread-block. Figure 4.2 depicts the different types of components of an MCM program.

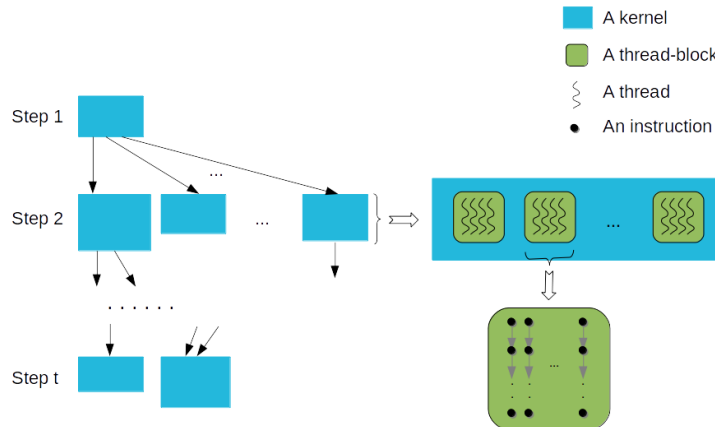


Figure 4.2: Overview of a many-core machine program

Scheduling and synchronization. At run time, an MCM machine schedules thread-blocks (from the same or different kernels) onto SMs, based on the dependencies specified by the edges of the DAG and the hardware resources required by each thread-block. Threads within a thread-block can cooperate with each other via the private memory of the SM running the thread-block.

Meanwhile, thread-blocks interact with each other via the global memory. In addition, threads within a thread-block are executed physically in parallel by an SM. Moreover, the programmer cannot make any assumptions on the order in which thread-blocks of a given kernel are mapped to the SMs. Hence, an MCM program runs correctly on any fixed number of SMs.

Memory access policy. All threads of a given thread-block can access simultaneously any memory cell of the private memory or the global memory: read/write conflicts are handled by the CREW (concurrent read, exclusive write) policy. However, we assume that read/write requests to the global memory by two different thread-blocks cannot be executed simultaneously. In case of simultaneous requests, one thread-block is chosen randomly and served first, then the other is served.

Toward analyzing program performance, we define two *machine parameters*:

U : time (expressed in clock cycles) to transfer one machine word between the global memory and the private memory of any SM; hence, we have $U > 0$, and

Z : size (expressed in machine words) of the private memory of any SM, which sets up an upper bound on several program parameters.

The private memory size Z sets several characteristics and limitations of an SM and, thus, of a thread-block. Indeed, each of the following quantities is at most equal to Z : the number of threads of a thread-block and the number of words in a data transfer between the global memory and the private memory. The quantity $1/U$ is a throughput measure and has the following property. If α and β are the maximum numbers of words, respectively, read and written to the global memory by one thread of a thread-block B , and ℓ is the number of threads per thread-block, then the total time T_D spent in data transfer between the global memory and the private memory of an SM executing B satisfies:

$$T_D \leq \begin{cases} (\alpha + \beta) U, & \text{if coalesced accesses occur, or} \\ \ell (\alpha + \beta) U, & \text{otherwise.} \end{cases} \quad (4.1)$$

On actual GPU devices, some hardware characteristics may reduce data transfer time, for instance, fast context switching between warps executed by an SM. Other hardware characteristics, like partition camping [102], may increase data transfer time. As an abstract machine, the MCM aims at capturing either the best or the worst scenario for data transfer time of a thread-block, which leads us to Relation (4.1).

Relation (4.1) calls for another comment. One could expect the introduction of a third machine parameter, say V , which would be the time to execute one *local operation* (arithmetic operation, read/write in the private memory), such that, if σ is the maximum number of local operations performed by one thread of a thread-block B , then the total time T_A spent in local operations by an SM executing B would satisfy

$$T_A \leq \sigma V. \quad (4.2)$$

Therefore, for the total running time T of the thread-block B , we would have

$$T = T_A + T_D \leq \sigma V + \epsilon (\alpha + \beta) U,$$

where ϵ is either 1 or ℓ . Instead of introducing this third machine parameter V , we let $V = 1$. Thus, U can be understood as the ratio of the time to transfer a machine word to the time to execute a local operation.

4.2.2 Many-core machine programs

Recall that each MCM program \mathcal{P} is a DAG $(\mathcal{K}, \mathcal{E})$, called the *kernel DAG* of \mathcal{P} , where each node $K \in \mathcal{K}$ represents a kernel, and each edge $E \in \mathcal{E}$ records the fact that a kernel call must precede another kernel call. In other words, a kernel call can be executed once all its predecessors in the DAG $(\mathcal{K}, \mathcal{E})$ have completed their execution.

Synchronization costs. Recall that each kernel decomposes into thread-blocks and that all threads within a given kernel execute the same serial program, but with possibly different input data. In addition, all threads within a thread-block are executed physically in parallel by an SM. It follows that the MCM kernel code needs no synchronization statement. Consequently, the only form of synchronization taking place among the threads executing a given thread-block is implied by code divergence [58]. This latter phenomenon can be seen as parallelism overhead. Furthermore, an MCM machine handles code divergence by eliminating the corresponding conditional branches via code replication [106]; thereby, the corresponding cost will be captured by the complexity measures (work, span and parallelism overhead) of the MCM model.

Scheduling costs. Since an MCM abstract machine has infinitely many SMs and since the kernel DAG defining an MCM program \mathcal{P} is assumed to be known when \mathcal{P} starts to execute, scheduling \mathcal{P} 's kernels onto the SMs can be done in time $O(\Gamma)$, where Γ is the total length of \mathcal{P} 's kernel code. Thus, we neglect those costs in comparison to the costs of data transfer between SMs' private memories and the global memory. We also note that assuming that the kernel DAG is known when \mathcal{P} starts to execute allows us to focus on parallelism overheads resulting from this data transfer.

Extending MCM machines to program DAGs unfolding dynamically at run time and integrating the resulting scheduling costs are a work in progress. This key observation helps understand the complexity measures introduced in Section 4.2.3.

Thread-block DAG. Since each kernel of the program \mathcal{P} decomposes into finitely many thread-blocks, we map \mathcal{P} to a second graph, called the *thread-block DAG* of \mathcal{P} , whose vertex set $\mathcal{B}(\mathcal{P})$ consists of all thread-blocks of the kernels of \mathcal{P} , such that (B_1, B_2) is an edge if B_1 is a thread-block of a kernel preceding the kernel of the thread-block B_2 in \mathcal{P} . This second graph defines two important quantities:

$N(\mathcal{P})$: number of vertices in the thread-block DAG of \mathcal{P} , and

$L(\mathcal{P})$: critical path length (where length of a path is the number of edges in that path) in the thread-block DAG of \mathcal{P} .

4.2.3 Complexity measures for the many-core machine model

Consider an MCM program \mathcal{P} given by its kernel DAG $(\mathcal{K}, \mathcal{E})$. Let $K \in \mathcal{K}$ be a kernel of \mathcal{P} and B be a thread-block of K . We define the *work* of B , denoted by $W(B)$, as the total number of local operations performed by all threads of B . We define the *span* of B , denoted by $S(B)$, as the maximum number of local operations performed by a thread of B . As before, let α and β be the maximum numbers of words read and written (from the global memory) by a thread of B , and ℓ be the number of threads per thread-block. Then, we define the *overhead* of B , denoted

by $O(B)$, as

$$\begin{aligned} &(\alpha + \beta) U, \text{ if memory accesses can be coalesced or} \\ &\ell (\alpha + \beta) U, \text{ otherwise.} \end{aligned} \tag{4.3}$$

Next, the *work* (resp. *overhead*) $W(K)$ (resp. $O(K)$) of the kernel K is the sum of the works (resp. overheads) of its thread-blocks, while the *span* $S(K)$ of the kernel K is the maximum of the spans of its thread-blocks. We consider now the entire program \mathcal{P} . The *work* $W(\mathcal{P})$ of \mathcal{P} is defined as the total work of all its kernels.

$$W(\mathcal{P}) = \sum_{K \in \mathcal{K}} W(K).$$

Regarding the graph $(\mathcal{K}, \mathcal{E})$ as a weighted-vertex graph, where the weight of a vertex $K \in \mathcal{K}$ is its span $S(K)$, we define the weight $S(\gamma)$ of any path γ from the first executing kernel to a terminal kernel (that is, a kernel with no successors in \mathcal{P}) as $S(\gamma) = \sum_{K \in \gamma} S(K)$. Then, we define the *span* $S(\mathcal{P})$ of \mathcal{P} as

$$S(\mathcal{P}) = \max_{\gamma} S(\gamma),$$

the longest path, counting the weight (span) of each vertex (kernel), in the kernel DAG. Finally, we define the *overhead* $O(\mathcal{P})$ of the program \mathcal{P} as the total overhead of all its kernels.

$$O(\mathcal{P}) = \sum_{K \in \mathcal{K}} O(K).$$

Observe that, according to Mirsky's theorem [90], the number π of parallel steps in \mathcal{P} (which form a partition of \mathcal{K} into anti-chains in the DAG $(\mathcal{K}, \mathcal{E})$ regarded as a partially ordered set) is greater or equal to the maximum length of a path in $(\mathcal{K}, \mathcal{E})$ from the first executing kernel to a terminal kernel.

4.2.4 A Graham-Brent theorem with parallelism overhead

Theorem 4.2.1 *The running time $T_{\mathcal{P}}$ of the program \mathcal{P} executed on P SMs satisfies the inequality: $T_{\mathcal{P}} \leq (N(\mathcal{P})/P + L(\mathcal{P}))C(\mathcal{P})$, where $C(\mathcal{P}) = \max_{B \in \mathcal{B}(\mathcal{P})} (S(B) + O(B))$.*

The proof is similar to that of the original result [11, 53], while the proof of the following corollary follows from Theorem 4.2.1 and from the fact that costs of scheduling thread-blocks onto SMs are neglected.

Corollary 4.2.2 *Let K be the maximum number of thread-blocks along an anti-chain of the thread-block DAG of \mathcal{P} . Then the running time $T_{\mathcal{P}}$ of the program \mathcal{P} satisfies:*

$$T_{\mathcal{P}} \leq (N(\mathcal{P})/K + L(\mathcal{P}))C(\mathcal{P}). \tag{4.4}$$

The analysis of the running time estimate is based on the thread-block DAG. In Figure 4.3, we show an example of a thread-block DAG with the assumption that each kernel executes in a coalesced manner.

As we shall see in Sections 4.3 through 4.5, Corollary 4.2.2 allows us to estimate the running time of an MCM program as a function of the number ℓ of threads per thread-block, the single machine parameter U and the thread-block DAG of \mathcal{P} . However, the dependence on the machine parameter Z (the size of a private memory) is only through inequalities specifying upper bounds for ℓ . In addition, in each of the case studies, there is no need to make any assumptions (like inequality constraints) on the machine parameter U .

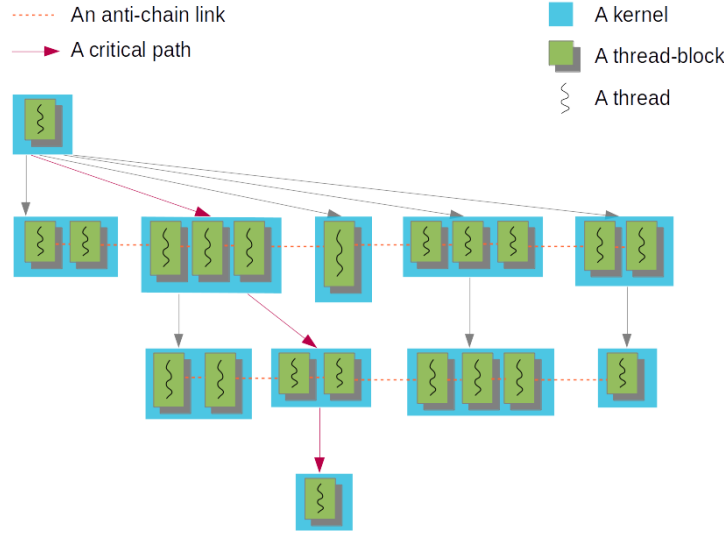


Figure 4.3: An example of a thread-block DAG

4.3 The Euclidean algorithm

Our first application of the MCM model deals with a multithreaded algorithm for computing the greatest common divisor (GCD) of two univariate polynomials. To specify notations, let \mathbb{K} be a field of coefficients (like the finite field $\mathbb{Z}/p\mathbb{Z}$ of prime characteristic p) and $\mathbb{K}[X]$ be the set of all univariate polynomials with coefficients in \mathbb{K} .

Our approach is based on the Euclidean algorithm, which can be reviewed in Chapter 4 of [74]. Given a positive integer s , we proceed by repeatedly calling a subroutine, see Algorithm 1. This subroutine takes as input a pair (a, b) of polynomials in $\mathbb{K}[X]$ with $\deg(a) \geq \deg(b) > 0$ and returns another pair (a', b') of polynomials in $\mathbb{K}[X]$, such that $\gcd(a, b) = \gcd(a', b')$ and, either $b' = 0$ (in which case we have $\gcd(a, b) = a'$), or we have $\deg(a') + \deg(b') \leq \deg(a) + \deg(b) - s$. We will take advantage of our MCM model to tune the program parameter s in order to obtain an optimized multithreaded version of the Euclidean algorithm.

Let n and m be positive integers such that $\deg(a) = n - 1$ and $\deg(b) = m - 1$, assuming $n \geq m$. We use a dense representation for encoding the polynomials a and b . Thus, n and m are the number of coefficients (zero or not) of a and b , respectively. Algorithm 1 uses two one-dimensional arrays of size n and m , respectively, to represent a and b , where the coefficient of the term of a (resp. b) in degree i is stored in $a[i]$ (resp. $b[i]$). Observe that Algorithm 1 updates a and b . We denote by d_a and d_b the degrees of these updated polynomials, while we reserve n and m for the initial values of the sizes of a and b .

Algorithm 1 is implemented as a kernel, which proceeds as follows. While $d_a \geq d_b$ holds, the polynomial a is replaced by $a - cX^d b$, where c is the leading coefficient of a divided by that of b and $d = d_a - d_b$. If this process reduces a to zero, or decreases d_a by s , in case $d_a \geq d_b$ still holds, then the updated pair (a, b) is returned. If the condition $d_a \geq d_b$ becomes false before d_a can decrease by s , then the roles of a and b are exchanged; thus, a becomes the divisor.

Hence, each call to this kernel either computes the GCD of its input polynomials, or makes the sum of their degrees decreased at least by s . Since this may require performing s division

Algorithm 1: OptGcdKer(a, b, s, da, db)**Input:** $a, b \in \mathbb{K}[X]$, an integer $s \geq 1$ and da, db store the current degrees of a and b respectively.**Output:** Either one of a, b was set to $\gcd(a, b)$ (and the other to 0), or $da + db$ reduced at least by s .

```

1  Let Alc, Blc, A, B be arrays of size  $s, s, \ell + s, \ell + s$  respectively with coefficients in  $\mathbb{K}$ , allocated
   on the private memory of the device; local integers  $u = v = w = e = 0$ ;
2   $j = \text{blockID} \cdot \text{blockDim} + \text{threadID}$ ;  $t = \text{threadID}$ ;
   /* copying from global memory */
3  if  $t < s$  then
4    Alc[ $t$ ] =  $a[da - t]$ ; Blc[ $t$ ] =  $b[db - t]$ ;
5  if  $t \geq s$  then
6    A[ $t - s$ ] =  $a[da - s \cdot \text{blockID} - t]$ ; B[ $t - s$ ] =  $b[db - s \cdot \text{blockID} - t]$ ;
   /* computing next remainders */
7  for ( $k = 0$ ;  $k < s$ ;  $k = k + 1$ ) do
8    if ( $da \geq db$  and  $db \geq 0$ ) then
9      if ( $u + t < s$ ) and ( $v + t < s$ ) then
10       Alc[ $u + t$ ] -= Blc[ $v + t$ ] · Alc[ $u$ ] · Blc[ $v$ ]-1;
11      if ( $u + t \geq s$ ) and ( $v + t \geq s$ ) then
12       A[ $w + t - s$ ] -= B[ $e + t - s$ ] · Alc[ $u$ ] · Blc[ $v$ ]-1;
13      if  $t == 0$  then
14       while Alc[ $u$ ] = 0 do
15         u =  $u + 1$ ; w =  $w + 1$ ; da =  $da - 1$ ;
16      if ( $db \geq da$ ) and ( $da \geq 0$ ) then
17       if ( $u + t < s$ ) and ( $v + t < s$ ) then
18        Blc[ $v + t$ ] -= Alc[ $u + t$ ] · Blc[ $v$ ] · Alc[ $u$ ]-1;
19       if ( $u + t \geq s$ ) and ( $v + t \geq s$ ) then
20        B[ $e + t - s$ ] -= A[ $w + t - s$ ] · Blc[ $v$ ] · Alc[ $u$ ]-1;
21       if  $t == 0$  then
22        while Blc[ $v$ ] = 0 do
23          v =  $v + 1$ ; e =  $e + 1$ ; db =  $db - 1$ ;
24  if  $t \geq s$  then
   /* writing to global memory */
25    a[ $da - s \cdot \text{blockID} - t$ ] = A[ $t - s$ ];
26    b[ $db - s \cdot \text{blockID} - t$ ] = B[ $t - s$ ];
27  if  $j == \min(da, db)$  then
28    Update da, db with the new degrees of  $a$  and  $b$ ;

```

steps (using either a or b as divisor), each thread-block must compute the leading coefficients of a and b at each division step.

From the specifications of Algorithm 1, it follows that computing $\gcd(a, b)$ requires at most $\lceil \frac{n+m}{s} \rceil$ kernel calls. With the goal of optimizing the use of computing resources and sharpening the analysis of our multithreaded GCD computation, we observe that these kernel calls can be

separated into two computational phases, which we call *ping-ping* and *ping-pong*. During the *ping-ping* phase, the inequality $d_a \geq d_b$ remains true, and thus this phase amounts to at most $\lceil \frac{n-m}{s} \rceil$ kernel calls. The subsequent kernel calls form the *ping-pong* phase, during which the role of the divisor alternates between a and b ; thus, there are at most $\lceil \frac{2m}{s} \rceil$ kernel calls in this second phase.

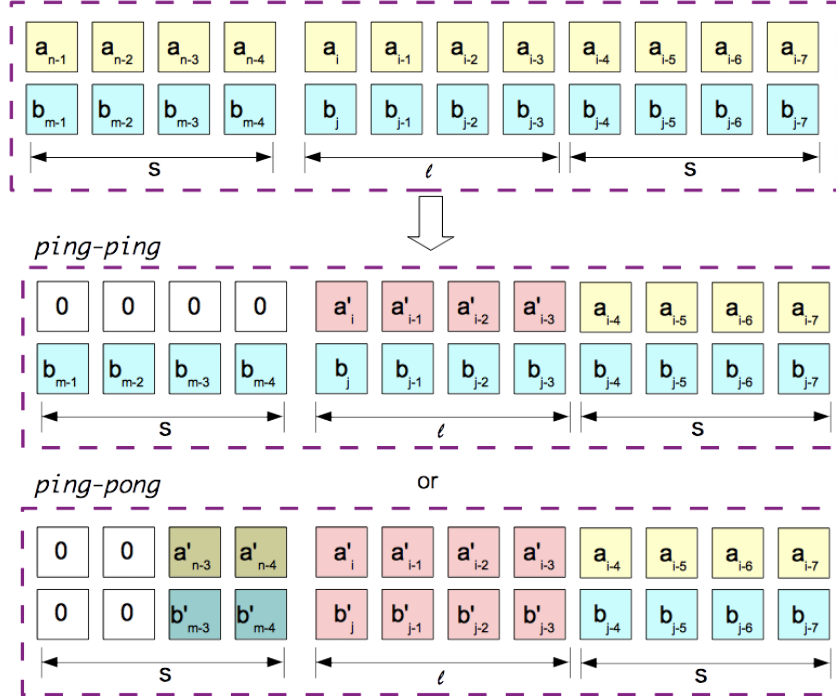


Figure 4.4: Illustration of reads and writes by a thread-block in either ping-ping or ping-pong phase of the Euclidean algorithm

Denoting by ℓ the number of threads per thread-block, we observe that each kernel call requires $\lceil \frac{m}{\ell} \rceil$ thread-blocks in the *ping-ping* phase. During the *ping-pong* phase, for a kernel called on polynomials with current degrees d_a and d_b , the number of required thread-blocks becomes $\min(\lceil \frac{d_a+1}{\ell} \rceil, \lceil \frac{d_b+1}{\ell} \rceil)$.

After executing a kernel call in the ping-ping phase, the s largest-degree coefficients of a have been set to zero; meanwhile, b is left unchanged. Each thread-block updates ℓ other coefficients of a . After executing a kernel call in the ping-pong phase, s largest-degree coefficients among a and b have been set to zero; meanwhile, each thread-block has updated 2ℓ other coefficients of a and b . See Figure 4.4 for an illustration of both scenarios.

To ensure that every kernel call (in either ping-ping or ping-pong phase) can perform (at most) s division steps correctly, each thread-block reads the s largest-degree coefficients from both a and b , as well as $\ell + s$ other consecutive coefficients from both a and b . Thereby, $4s + 2\ell$ coefficients must fit into the private memory of each streaming multiprocessor; hence, we have $4s + 2\ell \leq Z$.

We note that two consecutive thread-blocks have s common coefficients of both a and b . Moreover, all thread-blocks of a given kernel call read the s largest-degree coefficient of both a and b . Despite this duplication of data access and the corresponding duplication of work, this

multithreaded GCD algorithm is practically effective and is optimized for a relatively large s , as we shall see.

The work, span and parallelism overhead are given¹ respectively, by

$$\begin{aligned} W_s &= 3m^2 + 6nm + 3s + \frac{3(5ms + 4ns + 14m + 4n + 3s^2 + 6s)}{8\ell}, \\ S_s &= 3n + 3m \text{ and} \\ O_s &= \frac{4mU(2n+m+s)}{s\ell}. \end{aligned}$$

To determine a value range for s that minimizes the parallelism overhead of our multi-threaded algorithm, we choose $s = 1$ as a starting point. Let W_1 , S_1 and O_1 be the work, span and parallelism overhead at $s = 1$. The work ratio W_1/W_s is asymptotically equivalent to

$$\frac{(16\ell + 8)n + (8\ell + 19)m}{(16\ell + 4s + 4)n + (8\ell + 5s + 14)m},$$

when n (and thus m) escapes to infinity. The span ratio S_1/S_s is 1, and the parallelism overhead ratio O_1/O_s is

$$\frac{(2n + m + 1)s}{2n + m + s}.$$

We observe that when $s \in \Theta(\ell)$, the work is increased only by a constant factor, while the parallelism overhead is reduced by a factor in $\Theta(s)$.

Hence, choosing $s \in \Theta(\ell)$ seems a good choice. To verify this, we apply Corollary 4.2.2. One can easily check that the quantities characterizing the thread-block DAG of the computation are

$$N_s = \frac{2nm + m^2 + ms}{2s\ell}, \quad L_s = \frac{n + m}{s} \text{ and } C_s = 3s + 8U.$$

Then, applying Corollary 4.2.2, we estimate the running time on $\Theta(\frac{m}{\ell})$ SMs as

$$T_s = \frac{4n + 3m + s}{2s} (3s + 8U).$$

Denoting by T_1 the estimated running time when $s = 1$, the running time ratio $R = T_1/T_s$ on $\Theta(\frac{m}{\ell})$ SMs is given by

$$\frac{(4n + 3m + 1)(3 + 8U)s}{(4n + 3m + s)(3s + 8U)}.$$

When n and m escape to infinity, the latter ratio asymptotically becomes

$$\frac{(3 + 8U)s}{3s + 8U},$$

which is greater than 1 if and only if $s > 1$. Thus, the algorithm with $s = \Theta(\ell)$ performs better than that with $s = 1$. Figure 4.5 shows the experimental results with $s = \ell = 256$ and $s = 1$ on an NVIDIA Kepler architecture, which confirms our theoretical analysis.

¹ See the detailed analysis in the form of executable MAPLE worksheets of all applications: <http://www.csd.uwo.ca/~nxie6/projects/mcm/> or the PDF version of these worksheets in Appendix B.

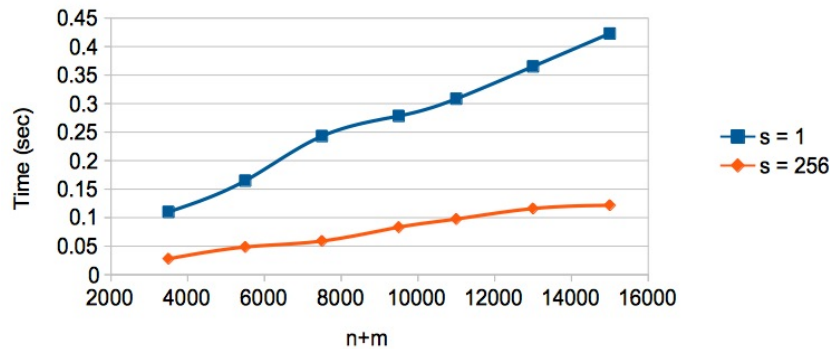


Figure 4.5: Running time on GeForce GTX 670 of our multithreaded Euclidean algorithm for univariate polynomials of sizes n and m over $\mathbb{Z}/p\mathbb{Z}$, where p is a 30-bit prime, whereas the program parameter takes values $s = 1$ and $s = 256$

4.4 Fast Fourier Transform

Let f be a univariate polynomial over the prime field of characteristic p , namely $\mathbb{F}_p := \mathbb{Z}/p\mathbb{Z}$, where p is a prime number greater than 2. Let n be the smallest power of 2, such that the degree of f is less than n , that is, $n = \min\{2^e \mid \deg(f) < 2^e \text{ and } e \in \mathbb{N}\}$. We assume that n divides $p - 1$, which guarantees that the field \mathbb{F}_p admits an n -th primitive root of unity. Hence, let $\omega \in \mathbb{F}_p$ such that $\omega^n = 1$ holds, while for all $0 \leq i < n$, we have $\omega^i \neq 1$. The n -point *Discrete Fourier Transform* (DFT) at ω is the linear map from the \mathbb{F}_p -vector space \mathbb{F}_p^n to itself, defined by $x \mapsto \text{DFT}_n x$ with the n -th DFT matrix given by

$$\text{DFT}_n = [\omega^{ij}]_{0 \leq i, j < n}. \quad (4.5)$$

Since ω is an n -th primitive root of unity, this map is invertible and its inverse is $1/n$ times the DFT at ω' , where ω' is the inverse of ω in \mathbb{F}_p .

A *fast Fourier transform* (FFT) is an algorithm to compute the DFT and its inverse. Two algorithms of the most commonly used FFTs' are that of Cooley & Tukey [33] and that of Stockham [109]. Before reviewing and analyzing those algorithms on an MCM machine, we introduce a few notations. Given an input vector v of length nm , we define the *stride permutation* L_m^{nm} as the permutation that maps the entry of v at position $in + j$ to position $jm + i$ for $0 \leq j \leq (m-1)$ and $0 \leq i \leq (n-1)$. We denote by I_x the identity matrix of order x . The symbols \oplus and \otimes are used for the *direct sum* and *Kronecker product*² [120] of matrices. We define the *twiddle matrix* $D_{x,y}$ as the diagonal matrix of order xy given by $\bigoplus_{j=0}^{x-1} \text{diag}(1, w^j, \dots, w^{j(y-1)})$.

4.4.1 Cooley & Tukey algorithm

Let $k = \log_2(n)$ and fix r for $0 < r < k$. Define $m = 2^r$ and thus m divides n . The algorithm of Cooley & Tukey is based on the following factorization³ of the matrix DFT_n

$$\text{DFT}_n = M_{k,r} M'_{k,r} M''_{k,r}, \quad (4.6)$$

²https://en.wikipedia.org/wiki/Kronecker_product

³Matrix multiplications are not commutative. Throughout, the product $\prod_{i=1}^s M_i$ stands for $M_1 M_2 \cdots M_s$, while $\prod_{i=s}^1 M_i$ means $M_s M_{s-1} \cdots M_1$.

where $M_{k,r} = \prod_{i=0}^{k-r-1} (I_{2^i} \otimes \text{DFT}_2 \otimes I_{2^{k-i-1}})(I_{2^i} \otimes D_{2,2^{k-i-1}})$, $M'_{k,r} = I_{2^{k-r}} \otimes \text{DFT}_m$ and $M''_{k,r} = \prod_{i=k-r-1}^0 I_{2^i} \otimes L_2^{2^{k-i}}$. Each of $M_{k,r}$, $M'_{k,r}$, $M''_{k,r}$ is a structured square matrix of order n representing a computational step of the algorithm.

As we shall see, the multiplications by the matrices $M''_{k,r}$ and $M_{k,r}$ are difficult to implement on a GPU-like architecture. Each factor of the matrix $M''_{k,r}$ is a Kronecker product of the form $I_{2^i} \otimes L_2^{2^{k-i}}$, that is, a diagonal matrix, which has order 2^i and is a permutation matrix. One may assume that, for a kernel implementing the multiplication by $I_{2^i} \otimes L_2^{2^{k-i}}$, all thread-blocks perform coalesced reads. But, for a small enough i , several non-coalesced memory accesses to the global memory may occur when writing back within one thread-block. Turning our attention now to the multiplication by $M_{k,r}$, we notice that, when the matrix $I_{2^{i-1}} \otimes D_{2,2^{k-i}}$ operates on a sub-vector of length 2^{k-i+1} , the powers $\{1, \omega^{2^i}, (\omega^{2^i})^2, \dots, (\omega^{2^i})^{2^{k-i}-1}\}$ need to be computed. A thread-block operating on a sub-vector of size x may need to compute x of those consecutive powers, which can be done in time $O(\log_2(x))$. Finally, multiplication by $M'_{k,r}$ causes no difficulty as long as m is large enough so as to avoid non-coalesced memory accesses. In our implementation, $m = 16$ is appropriate.

Let ℓ be the number of threads per thread-block, then $M_{k,r}$ and $M''_{k,r}$ are computed by $\log_2(n) - \log_2(m)$ kernel calls, respectively, requiring $\frac{n}{\ell}$ thread-blocks per kernel, and $M'_{k,r}$ is computed by one kernel with $\frac{n}{m\ell}$ thread-blocks.

We compute the work, span and parallelism overhead, respectively, as

$$\begin{aligned} W_{ct} &= n(34 \log_2(n) \log_2(\ell) + 47 \log_2(n) + 32m + 9 - 34 \log_2(m) \log_2(\ell) - 47 \log_2(m)), \\ S_{ct} &= 34 \log_2(n) \log_2(\ell) + 47 \log_2(n) + 32m \log_2(m) + 30m + 11 - 34 \log_2(\ell) \log_2(m) \\ &\quad - 79 \log_2(m) \text{ and} \\ O_{ct} &= \frac{2nU}{\ell} (4 \log_2(n) + \ell \log_2(\ell) + 1 - \log_2(\ell) - 4 \log_2(m)). \end{aligned}$$

To apply Corollary 4.2.2, one can easily check that those three quantities characterizing the thread-block DAG are

$$\begin{aligned} N_{ct} &= \frac{n}{m\ell} (2m \log_2(n) + 1 - 2m \log_2(m)), \\ L_{ct} &= 2 \log_2(n) - 2 \log_2(m) + 1 \text{ and} \\ C_{ct} &= 2U\ell + 34 \log_2(\ell) + 2U + 27. \end{aligned}$$

Thus, we estimate that the running time on $\Theta(\frac{n}{\ell})$ streaming multiprocessors is

$$T_{ct} = (4 \log_2(n) + 1 + \frac{1}{m} - 4 \log_2(m)) (2U\ell + 34 \log_2(\ell) + 2U + 27).$$

4.4.2 Stockham algorithm

The Stockham algorithm is based on the following factorization of the matrix DFT_n

$$\text{DFT}_n = \prod_{i=0}^{k-1} (\text{DFT}_2 \otimes I_{2^{k-1}})(D_{2,2^{k-i-1}} \otimes I_{2^i})(L_2^{2^{k-i}} \otimes I_{2^i}),$$

where $k = \log_2(n)$ as before. For each $0 \leq i < k$, one performs three matrix-vector multiplications:

$$A_{k,i} : x \mapsto (L_2^{2^{k-i}} \otimes I_{2^i})x, \quad A'_{k,i} : x \mapsto (D_{2,2^{k-i-1}} \otimes I_{2^i})x, \quad \text{and} \quad A''_{k,i} : x \mapsto (\text{DFT}_2 \otimes I_{2^{k-1}})x.$$

Thus, the Stockham algorithm can be implemented as $\log_2(n)$ calls to a kernel performing successively the matrix-vector multiplications defined by $A_{k,i}$, $A'_{k,i}$ and $A''_{k,i}$. Each of the corresponding matrices has a structure permitting coalesced read/write memory accesses. Indeed, for each of these matrices, all coefficients are null except along a few segments parallel to the diagonal. See [92] for details.

Let ℓ be the number of threads per thread-block; thus, each kernel requires $\frac{n}{\ell}$ thread-blocks. We compute the work, span and parallelism overhead, respectively, as

$$\begin{aligned} W_{sh} &= n 43 \log_2(n) + \frac{n}{4\ell} + 12\ell + 1 - 30n, \\ S_{sh} &= 43 \log_2(n) + 16 \log_2(\ell) + 3 \text{ and} \\ O_{sh} &= \frac{5nU \log_2(n)}{\ell} + \frac{5nU}{4\ell}. \end{aligned}$$

Applying Corollary 4.2.2, the quantities characterizing the thread-block DAG are

$$N_{sh} = \frac{n(8 \log_2(n) - 5)}{4\ell}, \quad L_{sh} = 3 \log_2(n) + 1 \text{ and } C_{sh} = 8 \log_2(\ell) + 4U + 17.$$

Hence, the running time estimate on $\Theta(\frac{n}{\ell})$ SMs is

$$T_{sh} = \log_2(n) (40 \log_2(\ell) + 20U + 85) - 2 \log_2(\ell) - U - \frac{17}{4}.$$

4.4.3 Comparison of running time estimates

The work ratio W_{ct}/W_{sh} is asymptotically equivalent to

$$\frac{4n(47 \log_2(n)\ell + 34 \log_2(n)\ell \log_2(\ell))}{172n \log_2(n)\ell + n + 48\ell^2},$$

when n escapes to infinity. Since $\ell \in O(Z)$, the quantity ℓ is bounded over on a given machine. Thus, the work ratio is asymptotically equivalent to $\log_2(\ell)$ when n escapes to infinity, while the span ratio S_{ct}/S_{sh} is asymptotically equivalent to

$$\frac{34 \log_2(n) \log_2(\ell) + 47 \log_2(n)}{43 \log_2(n) + 16 \log_2(\ell)},$$

which is also in $\Theta(\log_2(\ell))$. In other words, both the work and span of the algorithm of Cooley & Tukey are increased by $\Theta(\log_2(\ell))$ factor w.r.t their counterparts in the Stockham algorithm. Next, we compute the parallelism overhead ratio O_{ct}/O_{sh} as

$$\frac{8n(4 \log_2(n) + \ell \log_2(\ell) - \log_2(\ell) - 15)}{20n \log_2(n) + 5n - 4\ell}.$$

Applying Corollary 4.2.2, we obtain the running time ratio $R = T_{ct}/T_{sh}$ on $\Theta(\frac{n}{\ell})$ SMs as

$$R \sim \frac{\log_2(n)(2U\ell + 34 \log_2(\ell) + 2U)}{5 \log_2(n)(U + 2 \log_2(\ell))},$$

when n escapes to infinity. This latter ratio is greater than 1 if and only if $\ell > 1$.

Hence, the Stockham algorithm outperforms the Cooley & Tukey algorithm on an MCM machine. Table 4.1 shows the experimental results comparing the Cooley & Tukey and Stockham algorithms with $\ell = 128$ on an NVIDIA Kepler architecture. The experimentation confirms our theoretical analysis.

Table 4.1: Running time (in secs) of the Cooley & Tukey and Stockham FFT algorithms with the input size n on GeForce GTX 670

n	Cooley & Tukey	Stockham
2^{14}	0.583296	0.666496
2^{15}	0.826784	0.7624
2^{16}	1.19542	0.929632
2^{17}	2.07514	1.24928
2^{18}	4.66762	1.86458
2^{19}	9.11498	3.04365
2^{20}	16.8699	5.38781

4.5 Polynomial multiplication

Multithreaded algorithms for polynomial multiplication are our third application of the MCM model. As in Section 4.3, we denote by a and b two univariate polynomials with coefficients in the prime field \mathbb{F}_p , and we write their degrees $\deg(a) = n-1$ and $\deg(b) = m-1$, for two positive integers $n \geq m$. We compute the product $f = a \times b$ in two ways: plain multiplication and FFT-based multiplication. We describe the plain multiplication approach in Section 4.5.1; moreover, we analyze it so as to tune a program parameter s and thus obtain an optimized algorithm. In Section 4.5.2, we analyze an FFT-based multiplication algorithm that uses the Stockham FFT algorithm. Finally, we compare the plain and FFT-based multiplication algorithms in Section 4.5.3 via the MCM model and experimentally.

4.5.1 Plain multiplication

Our first multithreaded polynomial multiplication algorithm is based on the well-known *long multiplication*⁴ [120] and consists of two phases. During the *multiplication phase*, every coefficient of a is multiplied with every coefficient of b , and the resulting coefficient products are accumulated in an auxiliary array M . Then, during the *addition phase*, these accumulated products are added together to form the polynomial f . The top level algorithm shown in Algorithm 2 performs the multiplication phase once via Algorithm 3, and the addition phase by repeated calls to Algorithm 4. We consider as a program parameter the number $s > 0$ of coefficients that each thread writes back to the global memory at the end of each (multiplication or addition) phase.

We denote by ℓ the number of threads per thread-block. In the multiplication phase, each thread-block reads $s\ell + s - 1$ coefficients of a and s coefficients of b , and then computes ℓs^2 products followed by $\ell s(s-1)$ additions. Thus, each thread-block contributes $s\ell$ *partial sums* to a two-dimensional array M , whose format is $x \cdot y$ with $x = \frac{m}{s}$ and $y = n + s - 1$. This multiplication phase, illustrated by Figure 4.6, loads $s\ell + s - 1$ coefficients of a to guarantee the correctness of the results in M . Thereby, $2s\ell + 2s - 1$ coefficients must fit into the private memory, that is, $2s\ell + 2s - 1 \leq Z$, and the kernel requires $\frac{xy}{\ell s} = \frac{(n+s-1)m}{\ell s^2}$ thread-blocks.

In the addition phase, the x rows of the auxiliary array M are added pairwise in $\log_2(x)$ parallel steps. After each step, the number of rows in M is reduced by half, until we obtain

⁴http://en.wikipedia.org/wiki/Multiplication_algorithm

Algorithm 2: PlainMultiplicationGPU(a, b, s)**Input:** $a, b \in \mathbb{F}_p[X]$ with $n := \deg(a) + 1$ and $m := \deg(b) + 1$ and an integer $s \geq 1$.**Output:** $f \in \mathbb{F}_p[X]$ and $f = a \times b$.

```

1  $y = n + s - 1$ ;  $x = m/s$ ;
2 Let  $M$  be an array of size  $x \cdot y$ ;
3  $\ell$  is the number of threads per block;
4 MulKer $\lll x \cdot y/(s \cdot \ell), \ell \ggg (a, b, M, n, m, s)$ ;
5 for ( $i = 0$ ;  $i < \log_2 x$ ;  $i = i + 1$ ) do
6    $\lll$  AddKer $\lll x \cdot y/(2^{i+1} s \cdot \ell), \ell \ggg (M, f, y, s, x, i)$ ;
7 return  $f$ ;
```

Algorithm 3: MulKer(a, b, M, n, m, s)**Input:** $a, b, M \in \mathbb{F}_p[X]$ and an integer $s \geq 1$.

```

1  $\ell = \text{blockDim}$ ;  $t = \text{threadID}$ ;  $j = \text{blockID} \cdot \ell + t$ ;
2 Let  $B$  and  $A$  be two arrays of size  $s$  and  $\ell \cdot s + s - 1$  respectively, allocated on the private memory
  of the device;
3  $i = s \cdot \lfloor s \cdot j/(n + s - 1) \rfloor + t$ ;
4 if  $i < m$  and  $t < s$  then
5    $B[t] = b[i]$ ;
6  $i = s \cdot (j \bmod \frac{n+s-1}{s})$ ;
7 for ( $k = 0$ ;  $k < s$ ;  $k = k + 1$ ) do
8   if  $i + k \cdot \ell + t < n$  then
9      $A[k \cdot \ell + t] = a[i + k \cdot \ell + t]$ ;
10 if  $i - s + t > 0$  and  $t < s - 1$  then
11    $A[\ell \cdot s + t] = a[i - s + t]$ ;
12 else if  $t < s - 1$  then
13    $A[\ell \cdot s + t] = 0$ ;
14 for ( $e = 0$ ;  $e < s$ ;  $e = e + 1$ ) do
15    $h = 0$ ;
16   for ( $k = 0$ ;  $k < s$ ;  $k = k + 1$ ) do
17      $h += A[e \cdot \ell + k] \cdot B[k]$ ;
18    $M[s \cdot j + e] = h$ ;
```

only one row, that is, $f = a \times b$. To be more specific, at a parallel step k ($0 \leq k < \log_2(x)$), adding rows i and j (for $i < j$) as shown in Figure 4.7, the kernel requires $\frac{xy}{2^{k+1} \ell s} = \frac{(n+s-1)m}{2^{k+1} \ell s^2}$ thread-blocks, while each thread-block loads $s \ell$ elements of $M[i]$ and $M[j]$, respectively, and then adds $M[j]$ to $M[i]$.

We compute the work, span and parallelism overhead of the plain multiplication with an

Algorithm 4: AddKer(M, f, y, s, x, i)**Input:** $M, f, \in \mathbb{F}_p[X]$ and y, s, x, i are positive integers.

```

1  $j = \text{blockID} \cdot \text{blockDim} + \text{threadID};$ 
2  $h = s \cdot j \bmod y;$ 
3  $k = 2^i - 1 + 2^{i+1} \lfloor s \cdot j / y \rfloor;$ 
4 if  $h < 2^i s$  then
5   for ( $e = 0; e < s; e = e + 1$ ) do
6      $f[k \cdot s + h + e] += M[k \cdot x + h + e];$ 
7 else
8   for ( $e = 0; e < s; e = e + 1$ ) do
9      $M[(k + 2^i) \cdot x + h - 2^i s + e] += M[k \cdot x + h + e];$ 

```

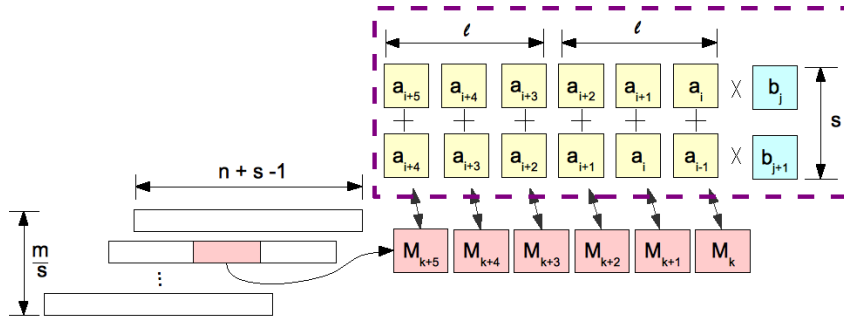


Figure 4.6: Multiplication phase: illustration of a thread-block reading coefficients from a, b and writing to the auxiliary array M

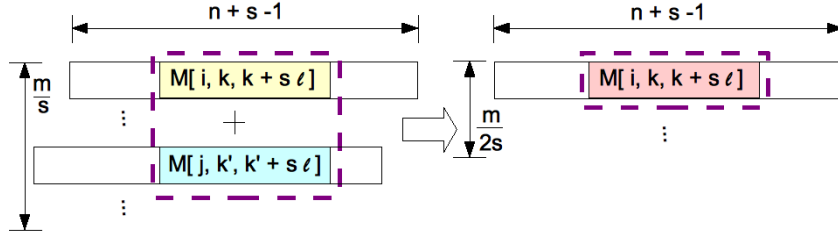


Figure 4.7: Addition phase: illustration of a thread-block reading and writing to the auxiliary array M

arbitrary s , respectively, as

$$W_s = (2m - \frac{1}{2})(n+s-1), \quad S_s = 2s^2 + s \log_2 \frac{m}{s} - s \quad \text{and} \quad O_s = \frac{(n+s-1)(5ms + 2m - 3s^2)U}{s^2 \ell}.$$

We also obtain the quantities characterizing the thread-block DAG that are required in order to apply Corollary 4.2.2:

$$N_s = \frac{(n+s-1)(2m-s)}{s^2 \ell}, \quad L_s = \log_2\left(\frac{m}{s}\right) + 1 \quad \text{and} \quad C_s = s(2s-1) + 2U(s+1).$$

We set $s = 1$ in the above two phases, and denote its work, span and parallelism overhead as W_1 , S_1 and O_1 , respectively. The work ratio

$$\frac{W_1}{W_s} = \frac{n}{n + s - 1}$$

is asymptotically constant as n escapes to infinity. The span ratio

$$\frac{S_1}{S_s} = \frac{\log_2(m) + 1}{s(\log_2(m/s) + 2s - 1)}$$

shows that S_s grows asymptotically with s . The parallelism overhead ratio is

$$\frac{O_1}{O_s} = \frac{n s^2 (7m - 3)}{(n + s - 1)(5ms + 2m - 3s^2)}.$$

We observe that, as n and m escape to infinity, this latter ratio is asymptotically equivalent to s . Applying Corollary 4.2.2, the estimated running time on $\Theta(\frac{(n+s-1)m}{\ell s^2})$ SMs is

$$T_s = \left(\frac{2m - s}{m} + \log_2\left(\frac{m}{s}\right) + 1 \right) (2Us + 2s^2 + 2U - s).$$

Let R be the ratio of the running time estimate between the algorithm with $s = 1$ and that for an arbitrary s , we obtain

$$R = \frac{(m \log_2(m) + 3m - 1)(1 + 4U)}{(m \log_2(\frac{m}{s}) + 3m - s)(2Us + 2U + 2s^2 - s)},$$

which is asymptotically equivalent to

$$\frac{2U \log_2(m)}{s(s + U) \log_2(m/s)}.$$

This latter ratio is smaller than 1 for $s > 1$. In other words, increasing s worsens the algorithm performance. In practice, as shown in Figure 4.8, setting $s = 4$ (where $\ell = 256$) performs best, while with larger s , the running time becomes slower. This practical observation is coherent with our theoretical analysis.

4.5.2 FFT-based multiplication

Let ω be an n -th root primitive of unity w , as defined in Section 4.4. We assume that the successive powers $\{1, w, w^2, \dots, w^{n-1}\}$ are pre-computed via a parallel prefix sum. The product $f = a \times b$ is computed as follows:

- (i) FFT computations: $a' = \text{DFT}_w(a)$ and $b' = \text{DFT}_w(b)$;
- (ii) Point-wise multiplication: $f' = a' \times b'$;
- (iii) FFT computation: $f' = \text{DFT}_{w^{-1}}(f')$;
- (iv) Scale the vector: $f = \frac{1}{n} f'$ and **return** f ;

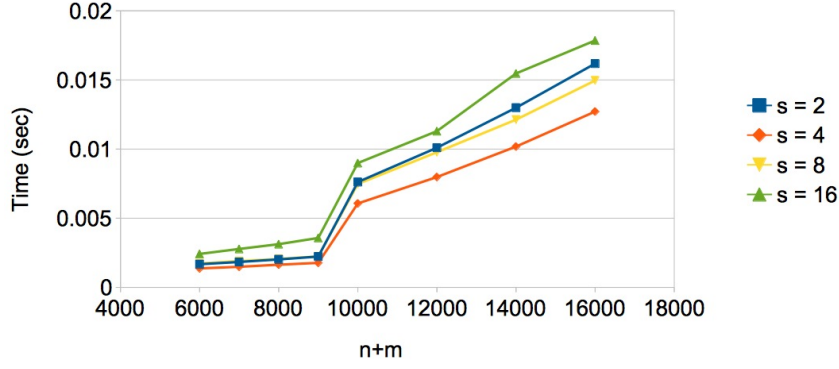


Figure 4.8: Running time of the plain polynomial multiplication algorithm with polynomials a ($\deg(a) = n - 1$) and b ($\deg(b) = m - 1$) and the parameter s on GeForce GTX 670

A kernel for computing the Stockham FFT algorithm has been described in Section 4.4. Two other kernels perform, respectively, the point-wise multiplication and vector scaling.

Let ℓ be the number of threads per thread-block. The analysis of the Stockham FFT algorithm is described in Section 4.4.3. To compute the point-wise multiplication, the kernel requires $\frac{n}{\ell}$ thread-blocks with each thread reading two data items and writing one data item back to the global memory. The final kernel, for vector scaling, also requires $\frac{n}{\ell}$ thread-blocks with each thread moving one data item between the global memory and the private memory. Hence, data movement in each kernel call is coalesced.

We compute the work, span and parallelism overhead of the overall FFT-based polynomial multiplication, respectively, as

$$W_{fft} = 129n \log_2(n) - 94n, \quad S_{fft} = 129 \log_2(n) - 94 \quad \text{and} \quad O_{fft} = \frac{nU(15 \log_2(n) - 4)}{\ell}.$$

Applying Corollary 4.2.2, we obtain the quantities characterizing the thread-block DAG:

$$N_{fft} = \frac{12n \log_2(n) - 5}{2\ell}, \quad L_{fft} = 9 \log_2(n) - 4 \quad \text{and} \quad C_{fft} = 4U + 25.$$

Thus, the running time estimate on $\Theta(\frac{n}{\ell})$ SMs is

$$T_{fft} = (15 \log_2(n) - \frac{13}{2})(4U + 25).$$

4.5.3 Comparison of running time estimates

Back to the plain multiplication, using $s = 4$ obtained from experimental results and setting $m = n$, we compute

$$W_{plain} = 2n^2 + \frac{11}{2}n - \frac{3}{2}, \quad S_{plain} = 4 \log_2(n) + 20 \quad \text{and} \quad O_{plain} = \frac{U(n+3)(11n-24)}{8\ell}.$$

We observe that W_{plain}/W_{fft} is in $O(\frac{n}{\log_2(n)})$, S_{plain}/S_{fft} is asymptotically constant, and O_{plain}/O_{fft} is in $O(\frac{n}{\log_2(n)})$. Next, the estimated running time of the plain multiplication on $\Theta(\frac{n(n+3)}{16\ell})$ SMs is

$$T_{plain} = \left(\frac{2(n-2)}{n} + \log_2(n) - 1 \right) (10U + 28).$$

The estimated running time ratio T_{plain}/T_{fft} is essentially constant when n escapes to infinity, while the plain multiplication performs more work and parallelism overhead.

However, the estimated running time of the plain multiplication on $\Theta(\frac{n}{\ell})$ SMs, in the case that we have limited resources, is

$$T'_{plain} = \left(\frac{(n+3)(n-2)}{8n} + \log_2(n) - 1 \right) (10U + 28),$$

whereas the estimated running time of the FFT-based multiplication is also based on $\Theta(\frac{n}{\ell})$ SMs. Thus, when n escapes to infinity, the estimated running time ratio T'_{plain}/T_{fft} on $\Theta(\frac{n}{\ell})$ SMs is asymptotically equivalent to

$$\frac{5U(n + 8\log_2(n))}{240U\log_2(n)},$$

and thus in $\Theta(n)$. Hence, the FFT-based multiplication outperforms the plain multiplication for a large enough n .

Figure 4.9 shows the experimental results comparing the plain and FFT-based multiplication algorithms with $\ell = 256$ on an NVIDIA Kepler architecture. We observe that for relatively small n , the plain multiplication performs better, but with n growing, the FFT-based multiplication becomes faster. Both phenomena were predicted by our theoretical analysis.

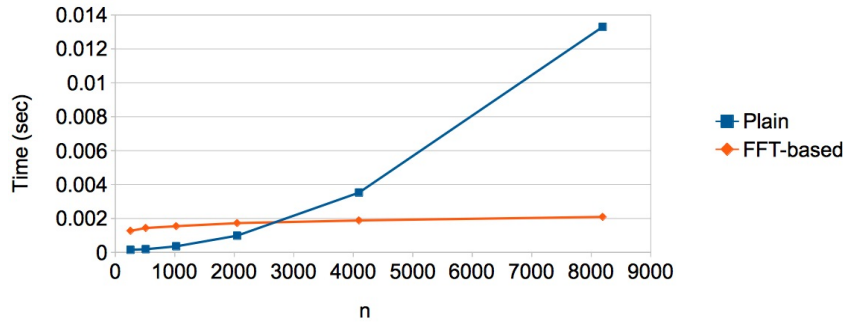


Figure 4.9: Running time of the plain and FFT-based multiplication algorithms with the input size n on GeForce GTX 670

4.6 Radix sort

In [103], Satish, Harris and Garland present a CUDA implementation of the radix sort algorithm. Assuming that all entries are non-negative integers of bit-size c , this CUDA implementation sorts n entries by performing $\frac{c}{s}$ passes, where s is a program parameter. In each pass, each thread-block first loads and sorts its tile using s iterations of 1-bit split, and writes back its 2^s -entry digit histogram and sorted data. Then, it performs a prefix sum over the histogram stored in a column-major order. Finally, each thread-block copies its elements to the correct output position.

Let ℓ be the number of threads per thread-block. Following [103], we assume that each thread deals with 4 elements. Then, for each thread-block, 4ℓ original elements, 4ℓ sorted elements and a 2^s -entry digit histogram must fit into the private memory; hence, $8\ell + 2^s \leq Z$.

Thus, the maximum parallelism overhead per thread-block is $9U$ (loading 4 elements, writing back 4 sorted elements and 1 value of the histogram).

We compute the work, span and parallelism overhead, respectively, as

$$\begin{aligned} W_s &= c \left(\frac{22s\ell + s + 12}{4s\ell} + \frac{2^s + 20 \cdot 2^s \ell}{16s\ell^2} + 1 \right) n + \frac{c(16 + 192\ell)}{16s}, \\ S_s &= c \left(8 \log_2(\ell) + \frac{16}{s} \log_2(\ell) + 41 + \frac{54}{s} \right) \text{ and} \\ O_s &= cU \left(\frac{9}{2s\ell} n + \frac{17 \cdot 2^s}{16s\ell^2} n - \frac{1}{s} \right). \end{aligned}$$

We view the case $s = 1$ as a naive radix sort algorithm, with the work, span and parallelism overhead given by W_1 , S_1 and O_1 , respectively. Letting n escaping to infinity, the work ratio W_1/W_s is asymptotically equivalent to:

$$\frac{W_1}{W_s} \sim \frac{104s\ell^2 + 92s\ell + 2s}{88s\ell^2 + 16\ell^2 + 20 \cdot 2^s \ell + 4s\ell + 48\ell + 2^s}.$$

Similarly, the span ratio

$$\frac{S_1}{S_s} = \frac{s(24 \log_2(\ell) + 95)}{(8s + 16) \log_2(\ell) + 41s + 54}$$

is asymptotically constant; meanwhile, the parallelism overhead ratio is asymptotically equivalent to:

$$\frac{O_1}{O_s} \sim \frac{s(72\ell + 34)}{72\ell + 17 \cdot 2^s}.$$

We notice that if $2^s = \Theta(\ell)$ holds, we can reduce the parallelism overhead by a factor of s , while increasing the work only by a constant factor. However, with $2^s = \Theta(\ell^2)$, we increase both the work and the parallelism overhead by a non-constant factor. In this latter scenario, we could not optimize the naive algorithm in any sense.

To apply Corollary 4.2.2, we compute the three quantities characterizing the thread-block DAG:

$$N_s = \frac{c}{s} \left(\frac{1}{2\ell} + \frac{2^s}{8\ell^2} \right) n, \quad L_s = \frac{5c}{s} \quad \text{and} \quad C_s = s(41 + 8 \log_2(\ell)) + 12 + 9U.$$

We also denote these three quantities N_1 , L_1 and C_1 for the naive algorithm when $s = 1$. Then, the ratio R of the running time estimates on $\Theta(\frac{n}{4\ell})$ SMs between the naive algorithm with $s = 1$ and that for an arbitrary s is:

$$R = \frac{(14\ell + 2)(8 \log_2(\ell) + 9U + 53)s}{(14\ell + 2^s)(8s \log_2(\ell) + 41s + 9U + 12)}.$$

We then replace s by $\log_2(\ell)$, since, in this case, we would like to determine whether the estimated overall running time is better or worse. The quotient of the leading terms of $\ell \log_2(\ell)$ in R becomes

$$\frac{112 \log_2(\ell) + 126U + 742}{120 \log_2(\ell) + 615}.$$

This ratio is greater than 1 for $\ell < 2^{15.75U + 15.875}$, which is realistic since $U > 0$. Therefore, letting $2^s = \Theta(\ell)$, the data transfer overhead can be reduced by a factor of s and lead to an optimized algorithm. This is consistent with the empirical results of [103].

4.7 Conclusion

We have presented a model of multithreaded computation combining the fork-join and SIMD parallelisms, with an emphasis on estimating parallelism overheads of GPU programs, so as to reduce communication and synchronization costs in GPU programs. In practice, our model determines a trade-off among *work*, *span* and *parallelism overhead* by checking the estimated overall running time so as to either (1) tune a program parameter or, (2) compare different algorithms independently of the hardware details.

Several applications illustrate the effectiveness of our model. With the Euclidean algorithm, plain multiplication and radix sort, we determine a range of values for a program parameter in order to optimize the corresponding algorithm in terms of parallelism overheads. Particularly for the Euclidean algorithm, our running time estimate matches that obtained with the systolic VLSI array model [17]. Moreover, our CUDA code in [60] implementing this optimized Euclidean algorithm runs in linear time w.r.t to the input polynomials up to degree 10,000. With FFT algorithms and polynomial multiplication algorithms, our model successfully compares, in each case, two different algorithms to determine the trend of their running time when the input data size grows. In all cases, experimentation validates the model analysis.

Chapter 5

MetaFork-to-CUDA: Generation of Parametric CUDA Kernels

In the past decade, the introduction of low-level heterogeneous programming models, in particular CUDA, has brought supercomputing to the level of the desktop computer. However, these models bring notable challenges, even to expert programmers. Indeed, fully exploiting the power of hardware accelerators with CUDA-like code often requires significant code optimization efforts. While this development can indeed yield high performance, it is desirable for some programmers to avoid the explicit management of device initialization and data transfer between memory levels. To this end, high-level models for accelerator programming have become an important research direction. With these models, programmers only need to annotate their C/C++ code to indicate which code portion is to be executed on the device and how data maps between host and device.

As of today, OpenMP [41, 13, 9] and OpenACC [113, 56] are among the most developed accelerator programming models. Both OpenMP and OpenACC are built on a host-centric execution model. The execution of the program starts on the host and may offload target regions to the device for execution. The device may have a separated memory space or may share memory with the host, so that memory coherence is not guaranteed and must be handled by the programmer. In OpenMP and OpenACC, the division of the work between thread-blocks within a grid and between threads within a thread-block can be expressed in a loose manner or even be ignored. This implies that code optimization techniques can be applied in order to derive efficient CUDA [95, 73]-like code.

In this chapter, we present the accelerator model of MetaFork (introduced in Section 2.4) together with the software framework that allows automatic generation of CUDA code from annotated C/C++ programs. One of the key properties of this CUDA code generator is that it supports the generation of CUDA kernel code where program parameters (like number of threads per thread-block) and machine parameters (like shared memory size) are allowed. These parameters need not to be known at code-generation-time: machine parameters and program parameters can be, respectively, determined and optimized when the generated code is installed on the targeted hardware.

The need for CUDA programs (more precisely, kernels) depending on program parameters and machine parameters is argued in Section 5.1. In Section 5.2, following the work reported in [55], we observe that generating *parametric* CUDA kernels requires the manipulation of sys-

tems of non-linear polynomial equations and the use of techniques like quantifier elimination (QE). To this end, we take advantage of the RegularChains library of MAPLE [27] and its QuantifierElimination command, which has been designed to efficiently support the non-linear polynomial systems coming from automatic parallelization.

Section 5.3 reports a preliminary implementation of parametric CUDA kernel code generation by the MetaFork compilation framework extending PPCG [115]. Finally, Section 5.4 gathers experimental data demonstrating the performance of our generated parametric CUDA code. These results show not only that the generation of parametric CUDA kernels helps optimize code independently of the values of the machine parameters of the targeted hardware, but also that automatic generation of parametric CUDA kernels can discover better values for the program parameters than those computed by a tool generating non-parametric CUDA kernels.

This chapter is related to the work reported in [19] and contains joint work with Changbo Chen, Xiaohui Chen and Marc Moreno Maza.

5.1 Optimizing CUDA kernels depending on program parameters

In Chapter 4, we propose a many-core machine (MCM) model for multithreaded computation combining the fork-join and SIMD parallelisms; meanwhile, a driving motivation in this work is to estimate the parallelism overheads (data communication and synchronization costs) of GPU programs. In practice, the MCM model determines a trade-off among *work*, *span* and *parallelism overhead* by checking the estimated overall running time so as to either (1) tune a program parameter or, (2) compare different algorithms independently of the hardware details.

The MCM model retains many of the characteristics of modern GPU architectures and programming models, like CUDA [95, 73] and OpenCL [111]. However, in order to support algorithm analysis with an emphasis on parallelism overheads, the MCM abstract machines admit a few simplifications and limitations with respect to actual many-core devices.

To further discuss CUDA kernel performance, let us consider now the programming model of CUDA itself and its differences w.r.t. the MCM model. In CUDA, instructions are issued per *warp*, and a warp consists of a fixed number S_{warp} of threads. Typically, S_{warp} is 32; thus, executing a thread-block on an SM means executing several warps in turn. If an operand of an executing instruction is not ready, then the corresponding warp stalls and context switching happens between warps running on the same SM.

Registers and shared memory are allocated for a thread-block as long as that thread-block is active. Once a thread-block is active, it stays active until all threads in that thread-block have completed. Context switching is very fast because registers and shared memory do not need to be saved and restored. The intention is to hide the latency (of data transfer between the global memory and the private memory of an SM) by having more memory transactions in fly. There is, of course, a hardware limitation to this, characterized by (at least) two numbers:

1. The maximum number of active warps per SM, denoted here by M_{warp} ; a typical value for M_{warp} is 48 on a Fermi NVIDIA GPU card, leading to a maximum number of $32 \times 48 = 1536$ active threads per SM.
2. The maximum number of active thread-blocks per SM, denoted here by M_{block} ; a typical

value for M_{block} is 8 on a Fermi NVIDIA GPU card.

One can now define a popular performance counter of CUDA kernels, the *occupancy* of an SM, given by $A_{\text{warp}}/M_{\text{warp}}$, where A_{warp} is the number of active warps on that SM. Since resources (registers, shared memory, thread slots) are allocated for an entire thread-block (as long as that thread-block is active), there are three potential limitations to occupancy: register usage, shared memory usage and thread-block size.

As in our discussion of the MCM model, we denote the thread-block size by ℓ . Regarding the possible values of ℓ , we observe that (1) the total number of active threads is bounded over by ℓM_{block} ; hence, a small value for ℓ may limit occupancy, and that (2) a larger value for ℓ reduces the number of registers and shared memory words available per thread; thus, this may limit data reuse within a thread-block and potentially increase the amount of data transfer between global memory and the private memory of an SM. Overall, this suggests again that generating the kernel code with ℓ and other program parameters considered as input arguments is a desirable goal. With such parametric code at hand, one can optimize, at run-time, the values of those program parameters (like ℓ), once the machine parameters (like S_{warp} , M_{warp} , M_{block} , Z (private memory size) and the size of the register file) are known.

5.2 Automatic parametric CUDA kernel generation

The general purpose of automatic parallelization is to convert sequential computer programs into multithreaded or vectorized code. Following the discussion of Section 5.1, we are interested here in the following more specific question.

Given a theoretically good parallel algorithm (e.g. divide-and-conquer matrix multiplication) and given a type of hardware that depends on various parameters (e.g. a GPGPU with Z words of private memory per SM and a maximum number M_{warp} of warps supported by an SM, etc.), we aim at automatically generating CUDA kernels that depend on the hardware parameters (Z , M_{warp} , etc.) as well as program parameters (e.g. the number ℓ of threads per thread-block). Thus, (1) those parameters need not to be known at compile-time, and (2) are encoded as symbols in the generated kernel code. For this reason, we call such CUDA kernels *parametric*.

In contrast, current technology requires that machine and program parameters are specialized to numerical values at the time of generating the GPGPU code, see [57, 6, 66, 115]. The *polyhedron model* [7] described in Section 2.5 is a powerful geometrical tool for analyzing the relation (w.r.t. data locality or parallelization) between the iterations of nested `for`-loops. Once the polyhedron representing the *iteration space* of a loop nest is calculated, techniques of linear algebra and linear programming can transform this polyhedron into another polyhedron encoding the loop steps into a coordinate system based on time and space (processors). From there, a parallel program can be generated.

For example, for the following code computing the product of two univariate polynomials a and b , both of degree n , and writing the result to c ,

```
for(int i = 0; i <= n; i++) { c[i] = 0; c[i+n] = 0; }
for(int i = 0; i <= n; i++) {
    for(int j = 0; j <= n; j++)
```

```

    c[i+j] += a[i] * b[j];
}

```

elementary dependence analysis [55] suggests to set $t(i, j) = n - j$ and $p(i, j) = i + j$, where t and p represent time and processor, respectively. Using Fourier-Motzkin elimination, projecting all constraints on the (t, p) -plane yields the following asynchronous schedule of the above code:

```

parallel_for (int p = 0; p <= 2*n; p++) {
    c[p]=0;
    for (int t = max(0, n-p); t <= min(n, 2*n-p); t++)
        c[p] += a[t+p-n] * b[n-t];
}

```

To be practically efficient, one should avoid a fine-grained parallelization, and this is achieved by grouping loop steps into so-called *tiles*, which are generally trapezoids [65]. It is also desirable for the generated code to depend on parameters such as tile, cache sizes and number of processors, etc. These extensions lead, however, to the manipulation of systems of non-linear polynomial equations and the use of techniques like quantifier elimination (QE). This was noticed by Gröbinger, Griebel and Lengauer in [55] who also observed that work remained to be done for adapting QE tools to the needs of automatic parallelization.

To illustrate these observations, we return to the above example and use a tiling approach: we consider a one-dimensional grid of thread-blocks where each thread-block is in charge of updating at most B coefficients of the polynomial c . Therefore, we introduce three variables B , b and u , where the latter two represent a thread-block index and a thread index (within a thread-block). This brings the following additional relations:

$$\begin{cases} 0 \leq b \\ 0 \leq u < B \\ p = bB + u, \end{cases} \quad (5.1)$$

to the previous system

$$\begin{cases} 0 < n \\ 0 \leq i \leq n \\ 0 \leq j \leq n \\ t = n - j \\ p = i + j. \end{cases} \quad (5.2)$$

To determine the target program, one needs to eliminate the variables i and j . In this case, Fourier-Motzkin elimination (FME) does not apply any more, due to the presence of non-linear constraints. If all the non-linear constraints appearing in a system of relations are polynomial constraints, the set of real solutions of such a system is a semi-algebraic set. The celebrated Tarski theorem [8] tells us that there always exists a quantifier elimination algorithm to project a semi-algebraic set of \mathbb{R}^n to a semi-algebraic set of \mathbb{R}^m , $m \leq n$. The most popular method for conducting quantifier elimination (QE) of a semi-algebraic set is through cylindrical algebraic decomposition (CAD) [31]. Implementation of QE and CAD can be found in software such as QEPCAD [18], REDUCE [64], MATHEMATICA [125] as well as the RegularChains library of

MAPLE [27]. Using the function `QuantifierElimination` (with options ‘precondition’=‘AP’, ‘output’=‘rootof’, ‘simplification’=‘L4’) in the `RegularChains` library, we obtain the following:

$$\left\{ \begin{array}{l} B > 0 \\ n > 0 \\ 0 \leq b \leq 2n/B \\ 0 \leq u < B \\ 0 \leq u \leq 2n - Bb \\ p = bB + u, \\ 0 \leq t \leq n, \\ n - p \leq t \leq 2n - p, \end{array} \right. \quad (5.3)$$

from where we derive the following program:

```
for (int p = 0; p <= 2*n; p++) { c[p]=0; }
parallel_for (int b = 0; b <= 2*n/B; b++) {
  for (int u = 0; u <= min(B-1, 2*n-B*b); u++) {
    int p = b * B + u;
    for (int t = max(0, n-p); t <= min(n, 2*n-p); t++)
      c[p] += a[t+p-n] * b[n-t];
  }
}
```

An equivalent CUDA kernel to the `parallel_for` part is as below:

```
int b = blockIdx.x;
int u = threadIdx.x;
if (u <= 2 * n - B * b) {
  int p = b * B + u;
  for (int t = max(0, n-p); t <= min(n, 2*n-p); t++)
    c[p] += a[t+p-n] * b[n-t];
}
```

We remark that the polynomial system defined by (5.1) and (5.2) has some special structure. In [54], Gröbinger, Griebel and Lengauer have exploited this structure to deduce a special algorithm to solve it and similar problems by implementing some parametric FME. Although the system (5.3) can be directly processed by `QuantifierElimination`, we figure out that it is much more efficient to use the following special QE procedure. We replace the product bB in system (5.1) by a new variable c , and thus obtain a system of linear constraints. We then apply FME to eliminate the variables i, j, t, p, u in a sequential order. Now we obtain a system of linear constraints in variables c, b, n, B . Next we replace c by bB and have again a system of non-linear constraints in variables b, n, B . We then call `QuantifierElimination` to eliminate the variables b, n, B . The correctness of the procedure is easy to verify.

5.3 The MetaFork-to-CUDA code generator

With the goal to generate efficient CUDA code from an input MetaFork program, the keyword `meta_schedule` is introduced to the MetaFork language such that one can annotate the Meta-

Fork code targeting many-cores in a precise manner. The semantic of the `meta_schedule` statement is an indication to the MetaFork-to-CUDA code generator that every `meta_for`-loop nest of the `meta_schedule` statement must translate to a CUDA kernel call. In Section 5.2, we illustrated the process of parametric CUDA kernel generation from a sequential C program using MetaFork as an intermediate language. In this section, we assume that, from a C program, one has generated a MetaFork program, which contains one or more `meta_schedule` statements.

A `meta_schedule` statement generating a one-dimensional grid with one-dimensional thread-blocks has the following structure

```
meta_schedule {
    // only for loops are supported here
    meta_for (int i = 0; i < gridDim.x; i++)
        // only for loops are supported here
        meta_for (int j = 0; j < blockDim.x; j++) {
            ... // nested for-loop body
        }
}
```

where the grid (resp. thread-block) dimension size is extracted from the outer (resp. inner) `meta_for` loop upper bound. Similarly, a `meta_schedule` statement generating a two-dimensional grid with two-dimensional thread-blocks has the following structure

```
meta_schedule {
    // only for loops are supported here
    meta_for (int u = 0; u < gridDim.y; u++)
        meta_for (int i = 0; i < gridDim.x; i++)
            // only for loops are supported here
            meta_for (int v = 0; v < blockDim.y; v++)
                meta_for (int j = 0; j < blockDim.x; j++) {
                    ... // nested for-loop body
                }
}
```

where the first two outer `meta_for` loops correspond to the grid and the inner `meta_for` loops to the thread-blocks.

To generate those parametric CUDA kernels, we rely on PPCG [115], a C-to-CUDA compilation framework, which we have modified, in order to generate compilable CUDA kernels. Figure 5.1 illustrates the software architecture of our C-to-CUDA code generator, based on PPCG. As shown in Figure 5.1, one can write a tiled MetaFork program directly or use QE to tile each loop nest in the MetaFork code. After that, one shall pass this tiled MetaFork code to our code generator, in particular, taking advantage of the sophisticated CUDA code generator in PPCG.

The original PPCG framework generates CUDA kernels with the number of threads per thread-block specified as a constant, explicitly given either by the user or by default using 32 for a one-dimensional kernel or 16×32 for a two-dimensional kernel. Our code generator focuses on generating parametric CUDA kernels and extends to support identifying `meta_for`

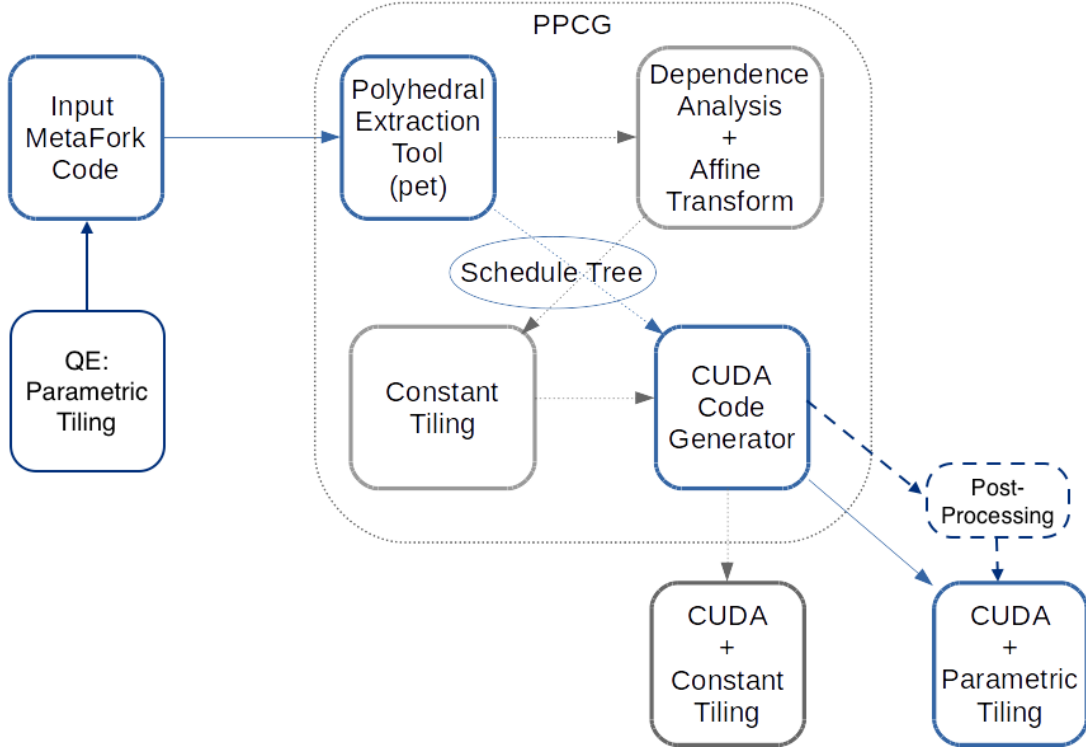


Figure 5.1: Overview of the implementation of the MetaFork-to-CUDA code generator

loops, such that CUDA code generation is fully automatic for using the global memory as well as, under some assumptions (demonstrated in Appendix C.1), for using the shared memory in the kernel code. Note that, for both data dependence and memory accessing pattern cases, the analysis involving non-linear expressions is not performed by PPCG or MetaFork. Consequently, either the programmer avoids such analysis by writing a proper MetaFork program, or a post-processing phase may be required.

Since the MetaFork code is obtained after computing affine transformation and tiling (via quantifier elimination), we have to bypass the process of computing affine transformation and tiling that PPCG performs. Thus, we modify the internal data structure, namely the schedule tree [117], which is used to represent the execution order of the input code in the polyhedral model and can be converted to the abstract syntax tree (AST). When we pass the MetaFork code to the software PET (*polyhedral extraction tool*) [116] for parsing, a schedule tree is initiated with the iteration space; however, we mark each `meta_for` as “permutable,” which indicates that later, it corresponds to a thread index or a thread-block index. More details on how we use the schedule tree to represent MetaFork and parametric CUDA programs are described in Appendix C.2. Furthermore, to generate the parametric variables, such as the grid and thread-block sizes, as symbols, we extend relevant data structures of PPCG to store those variable symbols so as to pass them from the schedule tree to the AST for CUDA code generation. Last but not least, we assign the thread-block indices to variables `b0`, `b1` and the thread indices to variables `t0`, `t1`.

Consider an example, a one-dimensional stencil computation, namely Jacobi. The original (and naive) C version is shown in Figure 5.2, where initialization statements have been

```

for (int t = 0; t < T; ++t) {
    for (int i = 1; i < N-1; ++i)
        b[i] = (a[i-1] + a[i] + a[i+1]) / 3;

    for (int i = 1; i < N-1; ++i)
        a[i] = b[i];
}

```

Figure 5.2: One-dimensional stencil computation, namely Jacobi, written in C

removed. From this C code fragment, we apply the tiling techniques mentioned in Section 5.2 and obtain the MetaFork code shown in Figure 5.3. Observe that the `meta_schedule` statement has two `meta_for` loop nests yielding two CUDA kernels.

```

int ub_v = (N - 2) / B;
meta_schedule {
    for (int t = 0; t < T; ++t) {
        meta_for (int v = 0; v < ub_v; v++) {
            meta_for (int u = 0; u < B; u++) {
                int p = v * B + u;
                b[p+1] = (a[p] + a[p+1] + a[p+2]) / 3;
            }
        }
        meta_for (int v = 0; v < ub_v; v++) {
            meta_for (int u = 0; u < B; u++) {
                int w = v * B + u;
                a[w+1] = b[w+1];
            }
        }
    }
}

```

Figure 5.3: 1D Jacobi written in MetaFork

Our MetaFork-to-CUDA code generator produces two kernel functions, a header file (for those two kernels) and a host code file where those kernels are called. Those two kernel functions are shown in Figure 5.4. In each kernel, we use the shared memory for those arrays read and use the global memory for those arrays written only once. Observe that `kernel0` and `kernel1` take a program parameter, the thread-block format `B`, as an argument, whereas non-parametric CUDA kernels usually take parameters $a, b, c, N, T, c0$ only. Correspondingly, the generated host code replacing `meta_schedule` and its body is shown in Figure 5.5. Data transfers between the CPU and GPU global memories are done before those two kernels launched and after those two kernels completed, respectively. In the case that the number of thread-blocks per grid, aka `ub_v` in the MetaFork code, exceeds the hardware limit, which is 32768 shown in the host code, each kernel uses 32768 as the grid dimension size, while inside the kernel code, the amount of work per thread-block is incremented via a serial loop.

```

__global__ void kernel0(int *a, int *b, int N, int T, int ub_v,
                        int B, int c0) {

    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    int private_p;
    __shared__ int shared_a[BLOCK_0+2]; // BLOCK_0 = B

    for (int c1 = b0; c1 < ub_v; c1 += 32768) {
        for (int c2 = t0; c2 <= min(B + 1, N - B * c1 - 1); c2 += B)
            shared_a[c2] = a[B * c1 + c2];
        __syncthreads();
        private_p = (((c1) * (B)) + (t0));
        b[private_p + 1] = (((shared_a[private_p - B * c1] +
                               shared_a[private_p - B * c1 + 1]) +
                               shared_a[private_p - B * c1 + 2]) / 3);
        __syncthreads();
    }
}

__global__ void kernel1(int *a, int *b, int N, int T, int ub_v,
                        int B, int c0) {

    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    int private_w;
    __shared__ int shared_b[BLOCK_0]; // BLOCK_0 = B

    for (int c1 = b0; c1 < ub_v; c1 += 32768) {
        if (N >= t0 + B * c1 + 2)
            shared_b[t0] = b[t0 + B * c1 + 1];
        __syncthreads();
        private_w = (((c1) * (B)) + (t0));
        a[private_w + 1] = shared_b[private_w - B * c1];
        __syncthreads();
    }
}

```

Figure 5.4: Generated parametric CUDA kernel for 1D Jacobi

In order to enable using the shared memory, we enforce PPCG to declare arrays in the shared memory anyway, while PPCG could decide not to use the shared memory because it fails to analyze non-linear expressions. However, this causes another issue, that is, how to calculate the index expressions of these arrays allocated in the shared memory. Recall that the generated kernel code should first copy the data from the global memory to its shared memory counterpart, and then use this shared memory counterpart to compute. In other words, to generate a copy statement, we shall insert into the schedule tree new nodes with the tiling size and affine transformation; however, these new nodes should be inserted to the schedule tree prior to where is the node referring to the statement with the arrays for computation. Using

```

if (T >= 1 && ub_v >= 1 && B >= 0) {
#define cudaCheckReturn(ret) \
do { \
    cudaError_t cudaCheckReturn_e = (ret); \
    if (cudaCheckReturn_e != cudaSuccess) { \
        fprintf(stderr, "CUDA error: %s\n", \
            cudaGetErrorString(cudaCheckReturn_e)); \
        fflush(stderr); \
    } \
    assert(cudaCheckReturn_e == cudaSuccess); \
} while(0)
#define cudaCheckKernel() \
do { \
    cudaCheckReturn(cudaGetLastError()); \
} while(0)

int *dev_a;
int *dev_b;

cudaCheckReturn(cudaMalloc((void **) &dev_a, (N) * sizeof(int)));
cudaCheckReturn(cudaMalloc((void **) &dev_b, (N) * sizeof(int)));

if (N >= 1) {
    cudaCheckReturn(cudaMemcpy(dev_a, a, (N) * sizeof(int),
                                cudaMemcpyHostToDevice));
    cudaCheckReturn(cudaMemcpy(dev_b, b, (N) * sizeof(int),
                                cudaMemcpyHostToDevice));
}
for (int c0 = 0; c0 < T; c0 += 1) {
    dim3 k0_dimBlock(B);
    dim3 k0_dimGrid(ub_v <= 32767 ? ub_v : 32768);
    kernel0 <<<k0_dimGrid, k0_dimBlock>>> (dev_a,dev_b,N,T,ub_v,B,c0);
    cudaCheckKernel();

    dim3 k1_dimBlock(B);
    dim3 k1_dimGrid(ub_v <= 32767 ? ub_v : 32768);
    kernel1 <<<k1_dimGrid, k1_dimBlock>>> (dev_a,dev_b,N,T,ub_v,B,c0);
    cudaCheckKernel();
}
if (N >= 1) {
    cudaCheckReturn(cudaMemcpy(a, dev_a, (N) * sizeof(int),
                                cudaMemcpyDeviceToHost));
    cudaCheckReturn(cudaMemcpy(b, dev_b, (N) * sizeof(int),
                                cudaMemcpyDeviceToHost));
}

cudaCheckReturn(cudaFree(dev_a));
cudaCheckReturn(cudaFree(dev_b));
}

```

Figure 5.5: Generated host code for 1D Jacobi

the 1D Jacobi code shown in Figure 5.3 as an example, we would like to insert a node in the schedule tree with the non-linear expression $v * B + u$, where the program parameter B specifies the thread-block format, so as to generate the copy statement `shared_a[c2] = a[B * c1 + c2]` or `shared_b[t0] = b[t0 + B * c1 + 1]` shown in Figure 5.4.

However, all basic data structures of PPCG do not support any non-linear expressions. For instance, one can use a data structure called `isl_aff` to store the affine transformation, such as $i * 32 + j$, while the pair (i, j) is stored by an internal data structure called `isl_space` and the pair $(32, 1)$ is stored by another internal data structure called `isl_set`, representing a vector of integers. Our solution to represent the non-linear expression:

$$v * B + u$$

is via a linear expression:

$$B + v * 1024 + u,$$

using a constant `1024` as the tile size but adding the program parameter B , such that the parametric information, especially the number B of threads per thread-block, is stored and passed to AST generation. Note that one can distinguish between the iteration counters, such as v and u , of loops and the program parameters, such as B , in the schedule tree. Next, during the phase of translating the AST to the CUDA code, whenever we recognize that particular format of linear expressions, we replace `1024` by B and remove B . By doing this, our prototype MetaFork-to-CUDA code generator admits the limitation to deal with other formats of non-linear expressions, which explains why post-processing is necessary for certain cases. Of course, improving this design is a work in progress so as to completely avoid post-processing.

5.4 Experimentation

In this section, we present experimental results on an NVIDIA Tesla M2050. Most of them were obtained by running times of CUDA programs generated with our preliminary implementation of our MetaFork-to-CUDA code generator described in Section 5.3, and the original version of the PPCG C-to-CUDA code generator [115]. We use eight simple test cases: *array reversal* (Figure 5.6, Table 5.1), *1D Jacobi* (Table 5.2), *2D Jacobi* (Figure 5.7, Table 5.3), *LU decomposition* (Figure 5.8, Table 5.4), *matrix transposition* (Figure 5.9, Table 5.5), *matrix addition* (Figure 5.10, Table 5.6), *matrix vector multiplication* (Figure 5.11, Table 5.7) and *matrix matrix multiplication* (Figure 5.13, Table 5.8). In all cases, we use dense representations for our matrices and vectors.

For both the PPCG C-to-CUDA and our MetaFork-to-CUDA code generators, Tables 5.1, 5.2, 5.3, 5.4 5.5, 5.6, 5.7 and 5.8 give the speedup factors of the generated code, as the timing ratio of the generated code to their untiled C code. Since PPCG determines a thread-block format, the timings in those tables corresponding to PPCG depend only on the input data size. Meanwhile, since the CUDA kernels generated by MetaFork are parametric, the MetaFork timings are obtained for various formats of thread-blocks and various input data sizes. Indeed, recall that our generated CUDA code admits parameters for the dimension sizes of the thread-blocks. This generated parametric code is then specialized with the thread-block formats listed in the first column of those tables.

Figures 5.6, 5.3 (with 5.2 and 5.4), 5.7, 5.8, 5.9, 5.10, 5.11 and 5.13 show the MetaFork code of eight examples with their untiled serial C programs and automatically generated CUDA kernels. In order to allocate unit sizes of shared memory in the kernel code, we predefine `BLOCK_0` and `BLOCK_1` (if applicable) as macros and specify their values at compile time. The tiled code for each MetaFork program is done by the quantifier elimination (QE) from the RegularChains library of MAPLE. These eight examples generated by PPCG are shown in Appendix D.

Array reversal. Both MetaFork and PPCG generate CUDA code that uses a one-dimensional kernel grid and the shared memory. We specialize the MetaFork generated parametric code successively to the thread-block size $B = 16, 32, 64, 128, 256, 512$; meanwhile, PPCG by default chooses 32 as the thread-block size. As we can see in Table 5.1, based on the generated parametric CUDA kernel, one can tune the thread-block size to be 256 to obtain the best performance.

Table 5.1: Speedup comparison of reversing a one-dimensional array between PPCG and MetaFork kernel code

Speedup (kernel)	Input size		
Thread-block size	2^{23}	2^{24}	2^{25}
PPCG			
32	8.312	8.121	8.204
MetaFork			
16	4.035	3.794	3.568
32	7.612	7.326	7.473
64	13.183	13.110	13.058
128	19.357	19.694	20.195
256	20.451	21.614	22.965
512	18.768	18.291	19.512

<pre> Serial code for (int i = 0; i < N; i++) Out[N - 1 - i] = In[i]; </pre>	<pre> __global__ void kernel0(int *In, int *Out, int N, int ub_v, int B) { int b0 = blockIdx.x; int t0 = threadIdx.x; int private_inoffset; int private_outoffset; __shared__ int shared_In[BLOCK_0]; // BLOCK_0 = B for (int c0 = b0; c0 < ub_v; c0 += 32768) { if (N >= t0 + B * c0 + 1) shared_In[t0] = In[t0 + B * c0]; __syncthreads(); private_inoffset = (((c0) * (B)) + (t0)); private_outoffset = (((N) - 1) - private_inoffset); Out[private_outoffset] = shared_In[private_inoffset - B * c0]; __syncthreads(); } } </pre>
<pre> MetaFork code int ub_v = N / B; meta_schedule { meta_for (int v = 0; v < ub_v; v++) meta_for (int u = 0; u < B; u++) { int inoffset = v * B + u; int outoffset = N - 1 - inoffset; Out[outoffset] = In[inoffset]; } } </pre>	

Figure 5.6: Serial code, MetaFork code and generated parametric CUDA kernel for array reversal

1D Jacobi. Our second example is a one-dimensional stencil computation, namely 1D Jacobi. The kernel generated by MetaFork uses a 1D kernel grid and the shared memory,

while the kernel generated by PPCG uses a 1D kernel grid and the global memory. PPCG by default chooses a thread-block format of 32, while MetaFork preferred format is 64.

Table 5.2: Speedup comparison of 1D Jacobi between PPCG and MetaFork kernel code

Speedup (kernel)	Input size		
Thread-block size kernel0, kernel1	$2^{13} + 2$	$2^{14} + 2$	$2^{15} + 2$
PPCG using the global memory			
32, 32	1.416	2.424	5.035
MetaFork			
16, 16	1.274	2.660	2.462
32, 32	1.967	3.386	5.268
64, 64	2.122	4.020	7.309
128, 128	1.787	3.234	6.168
256, 256	1.789	3.516	6.218
512, 512	2.193	3.518	6.070

2D Jacobi. Our next example is a two-dimensional stencil computation, namely 2D Jacobi. Both the CUDA kernels generated by MetaFork and PPCG use a 2D kernel grid and the global memory. PPCG by default chooses a thread-block format of 16×32 , while MetaFork preferred format varies based on input size.

Table 5.3: Speedup comparison of 2D Jacobi between PPCG and MetaFork kernel code

Speedup (kernel)	Input size		
Thread-block size	$(2^{12} + 2)^2$	$(2^{13} + 2)^2$	$(2^{14} + 2)^2$
PPCG			
16 * 32	11.230	11.303	9.785
MetaFork			
8 * 4	5.000	5.256	4.666
16 * 4	7.867	8.724	7.962
32 * 4	11.607	11.143	9.726
8 * 8	7.209	7.776	6.704
16 * 8	10.499	10.502	7.442
32 * 8	12.236	11.487	9.182
8 * 16	8.859	8.825	5.637
16 * 16	10.774	10.709	7.694
32 * 16	11.969	11.442	10.469

LU decomposition. MetaFork and PPCG both generate two CUDA kernels: one with a 1D grid and one with a 2D grid, both using the shared memory. The default selected thread-block formats for PPCG are 32 and 16×32 ; meanwhile, the preferred formats by MetaFork are 128 and 16×16 . Tuning the number of threads per thread-block in our parametric code allows MetaFork to outperform PPCG.

Matrix transpose. Both the CUDA kernels generated by MetaFork and PPCG use a 2D grid and the shared memory. PPCG by default chooses a thread-block format of 16×32 , while

```

Serial code

for (int t = 0; t < T; t++) {
  for (int i = 1; i < N-1; i++)
    for (int j = 1; j < N-1; j++)
      b[i][j] = (a[i-1][j] + a[i+1][j]
        + a[i][j-1] + a[i][j+1]) / 4;
  for (int i = 1; i < N-1; ++i)
    for (int j = 1; j < N-1; j++)
      a[i][j] = b[i][j];
}

MetaFork code

int dim0 = (N-2)/B0, dim1 = (N-2)/B1;
meta_schedule {
  for (int t = 0; t < T; t++) {
    meta_for (int v0=0; v0<dim0; v0++)
      meta_for (int v1= 0; v1<dim1; v1++)
        meta_for (int u0=0; u0<B0; u0++)
          meta_for (int u1=0; u1<B1; u1++) {
            int p = v0 * B0 + u0;
            int w = v1 * B1 + u1;
            b[p+1][w+1] = (a[p][w+1] +
              a[p+2][w+1] + a[p+1][w] +
              a[p+1][w+2]) / 4;
          }
        meta_for (int v0=0; v0<dim0; v0++)
          meta_for (int v1=0; v1<dim1; v1++)
            meta_for (int u0=0; u0<B0; u0++)
              meta_for (int u1=0; u1<B1; u1++)
                {
                  int i = v0 * B0 + u0;
                  int j = v1 * B1 + u1;
                  a[i+1][j+1] = b[i+1][j+1];
                }
          }
  }
}

__global__ void kernel0(int *a, int *b, int N, int T, int dim0,
  int dim1, int B0, int B1, int c0) {
  int b0 = blockIdx.y, b1 = blockIdx.x;
  int t0 = threadIdx.y, t1 = threadIdx.x;
  int private_p;
  int private_w;

  for (int c1 = b0; c1 < dim0; c1 += 256)
    for (int c2 = b1; c2 < dim1; c2 += 256) {
      private_p = (((c1) * (B0)) + (t0));
      private_w = (((c2) * (B1)) + (t1));
      b[(private_p + 1) * N + (private_w + 1)] =
        (((a[private_p * N + (private_w + 1)] +
          a[(private_p + 2) * N + (private_w + 1)])
          + a[(private_p + 1) * N + private_w]) +
          a[(private_p + 1) * N + (private_w + 2)]) / 4);
      __syncthreads();
    }
}

__global__ void kernel1(int *a, int *b, int N, int T, int dim0,
  int dim1, int B0, int B1, int c0) {
  int b0 = blockIdx.y, b1 = blockIdx.x;
  int t0 = threadIdx.y, t1 = threadIdx.x;
  int private_i;
  int private_j;

  for (int c1 = b0; c1 < dim0; c1 += 256)
    for (int c2 = b1; c2 < dim1; c2 += 256) {
      private_i = (((c1) * (B0)) + (t0));
      private_j = (((c2) * (B1)) + (t1));
      a[(private_i + 1) * N + (private_j + 1)] =
        b[(private_i + 1) * N + (private_j + 1)];
      __syncthreads();
    }
}

```

Figure 5.7: Serial code, MetaFork code and generated parametric CUDA kernel for 2D Jacobi

MetaFork preferred format is 8×32 . For MetaFork, the allocation unit size of shared memory for the input matrix is the same as thread-block format. However, for PPCG, the allocation unit size of shared memory for the input matrix is 32×32 , while the thread-block format is 16×32 . Thus, PPCG code transposes two coefficients of the matrix within each thread.

Matrix addition. Both the CUDA kernels generated by MetaFork and PPCG use a 2D grid and the global memory. The default chosen thread-block format for PPCG is 16×32 , while MetaFork preferred format is 32×8 .

Matrix vector multiplication. For both MetaFork and PPCG, the generated kernels use a 1D grid and the shared memory. The thread-block size chosen by PPCG is 32, while MetaFork preferred thread-block size varies based on different input sizes. PPCG has a slight advantage due to its ability here to analyze shared and local memory usage for the 2D array in a 1D kernel. For MetaFork, to enhance some components of the PPCG infrastructure with symbolic computation is required, which is a work in progress.

Within the current framework, we post-process the kernel code, such that coalesced accesses occur for copying the 2D array from the global memory to the shared memory. Fig-

Table 5.4: Speedup comparison of LU decomposition between PPCG and MetaFork kernel code

Speedup (kernel)			Input size	
Thread-block size kernel0, kernel1			$2^{10} * 2^{10}$	$2^{11} * 2^{11}$
PPCG				
32,	16	* 32	10.712	30.329
MetaFork				
128,	4	* 4	3.063	15.512
256,	4	* 4	3.077	15.532
512,	4	* 4	3.095	15.572
32,	8	* 8	10.721	37.727
64,	8	* 8	10.604	37.861
128,	8	* 8	10.463	37.936
256,	8	* 8	10.831	37.398
512,	8	* 8	10.416	37.840
32,	16	* 16	14.533	54.121
64,	16	* 16	14.457	54.034
128,	16	* 16	14.877	54.447
256,	16	* 16	14.803	53.662
512,	16	* 16	14.479	53.077

Table 5.5: Speedup comparison of matrix transpose between PPCG and MetaFork kernel code

Speedup (kernel)			Input size	
Thread-block size			$2^{13} * 2^{13}$	$2^{14} * 2^{14}$
PPCG				
16	*	32	62.656	103.703
MetaFork				
8	*	4	28.626	37.681
16	*	4	40.381	41.403
32	*	4	28.728	30.329
8	*	8	51.889	58.789
16	*	8	44.759	52.137
32	*	8	37.586	43.696
8	*	16	70.716	76.781
16	*	16	64.812	73.657
32	*	16	36.109	59.613
8	*	32	77.327	93.051
16	*	32	62.268	77.399

Figure 5.12 shows the post-processed CUDA kernel code. On line 16 for copying the input matrix from the global memory to the array, namely `shared_a`, allocated in the shared memory, we exchanged the row index with the column index of this array `shared_a`, so that threads within a warp access to the corresponding global memory in a coalesced manner. Table 5.7 shows the speedup factors obtained with the kernels generated by PPCG, MetaFork and MetaFork with

```

Serial code
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n-k-1; i++) {
        // column major representation
        // of L and U
        int p = i + k + 1;
        L[k][p] = U[k][p] / U[k][k];
        for (int j = k; j < n; j++)
            U[j][p] -= L[k][p] * U[j][k];
    }
}

MetaFork code
int ub = n / B, ut = n / Sqrt_T;
meta_schedule {
    for (int k = 0; k < n-1; k++) {
        meta_for (int bx = 0; bx < ub; bx++)
            meta_for (int ux = 0; ux < B; ux++)
                if ((k + 1 - bx * B < B) &&
                    (-B * bx + k < ux) &&
                    (ux < n - bx * B)) {
                    int l = bx * B + ux;
                    L[k][l] = U[k][l] / U[k][k];
                }

        meta_for (int bx = 0; bx < ut; bx++)
            meta_for (int by = 0; by < ut; by++)
                meta_for (int ux = 0; ux < Sqrt_T; ux++)
                    meta_for (int uy = 0;
                               uy < Sqrt_T; uy++) {
                        int i = by * Sqrt_T + uy;
                        if (i < n - k - 1) {
                            int j = bx * Sqrt_T + ux;
                            if (j < n - k) {
                                U[j+k][i+k+1] -=
                                    L[k][i+k+1] * U[j+k][k];
                            }
                        }
                    }
    }
}

__global__ void kernel0(double *L, double *U, int n, int ut,
                        int Sqrt_T, int ub, int B, int c0) {
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    int private_l;
    __shared__ double shared_U_1[1][1];

    {
        if (t0 == 0)
            shared_U_1[0][0] = U[c0 * n + c0];
        __syncthreads();
        for (int c1 = b0; c1 < ub; c1 += 32768) {
            if ((((((c0) + 1) - ((c1) * (B))) < (B)) &&
                (((- (B)) * (c1)) + (c0)) < (t0))) &&
                ((t0) < ((n) - ((c1) * (B)))))) {
                private_l = ((c1) * (B)) + (t0);
                L[c0 * n + private_l] =
                    (U[c0 * n + private_l] / shared_U_1[0][0]);
            }
        }
        __syncthreads();
    }
}

__global__ void kernel1(double *L, double *U, int n, int ut,
                        int Sqrt_T, int ub, int B, int c0) {
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    int private_i;
    int private_j;
    // BLOCK_0 = BLOCK_1 = Sqrt_T
    __shared__ double shared_L[1][BLOCK_1];
    __shared__ double shared_U_1[BLOCK_0][1];

    for (int c1 = b0; c1 < ut; c1 += 256) {
        if (t1 == 0 && n >= t0 + c0 + Sqrt_T * c1 + 1)
            shared_U_1[t0][0] = U[(t0 + c0 + Sqrt_T * c1) * n + c0];
        for (int c2 = b1; c2 < ut; c2 += 256) {
            if (t0 == 0 && n >= t1 + c0 + Sqrt_T * c2 + 2)
                shared_L[0][t1] =
                    L[c0 * n + (t1 + c0 + Sqrt_T * c2 + 1)];
            __syncthreads();
            private_i = (((c2) * (Sqrt_T)) + (t1));
            if (private_i < ((n) - (c0)) - 1) {
                private_j = ((c1) * (Sqrt_T)) + (t0);
                if (private_j < ((n) - (c0))) {
                    U[(private_j + c0) * n + (private_i + c0 + 1)] -=
                        (shared_L[0][private_i - Sqrt_T * c2] *
                         shared_U_1[private_j - Sqrt_T * c1][0]);
                }
            }
        }
        __syncthreads();
    }
}

```

Figure 5.8: Serial code, MetaFork code and generated parametric CUDA kernel for LU decomposition

post-processing, respectively. One can see that the performance of the parametric kernel with coalesced accesses is twice as good as that of the automatically generated kernels by MetaFork and PPCG.

Matrix matrix multiplication. For both MetaFork and PPCG, the generated kernels use a 2D grid and the shared memory. The thread-block size chosen by PPCG is 16×32 , while Meta-

```

Serial code
for (int v0 = 0; v0 < n; v0++)
  for (int v1 = 0; v1 < n; v1++)
    c[v0][v1] = a[v1][v0];

MetaFork code
int dim0 = n / B0, dim1 = n / B1;
meta_schedule {
  meta_for (int v0 = 0; v0 < dim0; v0++)
    meta_for (int v1 = 0; v1 < dim1; v1++)
      meta_for (int u0 = 0; u0 < B0; u0++)
        meta_for (int u1 = 0; u1 < B1; u1++)
          {
            int i = u0 + v0 * B0;
            int j = u1 + v1 * B1;
            c[j][i] = a[i][j];
          }
}

__global__ void kernel0(int *a, int *c, int n, int dim0,
                       int dim1, int B0, int B1) {
  int b0 = blockIdx.y, b1 = blockIdx.x;
  int t0 = threadIdx.y, t1 = threadIdx.x;
  int private_i;
  int private_j;
  // BLOCK_0 = B0, BLOCK_1 = B1
  __shared__ int shared_a[BLOCK_0][BLOCK_1];

  for (int c0 = b0; c0 < dim0; c0 += 256)
    for (int c1 = b1; c1 < dim1; c1 += 256) {
      if (n >= t0 + B0 * c0 + 1 && n >= t1 + B1 * c1 + 1)
        shared_a[t0][t1] =
          a[(t0 + B0 * c0) * n + (t1 + B1 * c1)];
      __syncthreads();
      private_i = ((t0) + ((c0) * (B0)));
      private_j = ((t1) + ((c1) * (B1)));
      c[private_j * n + private_i] =
        shared_a[private_i - B0 * c0][private_j - B1 * c1];
      __syncthreads();
    }
}

```

Figure 5.9: Serial code, MetaFork code and generated parametric CUDA kernel for matrix transpose

Table 5.6: Speedup comparison of matrix addition between PPCG and MetaFork kernel code

Speedup (kernel)			Input size	
Thread-block size			2^{12}	2^{13}
PPCG				
16	*	32	13.024	9.750
MetaFork				
8	*	4	19.520	20.329
16	*	4	32.971	35.227
32	*	4	54.233	49.734
8	*	8	28.186	30.221
16	*	8	44.783	42.008
32	*	8	56.650	50.547
8	*	16	33.936	32.793
16	*	16	45.015	41.606
32	*	16	54.426	47.930

Fork preferred thread-block size varies based on input sizes. For MetaFork, the allocation unit size of shared memory for each input matrix is the same as thread-block format. However, for PPCG, the allocation unit size of shared memory for each input matrix is 32×32 , while the thread-block format is 16×32 . In fact, PPCG code computes two coefficients of the output matrix within each thread, thus increasing index arithmetic amortization and occupancy.

We conclude this section with timings (in seconds) for the quantifier elimination (QE) required to generate MetaFork tiled code, see Table 5.9. Our tests are based on the latest version of the RegularChains library of MAPLE, available at www.regularchains.org. These results show that the use of QE is not a bottleneck in our C-to-CUDA code translation process, despite the theoretically high algebraic complexity of quantifier elimination.

```

Serial code
for (int v0 = 0; v0 < n; v0++)
  for (int v1 = 0; v1 < n; v1++)
    c[v0][v1] = a[v0][v1] + b[v0][v1];

MetaFork code
int dim0 = n / B0, dim1 = n / B1;
meta_schedule {
  meta_for (int v0 = 0; v0 < dim0; v0++)
    meta_for (int v1 = 0; v1 < dim1; v1++)
      meta_for (int u0 = 0; u0 < B0; u0++)
        meta_for (int u1 = 0; u1 < B1; u1++)
          {
            int i = u0 + v0 * B0;
            int j = u1 + v1 * B1;
            c[i][j] = a[i][j] + b[i][j];
          }
}

__global__ void kernel0(int *a, int *b, int *c, int n,
                        int dim0, int dim1, int B0, int B1) {
  int b0 = blockIdx.y, b1 = blockIdx.x;
  int t0 = threadIdx.y, t1 = threadIdx.x;
  int private_i;
  int private_j;

  for (int c0 = b0; c0 < dim0; c0 += 256)
    for (int c1 = b1; c1 < dim1; c1 += 256) {
      private_i = ((t0) + ((c0) * (B0)));
      private_j = ((t1) + ((c1) * (B1)));
      c[private_i * n + private_j] =
        (a[private_i * n + private_j] +
         b[private_i * n + private_j]);
      __syncthreads();
    }
}

```

Figure 5.10: Serial code, MetaFork code and generated parametric CUDA kernel for matrix addition

Table 5.7: Speedup comparison of matrix vector multiplication among PPCG kernel code, MetaFork kernel code and MetaFork kernel code with post-processing

Speedup (kernel)	Input size		
Thread-block size	2^{11}	2^{12}	2^{13}
PPCG			
32	3.954	3.977	5.270
MetaFork			
16	3.108	3.535	3.856
32	4.116	3.550	3.782
64	2.955	3.744	2.996
128	2.658	2.582	2.491
256	2.215	1.599	1.813
MetaFork with post-processing			
16	4.976	6.260	7.794
32	8.698	6.911	10.340
64	4.260	5.567	6.683

5.5 Conclusion

In this chapter, we have presented enhancements of the MetaFork language so as to provide the model of concurrency for SIMD on GPUs. Our objective is to facilitate automatic code translation of high-level programming models supporting hardware accelerator (like OpenMP and OpenACC) to low-level heterogeneous programming models (like CUDA).

As illustrated in Sections 5.3, MetaFork has language constructs to help generate efficient CUDA code. Moreover, the MetaFork framework relies on advanced techniques (quantifier elimination in non-linear polynomial expressions) for code optimization, in particular tiling. The experimentation reported in Section 5.4 shows the benefits of generating *parametric* CUDA kernels. Not only does this feature provide more portability, but it also helps to improve per-

```

Serial code
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    c[i] += a[i][j] * b[j];

MetaFork code
int dim = n / B;
meta_schedule {
  meta_for (int v = 0; v < dim; v++)
    for (int i = 0; i < n / 16; ++i)
      meta_for (int u = 0; u < B; u++)
        for (int j = 0; j < 16; ++j) {
          int p = v * B + u;
          c[p] += a[p][i*16+j]*b[i*16+j];
        }
}

__global__ void kernel0(int *a, int *b, int *c, int n, int dim,
                      int B) {

  int b0 = blockIdx.x;
  int t0 = threadIdx.x;
  int private_p;
  // BLOCK_0 = B
  __shared__ int shared_a[BLOCK_0][16];
  __shared__ int shared_b[16];
  __shared__ int shared_c[BLOCK_0];

  for (int c0 = b0; c0 < dim; c0 += 32768) {
    if (n >= t0 + B * c0 + 1)
      shared_c[t0] = c[t0 + B * c0];
    for (int c1 = 0; c1 < n / 16; c1 += 1) {
      if (n >= t0 + B * c0 + 1)
        for (int c3 = 0; c3 <= 15; c3 += 1)
          shared_a[t0][c3] =
            a[(t0 + B * c0) * n + (16 * c1 + c3)];
      if (t0 <= 15)
        shared_b[t0] = b[t0 + 16 * c1];
      __syncthreads();
      for (int c3 = 0; c3 <= 15; c3 += 1) {
        private_p = (((c0) * (B)) + (t0));
        shared_c[private_p - B * c0] +=
          (shared_a[private_p - B * c0][c3] * shared_b[c3]);
      }
      __syncthreads();
    }
    if (n >= t0 + B * c0 + 1)
      c[t0 + B * c0] = shared_c[t0];
    __syncthreads();
  }
}

```

Figure 5.11: Serial code, MetaFork code and generated parametric CUDA kernel for matrix vector multiplication

Table 5.8: Speedup comparison of matrix multiplication between PPCG and MetaFork kernel code

Speedup (kernel)			Input size	
Thread-block size			$2^{10} * 2^{10}$	$2^{11} * 2^{11}$
PPCG				
16	*	32	129.853	393.851
MetaFork				
8	*	4	32.157	96.652
16	*	4	54.578	171.621
32	*	4	53.399	156.493
8	*	8	60.358	182.557
16	*	8	87.919	287.002
32	*	8	84.057	289.930
8	*	16	100.521	299.228
16	*	16	100.264	330.965
32	*	16	85.928	247.220

formance with automatically generated code.

```

1  __global__ void kernel0(int *a, int *b, int *c, int n, int dim, int B) {
2      int b0 = blockIdx.x;
3      int t0 = threadIdx.x;
4      int private_p;
5      // BLOCK_0 = B
6      __shared__ int shared_a[BLOCK_0][BLOCK_0];
7      __shared__ int shared_b[BLOCK_0];
8      __shared__ int shared_c[BLOCK_0];
9
10     for (int c0 = b0; c0 < dim; c0 += 32768) {
11         if (n >= t0 + B * c0 + 1)
12             shared_c[t0] = c[t0 + B * c0];
13         for (int c1 = 0; c1 < n / BLOCK_0; c1 += 1) {
14             if (n >= t0 + B * c0 + 1)
15                 for (int c3 = 0; c3 < BLOCK_0; c3 += 1)
16                     shared_a[c3][t0] = a[(c3 + B * c0) * n + (B * c1 + t0)];
17             shared_b[t0] = b[t0 + B * c1];
18             __syncthreads();
19             for (int c3 = 0; c3 < BLOCK_0; c3 += 1) {
20                 private_p = (((c0) * (B)) + (t0));
21                 shared_c[private_p - B * c0] +=
22                     (shared_a[private_p - B * c0][c3] * shared_b[c3]);
23             }
24             __syncthreads();
25         }
26         if (n >= t0 + B * c0 + 1)
27             c[t0 + B * c0] = shared_c[t0];
28         __syncthreads();
29     }
30 }

```

Figure 5.12: Post-processing CUDA kernel with coalesced accesses for matrix vector multiplication

Table 5.9: Timings (in sec.) of quantifier elimination for eight examples

Example	Timing
Array reversal	0.072
1D Jacobi	0.948
2D Jacobi	7.735
LU decomposition	4.416
matrix transposition	1.314
matrix addition	1.314
matrix vector multiplication	0.072
matrix matrix multiplication	2.849


```

Serial code
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    for (int k = 0; k < n; ++k)
      c[i][j] += a[i][k] * b[k][j];

MetaFork code
int dim0 = n / B0, dim1 = n / B1;
meta_schedule {
  meta_for (int i = 0; i < dim0; i++)
    meta_for (int j = 0; j < dim1; j++)
      for (int k = 0; k < n/4; k++)
        meta_for (int v = 0; v < B0; v++)
          meta_for (int u = 0; u < B1; u++)
            {
              int p = i * B0 + v;
              int w = j * B1 + u;
              for (int z = 0; z < 4; z++)
                c[p][w] +=
                  a[p][4*k+z] * b[4*k+z][w];
            }
}

__global__ void kernel0(int *a, int *b, int *c, int n, int dim0,
                        int dim1, int B0, int B1) {
  int b0 = blockIdx.y, b1 = blockIdx.x;
  int t0 = threadIdx.y, t1 = threadIdx.x;
  int private_p;
  int private_w;
  // BLOCK_0 = B0, BLOCK_1 = B1
  __shared__ int shared_a[BLOCK_0][4];
  __shared__ int shared_b[4][BLOCK_1];
  __shared__ int shared_c[BLOCK_0][BLOCK_1];

  for (int c0 = b0; c0 < dim0; c0 += 256)
    for (int c1 = b1; c1 < dim1; c1 += 256) {
      if (n >= t0 + B0 * c0 + 1 && n >= t1 + B1 * c1 + 1)
        shared_c[t0][t1] =
          c[(t0 + B0 * c0) * n + (t1 + B1 * c1)];
      for (int c2 = 0; c2 < n / 4; c2 += 1) {
        if (t1 <= 3 && n >= t0 + B0 * c0 + 1)
          shared_a[t0][t1] =
            a[(t0 + B0 * c0) * n + (t1 + 4 * c2)];
        if (t0 <= 3 && n >= t1 + B1 * c1 + 1)
          shared_b[t0][t1] =
            b[(t0 + 4 * c2) * n + (t1 + B1 * c1)];
        __syncthreads();
        private_p = (((c0) * (B0)) + (t0));
        private_w = (((c1) * (B1)) + (t1));
        for (int c5 = 0; c5 <= 3; c5 += 1)
          shared_c[private_p - B0 * c0][private_w - B1 * c1] +=
            (shared_a[private_p - B0 * c0][c5] *
             shared_b[c5][private_w - B1 * c1]);
        __syncthreads();
      }
      if (n >= t0 + B0 * c0 + 1 && n >= t1 + B1 * c1 + 1)
        c[(t0 + B0 * c0) * n + (t1 + B1 * c1)] =
          shared_c[t0][t1];
      __syncthreads();
    }
}

```

Figure 5.13: Serial code, MetaFork code and generated parametric CUDA kernel for matrix matrix multiplication

Chapter 6

Generation of Optimized CUDA Kernel Code

In the previous chapter, we observed that, for several test cases, such as matrix multiplication and matrix transposition, the parametric CUDA kernels generated by MetaFork could not outperform those that are generated by PPCG, for which thread-block formats are given by numerical values. This observation motivates the work reported in this chapter. In Section 6.1, we revisit a particular test case, matrix multiplication, by exploring various optimization techniques, such as controlling the granularity of threads and using the local memory of the device. Since accessing shared (resp. local) memory has low latency, one can consider to declare an array allocated in the shared (resp. local) memory for temporarily storing the corresponding elements of the array in the global memory. In the sequel of this chapter, we refer such an array allocated in the shared (resp. local) memory as the shared (resp. local) memory counterpart of the array in the global memory. Furthermore, we apply those optimization techniques to five additional test cases in Section 6.2, so as to evaluate the benefits of those optimization techniques. All experimental results are collected on an NVIDIA Tesla M2050.

6.1 Case study: matrix multiplication

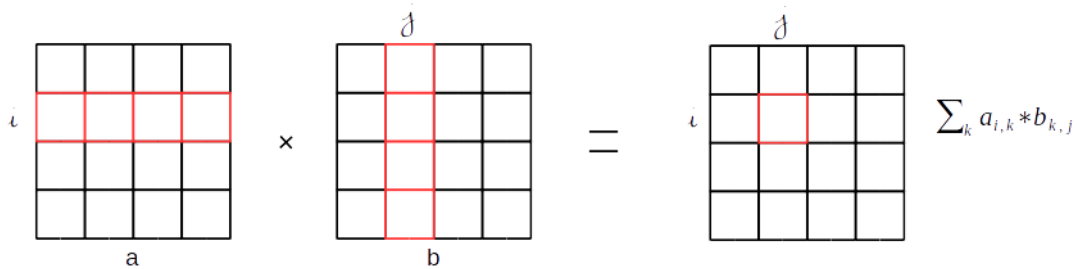


Figure 6.1: Multiplication of two matrices

The MetaFork code for matrix multiplication, shown in Figure 5.13 of Chapter 5, follows the naive implementation described in [73]. To be more precise, each thread-block computes

a block of the output matrix, while each thread computes one element of the output matrix. Figure 6.1 shows the formula to calculate the multiplication of two matrices with each block of size 1. Recall that when each input matrix is of order 2^{10} , the best speedup factor that MetaFork achieves is 100.521 for the thread-block format of size 8×16 , while PPCG achieves the speedup factor of 129.853 for the thread-block format of size 16×32 . Our generated kernel code uses the shared memory for each of the input and output matrices. However, we observe that PPCG generates a local memory counterpart of size 2 for the output matrix and two shared memory counterparts, each of size 32×32 , for input matrices. This means that for each thread, the kernel code generated by PPCG computes two elements of the output matrix.

Some studies [73, 118] demonstrate that better performance can be obtained, when one tunes the granularity of threads to increase the arithmetic intensity and hide the data transfer time. This suggests that the amount of work that each thread executes can become a program parameter. We can tune this parameter as we do with the thread-block format parameters. Thus, we introduce a granularity loop into the original MetaFork code as shown in Figure 6.2, where the w-loop on Line (13) increases the work per thread by a parameter s . Let each thread-block of the original MetaFork code compute $B_0 \times B_1$ elements of the output matrix. Then, for the new code with the granularity loop, each thread-block computes the same number of elements of the output matrix, while each thread computes s elements and the number of threads per thread-block reduces to $B_0 \times B_1 / s$ (that is, $B_0 \times ub_1$ as shown in Figure 6.2). In addition, we guarantee that the data, which threads access to, has a good alignment during each iteration of the w-loop, such that coalesced accesses from a warp of threads occur.

```

1 // n * n matrices
2 // Program parameters: B0, ub1, s
3 assert(BLOCK == min(B0, ub1 * s));
4 int dim0 = n / B0, dim1 = n / (ub1 * s);
5
6 meta_schedule {
7     meta_for (int i = 0; i < dim0; i++)
8         meta_for (int j = 0; j < dim1; j++)
9             for (int k = 0; k < n / BLOCK; ++k)
10                 meta_for (int v = 0; v < B0; v++)
11                     meta_for (int u = 0; u < ub1; u++)
12                         // Each thread computes BLOCK*s outputs
13                         for (int w = 0; w < s; ++w) {
14                             int p = i * B0 + v;
15                             int q = j * ub1 * s + w * ub1 + u;
16                             for (int z = 0; z < BLOCK; z++)
17                                 c[p][q] += a[p][BLOCK*k+z] * b[BLOCK*k+z][q];
18                         }
19 }

```

Figure 6.2: The MetaFork code with the granularity loop and good data alignment for matrix multiplication

Introducing the granularity loop leads to a non-linear expression

$$j * ub1 * s + w * ub1 + u$$

on Line (15) in Figure 6.2, which is used as a column index of the two-dimensional arrays `b` and `c`. The current MetaFork-to-CUDA code generator does not support this format of non-linear expressions yet. In order to see the benefits of using the granularity loop with good data alignment, we manually process the generated kernel code to obtain the new kernel code shown in Figure 6.3. In the first scenario, we continue to use the shared memory for input and output matrices in the generated kernel code. During the post-processing phase, we modify the indices of arrays allocated in the global and shared memories, respectively. Then we collect the speedup factors and achieved occupancies in Table 6.1, for various values of the number of threads per thread-block and the granularity of threads. For the best case, when the thread-block format is 8×32 and the granularity of threads is 4, the kernel code uses 23 registers and 8448 bytes of shared memory. We observe that this kernel achieves better performance, when compared to those kernels automatically generated by PPCG and MetaFork.

Table 6.1: Experimental results of matrix multiplication for the CUDA kernel with the shared memory for the output matrix and the granularity of threads

(a) speedup factors

Thread-block \ Granularity	2	4	8
(8, 8)	99.422	92.612	100.406
(16, 8)	130.053	143.995	130.213
(32, 8)	138.445	152.369	106.153
(64, 8)	124.984	128.164	84.301

(b) achieved occupancies

Thread-block \ Granularity	2	4	8
(8, 8)	0.332	0.330	0.326
(16, 8)	0.660	0.654	0.407
(32, 8)	0.823	0.813	0.332
(64, 8)	0.662	0.658	0.333

Input size $2^{10} * 2^{10}$

In the second scenario, we use the local memory, instead of the shared memory, for the output matrix. Thereby, we replace the code

```
__shared__ int shared_c[BLOCK_0][BLOCK_1]
```

on Line (11) in Figure 6.3 by

```
int private_c [1][STRIDE],
```

where `STRIDE` is predefined as a macro and is equal to s . For the array references

```
shared_c[t0][c5*ub1+t1],
```

which appeared at Lines (19), (30) and (38) on Figure 6.3, we replace each array reference by

```
private_c[0][c5].
```

```

1  __global__ void kernel0(int *a, int *b, int *c, int n, int dim0, int dim1,
2                                int B0, int ub1, int s) {
3      int b0 = blockIdx.y, b1 = blockIdx.x;
4      int t0 = threadIdx.y, t1 = threadIdx.x;
5      int private_p;
6      int private_q;
7      assert (BLOCK == min(B0, ub1 * s));
8      assert (BLOCK_0 == B0); assert (BLOCK_1 == ub1 * s);
9      __shared__ int shared_a[BLOCK_0][BLOCK];
10     __shared__ int shared_b[BLOCK][BLOCK_1];
11     __shared__ int shared_c[BLOCK_0][BLOCK_1];
12
13     for (int c0 = b0; c0 < dim0; c0 += 256)
14         for (int c1 = b1; c1 < dim1; c1 += 256) {
15             private_p = ((c0) * (B0)) + (t0);
16             private_q = (c1) * (ub1 * s) + (t1);
17             for (int c5 = 0; c5 < s; c5 += 1)
18                 if (n >= private_p + 1 && n >= private_q + (c5) * (ub1) + 1)
19                     shared_c[t0][c5*ub1+t1] =
20                         c[(private_p) * n + (private_q + (c5) * (ub1))];
21             for (int c2 = 0; c2 < n / BLOCK; c2 += 1) {
22                 if (t1 < BLOCK && n >= private_p + 1)
23                     shared_a[t0][t1] = a[(private_p) * n + (t1 + BLOCK * c2)];
24                 for (int c5 = 0; c5 < s; c5 += 1) {
25                     if (t0 < BLOCK && n >= private_q + (c5) * (ub1) + 1)
26                         shared_b[t0][(c5) * (ub1) + t1] =
27                             b[(t0 + BLOCK * c2) * n + (private_q + (c5) * (ub1))];
28                     __syncthreads();
29                     for (int c6 = 0; c6 < BLOCK; c6 += 1)
30                         shared_c[t0][c5*ub1+t1] +=
31                             (shared_a[t0][c6] * shared_b[c6][c5 * ub1 + t1]);
32                 }
33                 __syncthreads();
34             }
35             for (int c5 = 0; c5 < s; c5 += 1)
36                 if (n >= private_p + 1 && n >= private_q + (c5) * (ub1) + 1)
37                     c[(private_p) * n + (private_q + (c5) * (ub1))] =
38                         shared_c[t0][c5*ub1+t1];
39             __syncthreads();
40         }
41     }

```

Figure 6.3: Post-processing the generated CUDA kernel code for matrix multiplication with the granularity loop

We collect the experimental results in Table 6.2, which contains the speedup factors and achieved occupancies for various values of the number of threads per thread-block and the granularity of threads. For the best case, when the thread-block format is 8×32 and the granularity of threads is 4, the kernel code uses 16 bytes of stack frame, 22 registers and 2176 bytes of shared memory. However, the performance is worse than that of the kernel using the shared memory for the output matrix.

Table 6.2: Experimental results of matrix multiplication for the CUDA kernel with the local memory for the output matrix and the granularity of threads

(a) speedup factors			
Thread-block \ Granularity	2	4	8
(8, 8)	87.652	93.717	95.032
(16, 8)	127.22	133.620	127.204
(32, 8)	133.312	142.351	125.563
(64, 8)	114.836	124.092	116.646

(b) achieved occupancies			
Thread-block \ Granularity	2	4	8
(8, 8)	0.332	0.331	0.327
(16, 8)	0.661	0.655	0.647
(32, 8)	0.824	0.815	0.804
(64, 8)	0.662	0.658	0.651

Input size $2^{10} * 2^{10}$

One key reason explaining the relatively low performance of our kernel code using the local memory is the fact that the upper bound of the granularity loop is a variable instead of a constant. Indeed, this prevents the compiler from applying loop unrolling, which is an essential trick for exposing instruction level parallelism (ILP). Hence, we shall use a constant as the w -loop's upper bound at Line (13) on Figure 6.2. Correspondingly, we modify the generated kernel code at Line (24) on Figure 6.3 by changing the variable s to the constant `STRIDE`. Note that `STRIDE`, `BLOCK_0` and `BLOCK_1` are constants that are passed by the compilation flags `-DSTRIDE`, `-DBLOCK_0` and `-DBLOCK_1`, respectively. Then we collect the speedup factors for various values of the granularity size and the thread-block format in Table 6.3. Due to the loop unrolling, the usage of registers per thread increases from 22 to 27. At this point, the performance of the modified kernel has been further improved, compared to the kernel in the first scenario, that is, using the shared memory for the output matrix.

Table 6.3: For input matrices of order 2^{10} , speedup factors of the matrix multiplication kernel unrolling the computation

Thread-block \ Granularity	2	4	Resource usage
(16, 4)	79.038	89.233	26 registers, 2112 bytes of shared memory
(32, 4)	104.898	123.262	
(64, 4)	106.147	111.481	
(8, 8)	116.090	137.318	27 registers, 2304 bytes of shared memory
(16, 8)	165.192	194.326	
(32, 8)	166.310	180.835	
(64, 8)	137.937	161.536	

With the intention to maximize ILP, we unroll the loop for the copy-in and copy-out phases of the data transfer between the global memory and the local memory. Then, we modify the upper bounds on Lines (17) and (35) by changing the variable s to the constant `STRIDE`, so as to obtain the final modified kernel code as shown in Figure 6.4. Then we collect the experimental results in Table 6.4 for various values of the granularity size and the thread-block format. We

```

__global__ void kernel0(int *a, int *b, int *c, int n, int dim0, int dim1,
                        int B0, int ub1, int s) {

    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    int private_p;
    int private_q;
    assert (BLOCK == min(B0, ub1 * s));
    assert (BLOCK_0 == B0); assert (BLOCK_1 == ub1 * s);
    assert (STRIDE == s);
    __shared__ int shared_a[BLOCK_0][BLOCK];
    __shared__ int shared_b[BLOCK][BLOCK_1];
    int private_c[1][STRIDE];

    for (int c0 = b0; c0 < dim0; c0 += 256)
        for (int c1 = b1; c1 < dim1; c1 += 256) {
            private_p = (c0) * (B0) + (t0);
            private_q = (c1) * (ub1 * s) + (t1);
            for (int c5 = 0; c5 < STRIDE; c5 += 1)
                if (n >= private_p + 1 && n >= private_q + (c5) * (ub1) + 1)
                    private_c[0][c5] =
                        c[(private_p) * n + (private_q + (c5) * (ub1))];
            for (int c2 = 0; c2 < n / BLOCK; c2 += 1) {
                if (t1 < BLOCK && n >= private_p + 1)
                    shared_a[t0][t1] = a[(private_p) * n + (t1 + BLOCK * c2)];
                for (int c5 = 0; c5 < s; c5 += 1)
                    if (t0 < BLOCK && n >= private_q + (c5) * (ub1) + 1)
                        shared_b[t0][(c5) * (ub1) + t1] =
                            b[(t0 + BLOCK * c2) * n + (private_q + (c5) * (ub1))];
                __syncthreads();
                for (int c6 = 0; c6 < BLOCK; c6 += 1)
                    for (int c5 = 0; c5 < STRIDE; c5 += 1)
                        private_c[0][c5] +=
                            (shared_a[t0][c6] * shared_b[c6][c5 * ub1 + t1]);
                __syncthreads();
            }
            for (int c5 = 0; c5 < STRIDE; c5 += 1)
                if (n >= private_p + 1 && n >= private_q + (c5) * (ub1) + 1)
                    c[(private_p) * n + (private_q + (c5) * (ub1))] =
                        private_c[0][c5];
            __syncthreads();
        }
}

```

Figure 6.4: CUDA kernel with unrolling the granularity loop for matrix multiplication

observe that each thread, based on the value of B_0 , uses 38 or 34 registers, which are more than the default allowance, that is, 32 per thread, set by the compiler. In this case, register spilling [95] occurs, which causes the data to be accessed in the L2 cache, and slows down the execution time. Meanwhile, the nvcc [38] compiler for CUDA programs allows the programmer to set a number for the number of registers that each thread can use, as long as this number does not exceed the hardware limit (which is 64 on the NVIDIA Tesla M2050). Thus, we pass

a compilation flag `--maxrregcount=40` to allow each thread to use 40 registers. In the end, we obtain a significant improvement in performance.

Table 6.4: For input matrices of order 2^{10} , speedup factors of the matrix multiplication kernel unrolling the copy-in, computation and copy-out phases with a compilation flag `--maxrregcount=40`

Thread-block \ Granularity	2	4	Resource usage
(16, 4)	117.032	151.667	38 registers, 2112 bytes of shared memory
(32, 4)	151.555	184.812	
(64, 4)	132.588	171.805	
(8, 8)	145.972	178.111	34 registers, 2304 bytes of shared memory
(16, 8)	196.625	230.870	
(32, 8)	193.753	226.934	
(64, 8)	111.214	169.528	

Moreover, we realize that we compared our kernel code with the naive, serial matrix multiplication (shown in Figure 5.13 of Chapter 5), which lacks good data locality. Thus, using a blocking strategy, we implement the version of the serial C code as shown in Figure 6.5, which has a nearly optimal cache complexity. Finally, we collect speedup factors for the two kernels generated by PPCG and MetaFork against this latter C code. Of course, the MetaFork-generated kernel is the one described above. In particular, a post-processing phase is required for unrolling the granularity loop and using the local memory. Experimental results are collected in Table 6.5. For the best case, the MetaFork-generated kernel outperforms by a factor of 1.7 (resp. 1.9) the PPCG-generated kernel for matrices of order 2^{10} (resp. 2^{11}).

```

for (int i = 0; i < n; i += 32)
  for (int j = 0; j < n; j += 32)
    for (int k = 0; k < n; k += 32)
      for (int i0 = i; i0 < min(i + 32, n); i0++)
        for (int j0 = j; j0 < min(j + 32, n); j0++)
          for (int k0 = k; k0 < min(k + 32, n); k0++)
            c[i0][j0] += a[i0][k0] * b[k0][j0];

```

Figure 6.5: The serial C code with good data locality for matrix multiplication

6.2 Experimentation

In this section, we experiment with the granularity loop on the MetaFork code of five simple test cases: array reversal (Figure 6.6, Table 6.6), 1D Jacobi (Figure 6.7, Table 6.7), matrix addition (Figure 6.8, Table 6.8), matrix transposition (Figure 6.9, Table 6.9) and matrix vector multiplication (Figure 6.10, Table 6.10). For each test case, we compare the kernel code with the granularity loop against the kernel code without the granularity loop, that is, the code automatically generated in Chapter 5.

For matrix addition, the kernel uses the global memory for the input and output matrices. For matrix vector multiplication, we generate a local memory counterpart for the output vector

Table 6.5: Speedup factors obtained with kernels generated by PPCG and MetaFork with post-processing, respectively, w.r.t. the serial C code with good data locality for matrix multiplication

Speedup (kernel)	Input size			
Thread-block size	$2^{10} * 2^{10}$		$2^{11} * 2^{11}$	
PPCG				
(16, 32)	109		105	
MetaFork with post-processing				
	Granularity			
	2	4	2	4
(16, 4)	95	128	90	119
(32, 4)	128	157	125	144
(64, 4)	111	145	105	132
(8, 8)	131	151	126	146
(16, 8)	164	194	159	188
(32, 8)	163	187	158	202
(64, 8)	94	143	104	135

and two shared memory counterparts for both the input matrix and the input vector. For the other test cases, we generate shared memory counterparts for the input vectors or matrices and continue to use the global memory for the output. In the case that vectors or matrices are required to copy from/to the arrays allocated in the shared or local memory, post-processing is required. Figures 6.6, 6.7, 6.8, 6.9 and 6.10 show the MetaFork code and its kernel code for each test case. Tables 6.6, 6.7, 6.8 6.9 and 6.10 collect the speedup factors for various values of one thread-block dimension and its granularity of threads.

Array reversal. For the input vector of length 2^{25} , the kernel code generated in Chapter 5 achieves the best speedup factor of 22.965 for the thread-block format of size 256, while for the same thread-block format but with the granularity 4, the kernel with the granularity loop performs slightly better.

Table 6.6: Speedup factors of reversing a one-dimensional array for input vector of length 2^{25}

Thread-block \ Granularity	2	4	8
16	5.239	5.561	5.743
32	7.867	7.959	8.176
64	13.814	13.987	14.232
128	20.354	20.872	21.036
256	23.554	24.511	

1D Jacobi. For the input vector of length $2^{15} + 2$, the parametric kernels generated in Chapter 5 achieve the best speedup factor of 7.309 with the thread-block format of size 64, while for the thread-block format of size 128 and the granularity 2, the kernels with the granularity loop perform slightly better.

Matrix addition. For the input matrix of order 2^{12} , the kernel code generated in Chapter 5 achieves the best speedup factor of 56.650 for the thread-block format of size 32×8 , while

```

// Array with of size N
// Program parameters: B, s
int ub_v = N / B, ub_u = B / s;

meta_schedule {
    meta_for (int v = 0; v < ub_v; v++)
        meta_for (int u = 0; u < ub_u; u++)
            for (int w = 0; w < s; ++w) {
                int x = v * B + w * ub_u + u;
                int y = N - 1 - x;
                Out[y] = In[x];
            }
}

__global__ void kernel0(int *In, int *Out, int N, int ub_v,
                       int ub_u, int s, int B) {
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    int private_x;
    int private_y;
    __shared__ int shared_In[BLOCK_0]; // BLOCK_0 = B

    for (int c0 = b0; c0 < ub_v; c0 += 32768) {
        for (int c2 = 0; c2 < s; c2 += 1) {
            private_x = (c0) * (B) + (c2) * (ub_u) + (t0);
            if (N >= private_x + 1)
                shared_In[(c2) * (ub_u) + t0] = In[private_x];
            __syncthreads();
            private_y = ((N) - 1) - private_x;
            Out[private_y] = shared_In[(c2) * (ub_u) + t0];
        }
        __syncthreads();
    }
}

```

Figure 6.6: The MetaFork code and its kernel code with the granularity loop for reversing a one-dimensional array

Table 6.7: Speedup factors of 1D Jacobi for time iteration 4 and input vector of length $2^{15}+2$

Thread-block \ Granularity	2	4	8
16	3.340	4.357	4.975
32	4.785	5.252	5.206
64	5.927	6.264	6.412
128	10.400	8.952	5.793
256	6.859	6.246	

for the thread-block format of size 64×8 and the granularity 2, the kernel with the granularity loop performs almost the same.

Table 6.8: Speedup factors of matrix addition for input matrix of order 2^{12}

Thread-block \ Granularity	2	4	8
(4, 8)		21.561	23.844
(8, 8)	34.144	37.196	34.187
(16, 8)	45.906	43.287	41.956
(32, 8)	55.219	52.543	51.304
(64, 8)	56.815	55.843	54.402
(128, 8)	39.128	44.819	

Matrix transpose. For the input matrix of order 2^{14} , the kernel code generated in Chapter 5 achieves the best speedup factor of 93.051 for the thread-block format of size 8×32 , while for the same thread-block format but with the granularity 2, the kernel with the granularity loop outperforms those two kernels automatically generated by PPCG and MetaFork.

Matrix vector multiplication. For the input matrix of order 2^{13} and the input vector of length 2^{13} , the kernel code generated in Chapter 5 achieves the best speedup factor of 10.340 for the thread-block format of size 32. In this test case, since we use the local memory for the output vector, a constant is used as the upper bound of the granularity loop so as to allow loop

```

// Array with of size N + 2
// Program parameters: B, s
int ub_v = (N - 2) / B, ub_u = B / s;
meta_schedule {
    for (int t = 0; t < T; ++t) {
        meta_for (int v = 0; v < ub_v; v++)
            meta_for (int u = 0; u < ub_u; u++)
                for (int i = 0; i < s; ++i) {
                    int p = v * B + i * ub_u + u;
                    b[p+1] = (a[p]+a[p+1]+a[p+2])/3;
                }
        meta_for (int v = 0; v < ub_v; v++)
            meta_for (int u = 0; u < B; u++) {
                int w = v * B + u;
                a[w+1] = b[w+1];
            }
    }
}

__global__ void kernel0(int *a, int *b, int N, int T, int ub_v,
                        int B, int ub_u, int s, int c0) {
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    int private_p;
    __shared__ int shared_a[BLOCK_0+2]; // BLOCK_0 = B

#define min(x,y) ((x) < (y) ? (x) : (y))
    for (int c1 = b0; c1 < ub_v; c1 += 32768) {
        for (int c2 = t0; c2 <= min(B + 1, N - B * c1 - 1);
            c2 += ub_u)
            shared_a[c2] = a[B * c1 + c2];
        __syncthreads();
        private_p = (c1) * (B) + t0;
        for (int c3 = 0; c3 < s; c3 += 1) {
            b[private_p + (c3) * (ub_u) + 1] =
                (((shared_a[(c3) * (ub_u) + t0]
                  + shared_a[(c3) * (ub_u) + t0 + 1])
                  + shared_a[(c3) * (ub_u) + t0 + 2]) / 3);
        }
        __syncthreads();
    }
}

__global__ void kernel1(int *a, int *b, int N, int T, int ub_v,
                        int B, int ub_u, int s, int c0) {
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    int private_w;

    for (int c1 = b0; c1 < ub_v; c1 += 32768) {
        private_w = (((c1) * (B)) + (t0));
        a[private_w + 1] = b[private_w + 1];
        __syncthreads();
    }
}

```

Figure 6.7: The MetaFork code and its kernel code with the granularity loop for 1D Jacobi

```

// n * n matrices
// Program parameters: B0, ub_u1, s
int dim0 = n/B0, dim1 = n/B1, ub_u1 = B1/s;
meta_schedule {
    meta_for (int v0 = 0; v0 < dim0; v0++)
        meta_for (int v1 = 0; v1 < dim1; v1++)
            meta_for (int u0 = 0; u0 < B0; u0++)
                meta_for (int u1=0; u1<ub_u1; u1++)
                    for (int w = 0; w < s; ++w) {
                        int i = v0 * B0 + u0;
                        int j = v1 * B1 + w*ub_u1 + u1;
                        c[i][j] = a[i][j] + b[i][j];
                    }
}

__global__ void kernel0(int *a, int *b, int *c, int n, int dim0,
                        int dim1, int B0, int ub_u1, int s, int B1) {
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    int private_i;
    int private_j;

    for (int c0 = b0; c0 < dim0; c0 += 256)
        for (int c1 = b1; c1 < dim1; c1 += 256) {
            for (int c4 = 0; c4 < s; c4 += 1) {
                private_i = ((t0) + ((c0) * (B0)));
                private_j = (((c1) * (B1)) + ((c4) * (ub_u1))) + (t1));
                c[private_i * n + private_j] =
                    (a[private_i * n + private_j] +
                     b[private_i * n + private_j]);
            }
            __syncthreads();
        }
}

```

Figure 6.8: The MetaFork code and its kernel code with the granularity loop for matrix addition

unrolling for the copy-in, computation and copy-out phases. For the thread-block format of size 32 and the granularity 2, this kernel uses 22 registers and 8320 bytes of shared memory, while for the same thread-block format but without the granularity loop, the kernel automatically

Table 6.9: Speedup factors of matrix transpose for input matrix of order 2^{14}

Thread-block \ Granularity	2	4	8
(4, 32)	103.281	96.284	75.211
(8, 32)	111.971	90.625	85.422
(16, 32)	78.476	68.894	48.822
(32, 32)	45.084	46.425	32.824

```

__global__ void kernel0(int *a, int *c, int n, int dim0,
                        int dim1, int B0, int ub_u1, int s, int B1) {
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    int private_i;
    int private_j;
    // BLOCK_0 = B0, BLOCK_1 = B1
    __shared__ int shared_a[BLOCK_0][BLOCK_1];

    // n * n matrices
    // Program parameters: B0, ub_u1, s
    int dim0 = n/B0, dim1 = n/B1, ub_u1 = B1/s;

    meta_schedule {
        meta_for (int v0 = 0; v0 < dim0; v0++)
            meta_for (int v1 = 0; v1 < dim1; v1++)
                meta_for (int u0 = 0; u0 < B0; u0++)
                    meta_for (int u1=0; u1<ub_u1; u1++)
                        for (int w = 0; w < s; ++w) {
                            int i = v0 * B0 + u0;
                            int j = v1 * B1 + w * ub_u1 + u1;
                            c[j][i] = a[i][j];
                        }
    }

    for (int c0 = b0; c0 < dim0; c0 += 256)
        for (int c1 = b1; c1 < dim1; c1 += 256) {
            private_i = (((c0) * (B0)) + (t0));
            if (n >= t0 + B0 * c0 + 1 && n >= t1 + B1 * c1 + 1)
                for (int c4 = 0; c4 < s; c4 += 1)
                    shared_a[t0][t1+(c4)*(ub_u1)] =
                        a[(private_i) * n + (t1 + c4 * ub_u1 + B1 * c1)];
            __syncthreads();
            for (int c4 = 0; c4 < s; c4 += 1) {
                private_j = ((c1) * (B1)) + ((c4) * (ub_u1)) + (t1);
                c[private_j * n + private_i] =
                    shared_a[private_i - B0 * c0][private_j - B1 * c1];
            }
            __syncthreads();
        }
}

```

Figure 6.9: The MetaFork code and its kernel code with the granularity loop for matrix transpose

generated by MetaFork uses 29 registers and 4352 bytes of shared memory. However, the kernel with the granularity loop performs worse.

Table 6.10: Speedup factors of matrix vector multiplication for input matrix of order 2^{13} and input vector of length 2^{13} (An error indicates that the total amount of required shared memory exceeds the hardware limit.)

Thread-block \ Granularity	2	4	8
8	2.975	2.677	1.974
16	4.281	3.424	3.377
32	5.477	1.916	0.988
64	2.286	Error	Error

6.3 Conclusion

We have experimented with thread granularity control for six test cases. For two of these test cases, matrix multiplication and matrix vector multiplication, we use the local memory for the

```

// N * N matrix and N vector
// Program parameters: B, s
// BLOCK = B / s, STRIDE = s
int ub_v = N / B, ub_u = B / s;
meta_schedule {
    meta_for (int v = 0; v < ub_v; v++)
        for (int i = 0; i < n / BLOCK; ++i)
            meta_for (int u = 0; u < ub_u; u++)
                for (int j = 0; j < BLOCK; ++j)
                    for (int w = 0; w < STRIDE; ++w)
                        {
                            int p = v * B + w * ub_u + u;
                            c[p] += a[p][i*BLOCK+j] *
                                b[i*BLOCK+j];
                        }
}

__global__ void kernel0(int *a, int *b, int *c, int N, int ub_v,
                        int ub_u, int s, int B) {
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    // BLOCK_0 = B0, STRIDE = s
    __shared__ int shared_a[BLOCK_0][BLOCK_0/STRIDE];
    __shared__ int shared_b[BLOCK_0/STRIDE];
    int private_c[STRIDE];

    for (int c0 = b0; c0 < ub_v; c0 += 32768) {
        for (int c4 = 0; c4 < STRIDE; c4 += 1)
            private_c[c4] = c[ub_u * c4 + c0 * B + t0];
        for (int c1 = 0; c1 < N / ub_u; c1 += 1) {
            for (int c4 = 0; c4 < s; c4 += 1)
                for (int c3 = 0; c3 < ub_u; c3 += 1)
                    shared_a[c4 * ub_u + c3][t0] =
                        a[(ub_u * c1 + c0 * B + c3) * N + (ub_u * c1 + t0)];
                    shared_b[t0] = b[t0 + ub_u * c1];
                __syncthreads();
                for (int c3 = 0; c3 < ub_u; c3 += 1)
                    for (int c4 = 0; c4 < STRIDE; c4 += 1)
                        private_c[c4] +=
                            (shared_a[c4 * ub_u + t0][c3] * shared_b[c3]);
                    __syncthreads();
                }
            for (int c4 = 0; c4 < STRIDE; c4 += 1)
                c[ub_u * c4 + c0 * B + t0] = private_c[c4];
            __syncthreads();
        }
    }
}

```

Figure 6.10: The MetaFork code and its kernel code with the granularity loop for matrix vector multiplication

output matrix or vector in order to expose ILP by helping the compiler to apply loop unrolling. We believe that these techniques could help the performance of CUDA kernels. Although matrix transposition and matrix multiplication show a significant improvement, matrix vector multiplication performs worse, and other test cases behave slightly better or almost the same as the original kernel without the granularity loop control. The question whether we should apply those techniques in process of automatic code generation remains unanswered. This leads to the next chapter where we discuss an algorithm to generate a case distinction depending on available hardware resources.

However, adding a granularity loop in the MetaFork code violates those assumptions listed in Appendix C. Thus, for using the shared memory, post-processing is required to obtain correct index expressions of the shared memory counterparts. This manual modification is difficult and painful, especially for non-CUDA experts. To fully support any non-linear expressions in the current MetaFork-to-CUDA code generator, simply hacking into PPCG source code is not enough to solve the issue. We leave it to future work.

Moreover, our automatic code generator described in Chapter 5 does not support using the local memory. We could extend our code generator to allow the user to choose whether to use the local memory in the kernel code. Alternatively, we could automatically detect the situations when the local memory should be used. To be more specific, based on those test cases that we experienced, we can use the local memory for the output array when each element in this array is accessed more than once by each thread.

Chapter 7

Towards Comprehensive Parametric CUDA Kernel Generation

In Chapter 5, we demonstrated that, from an annotated C code, it was possible to generate CUDA kernels that depend on program parameters considered unknown at compile-time. Our experimental results in Chapters 5 and 6 suggest that those *parametric CUDA kernels* could help with increasing portability and performance of CUDA code.

In the present chapter, we enhance this strategy as follows. First, we propose an algorithm for *comprehensive optimization* allowing us to optimize C code (and in particular CUDA code) depending on unknown machine and program parameters. Then, we use this algorithm to generate optimized parametric CUDA kernels, in the form of a case distinction based on the possible values of the machine and program parameters. We call *comprehensive parametric CUDA kernels* the resulting CUDA kernels, see Section 7.2.

In broad terms, this is a decision tree, where each edge holds a Boolean expression, given by a conjunction of polynomial constraints, and each leaf is either a CUDA kernel or the symbol \emptyset , such that for each leaf K , with $K \neq \emptyset$, we have:

1. K works correctly under the conjunction of the Boolean expressions located between the root node and the leaf, and
2. K is semantically equivalent to a common input annotated C code \mathcal{P} .

In each Boolean expression, the unknown variables represent machine parameters (like hardware resource limits), program parameters (like dimension sizes of thread-blocks) or data parameters (like input data size). The symbol \emptyset is used to denote a situation (in fact, value ranges for the machine and program parameters) where no CUDA kernel equivalent to \mathcal{P} is provided.

The intention, with the concept of comprehensive parametric CUDA kernels, is to automatically generate optimized CUDA kernels from an annotated C code without knowing the numerical values of some, or all, of the machine and program parameters. This naturally yields a case distinction depending on the values of those parameters. Indeed, some optimization techniques (like loop unrolling) can only be applied when enough computing resources are available, while other optimization techniques (like common sub-expression elimination) can be applied to reduce computing resource consumption. These case distinctions can be handled by techniques from symbolic computation, for which software libraries are available, in particular, the RegularChains library freely available at www.regularchains.org.

Let us illustrate, with an example, the notion of comprehensive parametric CUDA kernels,

along with a procedure to generate them. For computing the sum of two matrices a and b of order N , our input is the `meta_for`-loop nest within the `meta_schedule` statement on the right-hand portion of Figure 7.1, whereas the serial code without tiling is provided on the left-hand portion of Figure 7.1.

<pre> for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) c[i][j] = a[i][j] + b[i][j]; </pre>	<pre> int dim0 = N/B0, dim1 = N/(2*B1); meta_schedule { meta_for (int v = 0; v < dim0; v++) meta_for (int p = 0; p < dim1; p++) meta_for (int u = 0; u < B0; u++) meta_for (int q = 0; q < B1; q++) { int i = v * B0 + u; int j = p * B1 + q; if (i < N && j < N/2) { c[i][j] = a[i][j] + b[i][j]; c[i][j+N/2] = a[i][j+N/2] + b[i][j+N/2]; } } } </pre>
---	--

(a) Before tiling, the C program

(b) After tiling, the MetaFork program

Figure 7.1: Matrix addition written in C (the left-hand portion) and in MetaFork (the right-hand portion) with a `meta_for` loop nest, respectively

We make the following simplistic assumptions for the translation of this `meta_for`-loop nest to a CUDA program.

1. The target machine has two parameters: the maximum number R_1 of registers per thread, and the maximum number R_2 of threads per thread-block; moreover, all other hardware limits are ignored.
2. The generated kernels depend on two program parameters, B_0 and B_1 , which define the format of a 2D thread-block.
3. The optimization strategy (w.r.t. register usage per thread) consists in reducing the work per thread via removing the 2-way loop unrolling [123].

The possible comprehensive parametric CUDA kernels are given by the pairs (C_1, K_1) and (C_2, K_2) , where C_1, C_2 are two sets of algebraic constraints on the machine and program parameters and K_1, K_2 are two CUDA kernels that are optimized under the constraints, respectively, given by C_1, C_2 , see Figure 7.2. The following computational steps yield the pairs (C_1, K_1) and (C_2, K_2) .

- (S1) Tiling techniques, based on quantifier elimination (QE), are applied to the `meta_for` loop nest of Figure 7.1 in order to decompose the matrices into tiles of format $B_0 \times B_1$, see [19] for details.
- (S2) The tiled MetaFork code is mapped to an intermediate representation (IR) say that of LLVM¹, or alternatively, to PTX² code.

¹ Quoting Wikipedia: “The LLVM compiler infrastructure project (formerly Low Level Virtual Machine [75, 9]) is a framework for developing compiler front ends and back ends”.

²The *Parallel Thread Execution* (PTX) [39] is the pseudo-assembly language to which CUDA programs are

- (S3) Using this IR (or PTX) code, one can *estimate* the number of registers that a thread requires; thus, using LLVM IR on this example, we obtain an estimate of 14.
- (S4) Next, we apply the optimization strategy, yielding a new IR (or PTX) code, for which register pressure reduces to 10. Since no other optimization techniques are considered, the procedure stops with the result shown in Figure 7.2.

Based on the above steps, Figure 7.3 shows the decision tree for generating these two pairs, each of them consisting of a system of polynomial constraints and a CUDA kernel for matrix addition. Note that, strictly speaking, the kernels K_1 and K_2 on Figure 7.2 should be given by PTX code. But for simplicity, we are presenting them by the CUDA code counterpart.

$$C_1 : \begin{cases} B_0 \times B_1 \leq R_1 \\ 14 \leq R_2 \end{cases}$$

```

__global__ void K1(int *a, int *b, int *c, int N,
                  int B0, int B1) {
    int i = blockIdx.y * B0 + threadIdx.y;
    int j = blockIdx.x * B1 + threadIdx.x;
    if (i < N && j < N/2) {
        a[i*N+j] = b[i*N+j] + c[i*N+j];
        a[i*N+j+N/2] = b[i*N+j+N/2] + c[i*N+j+N/2];
    }
}
dim3 dimBlock(B1, B0);
dim3 dimGrid(N/(2*B1), N/B0);
K1 <<<dimGrid, dimBlock>>> (a, b, c, N, B0, B1);

```

$$C_2 : \begin{cases} B_0 \times B_1 \leq R_1 \\ 10 \leq R_2 < 14 \end{cases}$$

```

__global__ void K2(int *a, int *b, int *c, int N,
                  int B0, int B1) {
    int i = blockIdx.y * B0 + threadIdx.y;
    int j = blockIdx.x * B1 + threadIdx.x;
    if (i < N && j < N)
        a[i*N+j] = b[i*N+j] + c[i*N+j];
}
dim3 dimBlock(B1, B0);
dim3 dimGrid(N/B1, N/B0);
K2 <<<dimGrid, dimBlock>>> (a, b, c, N, B0, B1);

```

Figure 7.2: Comprehensive translation of MetaFork code to two kernels for matrix addition

One can observe that loop unrolling is applied to K_1 so as to increase arithmetic intensity. This code transformation increases register pressure, which is possible under the constraints of C_1 but not under those of C_2 .

In general, to achieve a *comprehensive translation* of the annotated C program \mathcal{P} into CUDA kernels, one could be tempted to proceed as follows:

- (S1) Perform a *comprehensive optimization* of \mathcal{P} , as a MetaFork program, by applying the comprehensive optimization algorithm demonstrated in Section 7.1.
- (S2) Apply the MetaFork-to-CUDA code generator introduced in Chapter 5 to each MetaFork program generated in the first step.

However, since the system of polynomial constraints associated with each optimized MetaFork program is determined by an IR representation of that program, this system would not be accurate for the CUDA code generated by a source-to-source MetaFork-to-CUDA code generator.

compiled by NVIDIA's `nvcc` compiler. PTX code can also be generated from (enhanced) LLVM IR, using `nvptx` back-end [68], following the work of [99].

and such code transformation is valid.

Section 7.1.1 states the hypotheses made on the input code fragment. Section 7.1.2 specifies the notations for the hardware characteristics of the targeted device. In Section 7.1.3, we describe the evaluation of resource and performance counters. In Section 7.1.4, we define the optimization strategies that can reduce resource counters or increase performance counters. Section 7.1.5 formally gives the definition of comprehensive optimization of an input code fragment. Section 7.1.6 specifies the data structures that are used in our algorithm. Finally, in Section 7.1.7, we demonstrate our algorithm for comprehensive optimization.

7.1.1 Hypotheses on the input code fragment

We consider a sequence S of statements from the C programming language and introduce the following.

Definition 1 We call parameter of S any scalar variable that is

- (i) read in S at least once and
- (ii) never written in S .

We call data of S any non-scalar variable (e.g. array) that is not initialized but possibly overwritten within S . If a parameter of S gives a dimension size of a data of S , then this parameter is called a data parameter; otherwise, it is simply called a program parameter.

Notation 1 We denote by D_1, \dots, D_u and E_1, \dots, E_v the data parameters and program parameters of S , respectively.

Hypothesis 3 We make the following assumptions on S .

- (H1) All parameters are assumed to be non-negative integers.
- (H2) We assume that S can be viewed as the body of a valid C function having the parameters and data of S as unique arguments.

Example 3 S can be the body of a kernel function in CUDA. Recall that the kernel code for computing matrix vector multiplication in Figure 5.12 of Chapter 5. This kernel code multiplies a square matrix a of order n with a vector b of length n and stores the result to a vector c of length n . We note that a , b and c are the data, and that n is the data parameter. Moreover, the grid and thread-block dimensions of this kernel are specified as dim and B , respectively, which are then the program parameters.

7.1.2 Hardware resource limits and performance measures

We denote by R_1, \dots, R_s the hardware resource limits of the targeted hardware device. Examples of these quantities for the NVIDIA Kepler micro-architecture are:

- the maximum number of registers to be allocated per thread,
- the maximum number of shared memory words to be allocated per thread-block,
- the maximum number of threads in a thread-block.

We denote by P_1, \dots, P_t the performance measures of a program running on the device. These are dimensionless quantities typically defined as percentages. Examples of these quantities for the NVIDIA Kepler micro-architecture are:

- the ratio of the actual to the maximum number of words that can be read or written per unit of time from the global memory,
- the ratio of the actual to the maximum number of floating point operations that can be performed per unit of time by all streaming processors (SMs),
- the SM occupancy, that is, the ratio of active warps to the maximum number of active warps,
- the cache hit rate in an SM.

For a given hardware device, R_1, \dots, R_s are positive integers, and each of them is the maximum value of a hardware resource. Meanwhile, P_1, \dots, P_t are rational numbers between 0 and 1. However, for the purpose of writing code portable across a variety of devices with similar characteristics, the quantities R_1, \dots, R_s and P_1, \dots, P_t will be treated as unknown and independent variables. These hardware resource limits and performance measures will be called the *machine parameters*.

Each function K (and, in particular, our input code fragment S) written in the C language for the targeted hardware device has *resource counters* r_1, \dots, r_s and *performance counters* p_1, \dots, p_t corresponding, respectively, to R_1, \dots, R_s and P_1, \dots, P_t . In other words, the quantities r_1, \dots, r_s are the amounts of resources, corresponding to R_1, \dots, R_s , respectively, that K requires for executing. Similarly, the quantities p_1, \dots, p_t are the performance measures, corresponding to P_1, \dots, P_t , respectively, that K exhibits when executing. Therefore, the inequalities $0 \leq r_1 \leq R_1, \dots, 0 \leq r_s \leq R_s$ must hold for the function K to execute correctly. Similarly, $0 \leq p_1 \leq 1, \dots, 0 \leq p_t \leq 1$ are satisfied by the definition of the performance measures.

Remark 1 We note that $r_1, \dots, r_s, p_1, \dots, p_t$ may be numerical values, which we can assume to be non-negative rational numbers. This will be the case, for instance, for the minimum number of registers required per thread in a thread-block. The resource counters r_1, \dots, r_s may also be polynomial expressions whose indeterminate variables can be program parameters (like the dimension sizes of a thread-block or grid) or data parameters (like the input data sizes). Meanwhile, the performance counters p_1, \dots, p_t may further depend on the hardware resource limits (like the maximum number of active warps supported by an SM). To summarize, we observe that r_1, \dots, r_s are polynomials in $\mathbb{Q}[D_1, \dots, D_u, E_1, \dots, E_v]$ and p_1, \dots, p_t are rational functions where numerators and denominators are in $\mathbb{Q}[D_1, \dots, D_u, E_1, \dots, E_v, R_1, \dots, R_s]$. Moreover, we can assume that the denominators of those rational functions are positive.

Example 4 For computing the product of a dense square matrix of order N by a dense vector of length N , consider the serial C code on the left-hand portion of Figure 7.4 and the corresponding MetaFork code on the right-hand portion of Figure 7.4. The `meta.schedule` statement yields the CUDA kernel shown in Figure 5.12 of Chapter 5. The grid of this kernel is one-dimensional of size `dim`, and its thread-blocks are also one-dimensional, each counting `B` threads.

Observe that, in the process of generating this CUDA kernel, some optimization techniques (like using the shared memory for arrays a , b and c in Figure 5.12 of Chapter 5) are applied, while other optimization techniques remain to be applied (like the granularity of threads as used in Figure 6.10 of Chapter 6). If R_2 and R_3 are two machine parameters that define, respectively, the maximum number of threads in a thread-block and the maximum number of shared memory words per thread-block, then the following constraints must hold:

$$B \leq R_2 \text{ and } B^2 + 2B \leq R_3.$$

<pre> for (int p = 0; p < N; p++) for (int q = 0; q < N; q++) c[p] += a[p][q] * b[q]; </pre>	<pre> int dim = N / B; meta_schedule { meta_for (int v = 0; v < dim; v++) for (int i = 0; i < dim; ++i) meta_for (int u = 0; u < B; u++) for (int j = 0; j < B; ++j) { int p = v * B + u; int q = i * B + j; c[p] += a[p][q] * b[q]; } } </pre>
--	---

(a) Before tiling, the C program

(b) After tiling, the MetaFork program

Figure 7.4: Matrix vector multiplication written in C (the left-hand portion) and in MetaFork (the right-hand portion), respectively

Note that, in Figure 5.12 of Chapter 5, each thread-block allocates a unit size B^2 of shared memory words for the array a , and a unit size B of shared memory words, respectively, for each of arrays b and c . Thus, this leads to the inequality $B^2 + 2B \leq R_3$.

7.1.3 Evaluation of resource and performance counters

Let $G_C(S)$ be the control flow graph (CFG) [123] of S . Hence, the statements in the basic blocks of $G_C(S)$ are C statements, and we call such a CFG the *source CFG*. We also map S to an intermediate representation, which, itself, is encoded in the form of a CFG, denoted by $G_L(S)$, and we call it the *IR CFG*. Here, we refer to the landmark textbook [2] for the notion of the control flow graph and that of intermediate representation.

We observe that S can trivially be reconstructed from $G_C(S)$; hence, the knowledge of S and that of $G_C(S)$ can be regarded as equivalent. In contrast, $G_L(S)$ depends not only on S but also on the optimization strategies that are applied to the IR of S .

Equipped with $G_C(S)$ and $G_L(S)$, we assume that we can estimate each of the resource counters r_1, \dots, r_s (resp. performance counters p_1, \dots, p_t) by applying functions f_1, \dots, f_s (resp. g_1, \dots, g_t) to either $G_C(S)$ or $G_L(S)$. We call f_1, \dots, f_s (resp. g_1, \dots, g_t) the *resource* (resp. *performance*) evaluation functions.

For instance, when S is the body of a CUDA kernel and S reads (resp. writes) a given array, computing the total amount of elements read (resp. written) by one thread-block can be determined from $G_C(S)$. Meanwhile, computing the minimum number of registers to be allocated to a thread executing S requires the knowledge of $G_L(S)$.

7.1.4 Optimization strategies

In order to reduce the consumption of hardware resources and increase performance counters, we assume that we have w optimization procedures O_1, \dots, O_w , each of them mapping either a source CFG to another source CFG, or an IR CFG to another IR CFG. Of course, we assume the code transformations performed by O_1, \dots, O_w preserve semantics.

We associate each resource counter r_i , for $i = 1 \dots s$, with a non-empty subset $\sigma(r_i)$ of $\{O_1, \dots, O_w\}$, such that we have

$$f_i(O(S)) \leq f_i(S) \text{ for } O \in \sigma(r_i). \quad (7.1)$$

Hence, $\sigma(r_i)$ is a subset of the optimization strategies among O_1, \dots, O_w that have the potential to reduce r_i . Of course, the intention is that for at least one $O \in \sigma(r_i)$, we have $f_i(O(S)) < f_i(S)$. A reason for not finding such O would be that S cannot be further optimized w.r.t. r_i . We also make a natural *idempotence* assumption:

$$f_i(O(O(S))) = f_i(O(S)) \text{ for } O \in \sigma(r_i). \quad (7.2)$$

Similarly, we associate each performance counter p_i , for $i = 1 \dots t$, with a non-empty subset $\sigma(p_i)$ of $\{O_1, \dots, O_w\}$, such that we have

$$g_i(O(S)) \geq g_i(S) \text{ and } g_i(O(O(S))) = g_i(O(S)) \text{ for } O \in \sigma(p_i). \quad (7.3)$$

Hence, $\sigma(p_i)$ is a subset of the optimization strategies among O_1, \dots, O_w that have the potential to increase p_i . The intention is, again, that for at least one $O \in \sigma(p_i)$, we have $g_i(O(S)) > g_i(S)$.

7.1.5 Comprehensive optimization

Let C_1, \dots, C_e be semi-algebraic systems with $P_1, \dots, P_t, R_1, \dots, R_s, D_1, \dots, D_u, E_1, \dots, E_v$ as indeterminate variables. Let S_1, \dots, S_e be fragments of C programs such that the parameters of each of them are among $D_1, \dots, D_u, E_1, \dots, E_v$.

Definition 2 We say that the sequence of pairs $(C_1, S_1), \dots, (C_e, S_e)$ is a comprehensive optimization of S w.r.t.

- the resource evaluation functions f_1, \dots, f_s ,
- the performance evaluation functions g_1, \dots, g_t and
- the optimization strategies O_1, \dots, O_w

if the following conditions hold:

- (i) [constraint soundness] Each of the semi-algebraic systems C_1, \dots, C_e is consistent, that is, admits at least one real solution.
- (ii) [code soundness] For all real values $h_1, \dots, h_t, x_1, \dots, x_s, y_1, \dots, y_u, z_1, \dots, z_v$ of $P_1, \dots, P_t, R_1, \dots, R_s, D_1, \dots, D_u, E_1, \dots, E_v$ respectively, for all $i \in \{1, \dots, e\}$ such that $(h_1, \dots, h_t, x_1, \dots, x_s, y_1, \dots, y_u, z_1, \dots, z_v)$ is a solution of C_i , then the code fragment S_i produces the same output as S on any data that makes S execute correctly.
- (iii) [coverage] For all real values $y_1, \dots, y_u, z_1, \dots, z_v$ of $D_1, \dots, D_u, E_1, \dots, E_v$, respectively, there exist $i \in \{1, \dots, e\}$ and real values $h_1, \dots, h_t, x_1, \dots, x_s$ of $P_1, \dots, P_t, R_1, \dots, R_s$, such that $(h_1, \dots, h_t, x_1, \dots, x_s, y_1, \dots, y_u, z_1, \dots, z_v)$ is a solution of C_i and S_i produces the same output as S on any data that makes S execute correctly.
- (iv) [optimality] For every $i \in \{1, \dots, s\}$ (resp. $\{1, \dots, t\}$), there exists $\ell \in \{1, \dots, e\}$ such that for all $O \in \sigma(r_i)$ (resp. $\sigma(p_i)$) we have $f_i(O(S_\ell)) = f_i(S_\ell)$ (resp. $g_i(O(S_\ell)) = g_i(S_\ell)$).

To summarize Definition 2 in non technical terms:

- Condition (i) states that each system of constraints is meaningful.
- Condition (ii) states that as long as the machine, program and data parameters satisfy C_i , the code fragment \mathcal{S}_i produces the same output as \mathcal{S} on whichever data that makes \mathcal{S} execute correctly.
- Condition (iii) states that as long as \mathcal{S} executes correctly on a given set of parameters and data, there exists a code fragment \mathcal{S}_i , for suitable values of the machine parameters, such that \mathcal{S}_i produces the same output as \mathcal{S} on that set of parameters and data.
- Condition (iv) states that for each resource counter r_i (performance counter p_i), there exists at least one code fragment \mathcal{S}_ℓ for which this counter is optimal in the sense that it cannot be further optimized by the optimization strategies from $\sigma(r_i)$ (resp. $\sigma(p_i)$).

7.1.6 Data-structures

The algorithm presented in Section 7.1.7 computes a comprehensive optimization of \mathcal{S} w.r.t. the evaluation functions $f_1, \dots, f_s, g_1, \dots, g_t$ and optimization strategies O_1, \dots, O_w .

Hereafter, we define the main data-structure used during the course of the algorithm. We associate \mathcal{S} with what we call a *quintuple*, denoted by $Q(\mathcal{S})$ and defined as follows:

$$Q(\mathcal{S}) = (G_C(\mathcal{S}), \lambda(\mathcal{S}), \omega(\mathcal{S}), \gamma(\mathcal{S}), C(\mathcal{S}))$$

where

1. $\lambda(\mathcal{S})$ is the sequence of the optimization procedures among O_1, \dots, O_w that have already been applied to the IR of \mathcal{S} ; hence, $G_C(\mathcal{S})$ together with $\lambda(\mathcal{S})$ defines $G_L(\mathcal{S})$; initially, $\lambda(\mathcal{S})$ is empty,
2. $\omega(\mathcal{S})$ is the sequence of the optimization procedures among O_1, \dots, O_w that have not been applied so far to either $G_C(\mathcal{S})$ or $G_L(\mathcal{S})$; initially, $\omega(\mathcal{S})$ is O_1, \dots, O_w ,
3. $\gamma(\mathcal{S})$ is the sequence of resource and performance counters that remain to be evaluated on \mathcal{S} ; initially, $\gamma(\mathcal{S})$ is $r_1, \dots, r_s, p_1, \dots, p_t$,
4. $C(\mathcal{S})$ is the sequence of the constraints (polynomial equations and inequalities) on $P_1, \dots, P_t, R_1, \dots, R_s, D_1, \dots, D_u, E_1, \dots, E_v$ that have been computed so far; initially, $C(\mathcal{S})$ is $1 \geq P_1 \geq 0, \dots, 1 \geq P_t \geq 0, R_1 \geq 0, \dots, R_s \geq 0, D_1 \geq 0, \dots, D_u \geq 0, E_1 \geq 0, \dots, E_v \geq 0$.

We say that the quintuple $Q(\mathcal{S})$ is *processed* whenever $\gamma(\mathcal{S})$ is empty; otherwise, we say that the quintuple $Q(\mathcal{S})$ is *in-process*.

Remark 2 For the above $Q(\mathcal{S})$, each of the sequences $\lambda(\mathcal{S})$, $\omega(\mathcal{S})$, $\gamma(\mathcal{S})$ and $C(\mathcal{S})$ is implemented as a stack in Algorithms 5 and 6. Hence, we need to specify how operations on a sequence are performed on the corresponding stack. Let s_1, s_2, \dots, s_N is a sequence.

- Popping one element out of this sequence returns s_1 and leaves that sequence with s_2, \dots, s_N ,
- Pushing an element t_1 on s_1, s_2, \dots, s_N will update that sequence to $t_1, s_1, s_2, \dots, s_N$.
- Pushing a sequence of elements t_1, t_2, \dots, t_M on s_1, s_2, \dots, s_N will update that sequence to $t_M, \dots, t_2, t_1, s_1, s_2, \dots, s_N$.

7.1.7 The algorithm

Algorithm 5 is the top-level procedure. If its input is a processed quintuple $Q(\mathcal{S})$, then it returns the pair $(G_C(\mathcal{S}), \lambda(\mathcal{S}))$ (such that, after optimizing \mathcal{S} with the optimization strategies in $\lambda(\mathcal{S})$,

one can generate the IR of the optimized S) together with the system of constraints $C(S)$. Otherwise, Algorithm 5 is called recursively on each quintuple returned by $\text{Optimize}(Q(S))$. The pseudo-code of the Optimize routine is given by Algorithm 6.

Algorithm 5: ComprehensiveOptimization ($Q(S)$)

Input: The quintuple $Q(S)$

Output: A *comprehensive optimization* of S w.r.t. the resource evaluation functions f_1, \dots, f_s , the performance evaluation functions g_1, \dots, g_t and the optimization strategies O_1, \dots, O_w

```

1 if  $\gamma(S)$  is empty then
2   return  $((G_C(S), \lambda(S)), C(S))$ ;
3 The output stack is initially empty;
4 for each  $Q(S') \in \text{Optimize}(Q(S))$  do
5   Push ComprehensiveOptimization( $Q(S')$ ) on the output stack;
6 return the output stack;
```

Remark 3 We make a few observations about Algorithm 6.

- (R1) Observe that at Line (5), a deep copy of the input $Q(S')$ is made, and this copy is called $Q(S'')$. This duplication allows the computations to *fork*. Note that at Line (6), $Q(S')$ is modified.
- (R2) In this forking process, we call $Q(S')$ the *accept* branch and $Q(S'')$ the *refuse* branch. In the former case, the relation $0 \leq v_i \leq R_i$ holds thus implying that enough R_i -resources are available for executing the code fragment S' . In the latter case, the relation $R_i < v_i$ holds thus implying that *not* enough R_i -resources are available for executing the code fragment S'' .
- (R3) Observe that v_i is either a numerical value, a polynomial in $\mathbb{Q}[D_1, \dots, D_u, E_1, \dots, E_v]$ or a rational function where its numerator and denominator are in $\mathbb{Q}[D_1, \dots, D_u, E_1, \dots, E_v, R_1, \dots, R_s]$.
- (R4) At Lines (18-20), a similar forking process occurs. Here again, we call $Q(S')$ the *accept* branch and $Q(S'')$ the *refuse* branch. In the former case, the relation $0 \leq v_i \leq P_i$ implies that the P_i -performance counter may have reached its maximum ratio; hence, no optimization strategies are applied to improve this counter. In the latter case, the relation $P_i < v_i \leq 1$ holds thus implying that the P_i -performance counter has not reached its maximum value; hence, optimization strategies are applied to improve this counter if such optimization strategies are available. Observe that if this optimization strategy does make the estimated value of P_i larger then an algebraic contradiction would happen and the branch will be discarded.
- (R5) Line (30) in Algorithm 6 requires non-trivial computations with polynomial equations and inequalities. The necessary algorithms can be found in [22] and are implemented in the RegularChains library of MAPLE.

Remark 4 We make a few remarks about the handling of algebraic computation during the execution of Algorithm 6:

- (R6) Each system of algebraic constraints C is updated by adding a polynomial inequality to it at either Lines (6), (7), (19) or (20). This incremental process can be performed by the RealTriangularize algorithm [22] and implemented in the RegularChains library.

Algorithm 6: Optimize**Input:** A quintuple $Q(S')$ **Output:** A stack of quintuples

```

1 Initialize an empty stack, called result;
2 Take out from  $\gamma(S')$  the next resource or performance counter to be evaluated, say  $c$ ;
3 Evaluate  $c$  on  $S'$  (using the appropriate functions among  $f_1, \dots, f_s, g_1, \dots, g_t$ ) thus obtaining a
  value  $v_i$ , which can be either a numerical value, a polynomial in  $\mathbb{Q}[D_1, \dots, D_u, E_1, \dots, E_v]$  or a
  rational function where its numerator and denominator are in  $\mathbb{Q}[D_1, \dots, D_u, E_1, \dots, E_v,
  R_1, \dots, R_s]$ ;
4 if  $c$  is a resource counter  $r_i$  then
5   Make a deep copy  $Q(S'')$  of  $Q(S')$ , since we are going to split the computation into two
   branches:  $R_i < v_i$  and  $0 \leq v_i \leq R_i$ ;
6   Add the constraint  $0 \leq v_i \leq R_i$  to  $C(S')$  and push  $Q(S')$  onto result;
7   Add the constraint  $R_i < v_i$  to  $C(S'')$  and search  $\omega(S'')$  for an optimization strategy of  $\sigma(r_i)$ ;
8   if no such optimization strategy exists then
9     return result;
10  else
11    Apply such an optimization strategy to  $Q(S'')$  yielding  $Q(S''')$ ;
12    Remove this optimization strategy from  $\omega(S''')$ ;
13    if this optimization strategy is applied to the IR of  $S''$  then
14      Add it to  $\lambda(S''')$ ;
15    Push  $r_1, \dots, r_{i-1}, r_i$  onto  $\gamma(S''')$ ;
16    Make a recursive call to Optimize on  $Q(S''')$  and push the returned quintuples onto
    result;
17 if  $c$  is a performance counter  $p_i$  then
18   Make a deep copy  $Q(S'')$  of  $Q(S')$ , since we are going to split the computation into two
   branches:  $0 \leq v_i \leq P_i$  and  $P_i < v_i \leq 1$ ;
19   Add the constraint  $0 \leq v_i \leq P_i$  to  $C(S')$  and push  $Q(S')$  onto result;
20   Add the constraint  $P_i < v_i \leq 1$  to  $C(S'')$  and search  $\omega(S'')$  for an optimization strategy of
    $\sigma(p_i)$ ;
21   if no such optimization strategy exists then
22     return result;
23   else
24     Apply such an optimization strategy to  $Q(S'')$  yielding  $Q(S''')$ ;
25     Remove this optimization strategy from  $\omega(S''')$ ;
26     if this optimization strategy is applied to the IR of  $S''$  then
27       Add it to  $\lambda(S''')$ ;
28     Push  $r_1, \dots, r_s, p_i$  onto  $\gamma(S''')$ ;
29     Make a recursive call to Optimize on  $Q(S''')$  and push the returned quintuples onto
    result;
30 Remove from result any quintuple with an inconsistent system of constraints;
31 return result;

```

- (R7) Each of these inequalities can be either strict (using $>$) or large (using \leq); the left-hand side is a polynomial of either $\mathbb{Q}[D_1, \dots, D_u, E_1, \dots, E_v]$ or $\mathbb{Q}[D_1, \dots, D_u, E_1, \dots, E_v, R_1, \dots, R_s]$, and the right-hand side is either one of the variables R_1, \dots, R_s or a polynomial of $\mathbb{Q}[D_1, \dots, D_u, E_1, \dots, E_v, R_1, \dots, R_s]$ times one of the variables P_1, \dots, P_t .
- (R8) Because of the recursive calls at Lines (16) and (29) several inequalities involving the same variable among $R_1, \dots, R_s, P_1, \dots, P_t$ may be added to a given system C . As a result, C may become inconsistent. For instance if $10 \leq R_1$ and $R_1 < 10$ are both added to the same system C . Note that inconsistency is automatically detected by the `RealTriangularize` algorithm.
- (R9) When using `RealTriangularize`, variables should be ordered. We choose a variable ordering such that
- any of P_1, \dots, P_t is greater than any of the other variables,
 - any of R_1, \dots, R_s is greater than any of $D_1, \dots, D_u, E_1, \dots, E_v$.
- (R10) Then, the `RealTriangularize` represents the solution of C as the union of the solution sets of finitely many regular semi-algebraic systems. Each such regular semi-algebraic system Υ consists of
- polynomial constraints involving $D_1, \dots, D_u, E_1, \dots, E_v$ only,
 - polynomial constraints involving $D_1, \dots, D_u, E_1, \dots, E_v, R_1, \dots, R_s$ only and of positive degree in at least one of R_1, \dots, R_s ,
 - constraints that are linear in P_1, \dots, P_t and polynomial in $D_1, \dots, D_u, E_1, \dots, E_v, R_1, \dots, R_s$.

Let us denote by $\Upsilon_{D,E}$, $\Upsilon_{D,E,R}$ and $\Upsilon_{D,E,R,P}$ these three sets of polynomial constraints, respectively. It follows from the properties of regular semi-algebraic systems that

- (a) $\Upsilon_{D,E}$ is consistent,
- (b) for all real values $y_1, \dots, y_u, z_1, \dots, z_v$ of $D_1, \dots, D_u, E_1, \dots, E_v$ such that $(y_1, \dots, y_u, z_1, \dots, z_v)$ solves $\Upsilon_{D,E}$, there exists real values x_1, \dots, x_s of R_1, \dots, R_s such that $(y_1, \dots, y_u, z_1, \dots, z_v, x_1, \dots, x_s)$ solves $\Upsilon_{D,E,R}$ and real values h_1, \dots, h_t of P_1, \dots, P_t such that $(y_1, \dots, y_u, z_1, \dots, z_v, x_1, \dots, x_s, h_1, \dots, h_t)$ solves $\Upsilon_{D,E,R,P}$.

Notation 2 We associate the execution of Algorithm 5, applied to $Q(S)$, with a tree denoted by $\mathcal{T}(Q(S))$ and where both nodes and edges of $\mathcal{T}(Q(S))$ are labelled. We use the same notations as in Algorithm 6. We define $\mathcal{T}(Q(S))$ recursively as follows:

- (T1) We label the root of $\mathcal{T}(Q(S))$ with $Q(S)$.
- (T2) If $\gamma(S)$ is empty, then $\mathcal{T}(Q(S))$ has no children; otherwise, two cases arise:
 - (T2.1) If no optimization strategy is to be applied for optimizing the counter c , then $\mathcal{T}(Q(S))$ has a single subtree, which is that associated with `Optimize($Q(S')$)` where $Q(S')$ is obtained from $Q(S)$ by augmenting $C(S)$ either with $0 \leq v_i \leq R_i$ if c is a resource counter or with $0 \leq v_i \leq P_i$ otherwise.
 - (T2.2) If an optimization strategy is applied, then $\mathcal{T}(Q(S))$ has two subtrees:
 - (T2.2.1) The first one is the tree associated with `Optimize($Q(S')$)` (where $Q(S')$ is defined as above) and is connected to its parent node by the *accept* edge, labelled with either $0 \leq v_i \leq R_i$ or $0 \leq v_i \leq P_i$; see Figure 7.5.
 - (T2.2.2) The second one is the tree associated with `Optimize($Q(S''')$)` (where $Q(S''')$ is obtained by applying the optimization strategy to the deep copy of the input

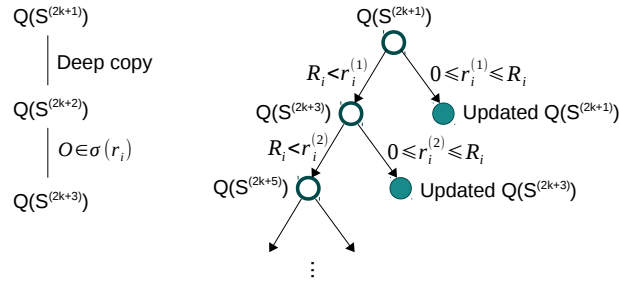
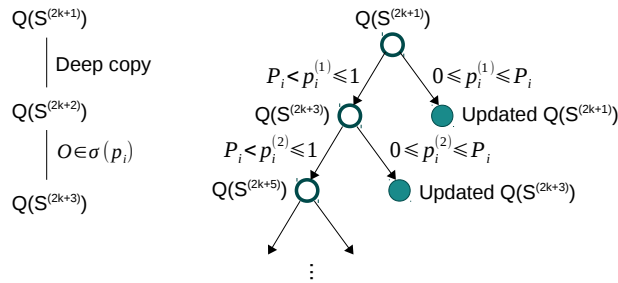
(a) The decision subtree for resource counter r_i (b) The decision subtree for performance counter p_i

Figure 7.5: The decision subtree for resource or performance counters

quintuple $Q(S)$ and is connected to its parent node by the *refuse* edge, labelled with either $R_i < v_i$ or $P_i < v_i \leq 1$; see Figure 7.5.

Observe that every node of $\mathcal{T}(Q(S))$ is labelled with a quintuple and every edge is labelled with an inequality constraint.

Remark 5 Figure 7.5 illustrates how Algorithm 6, applied to $Q(S')$, generates the associated tree $\mathcal{T}(Q(S'))$. The cases for a *resource counter* and a *performance counter* are distinguished in the sub-figures (a) and (b), respectively. Observe that, in both cases, the accept edges go *south-east*, while the refuse edges go *south-west*.

Lemma 7.1.1 *The height of the tree $\mathcal{T}(Q(S))$ is at most $w(s + t)$. Therefore, Algorithm 5 terminates.*

Proof Consider a path Γ from the root of $\mathcal{T}(Q(S))$ to any node N of $\mathcal{T}(Q(S))$. Observe that Γ counts at most w refuse edges. Indeed, following a refuse edge decreases by one the number of optimization strategies to be used. Observe also that the length of every sequence of consecutive accept edges is at most $s + t$. Indeed, following an accept edge decreases by one the number of resource and performance counters to be evaluated. Therefore, the number of edges in Γ is at most $w(s + t)$.

Lemma 7.1.2 *Let $U := \{U_1, \dots, U_z\}$ be a subset of $\{O_1, \dots, O_w\}$. There exists a path from the root of $\mathcal{T}(Q(S))$ to a leaf of $\mathcal{T}(Q(S))$ along which the optimization strategies being applied are exactly those of U .*

Proof Let us start at the root of $\mathcal{T}(Q(S))$ and apply the following procedure:

1. follow the refuse edge if it uses an optimization strategy from $\{U_1, \dots, U_z\}$,
2. follow the accept edge, otherwise.

This creates a path from the root of $\mathcal{T}(Q(S))$ to a leaf with the desired property.

Definition 3 Let $i \in \{1, \dots, s\}$ (resp. $\{1, \dots, t\}$). Let N be a node of $\mathcal{T}(Q(S))$ and $Q(S_N)$ be the quintuple labelling this node. We say that r_i (resp. p_i) is optimal at N w.r.t. the evaluation function f_i (resp. g_i) and the subset $\sigma(r_i)$ (resp. $\sigma(p_i)$) of the optimization strategies O_1, \dots, O_w , whenever for all $O \in \sigma(r_i)$ (resp. $\sigma(p_i)$) we have $f_i(O(S_N)) = f_i(S_N)$ (resp. $g_i(O(S_N)) = g_i(S_N)$).

Lemma 7.1.3 Let $i \in \{1, \dots, s\}$ (resp. $\{1, \dots, t\}$). There exists at least one leaf L of $\mathcal{T}(Q(S))$ such that r_i (resp. p_i) is optimal at L w.r.t. the evaluation function f_i (resp. g_i) and the subset $\sigma(r_i)$ (resp. $\sigma(p_i)$) of the optimization strategies O_1, \dots, O_w .

Proof Apply Lemma 7.1.2 with $U = \sigma(r_i)$ (resp. $U = \sigma(p_i)$).

Lemma 7.1.4 Algorithm 5 satisfies its output specifications.

Proof From Lemma 7.1.1, we know that Algorithm 5 terminates. So let $(C_1, S_1), \dots, (C_e, S_e)$ be its output. We shall prove $(C_1, S_1), \dots, (C_e, S_e)$ satisfies the conditions (i) to (iv) of Definition 2. Condition (i) is satisfied by the properties of the `RealTriangularize` algorithm, see Section 2.6. Condition (ii) follows clearly from the assumption that the code transformations performed by O_1, \dots, O_w preserve semantics. Observe that each time a polynomial inequality is added to a system of constraints, the negation of this inequality is also to the same system in another branch of the computations. By using a simple induction on $s + t$, we deduce that Condition (iii) is satisfied. Finally, we prove Condition (iv) by using Lemma 7.1.3.

7.2 Comprehensive translation of an annotated C program into CUDA kernels

Given a high-level model for accelerator programming (like OpenCL [111], OpenMP [41, 13, 9], OpenACC [113, 56] or MetaFork [29]), we consider the problem of translating a program written for such a high-level model into a programming model for GPGPU devices, such as CUDA [95, 73]. We assume that the numerical values of some, or all, of the hardware characteristics of the targeted GPGPU device are unknown. Hence, these quantities are treated as symbols. Similarly, we would like that some, or all, of the program parameters remain symbols in the generated code.

In our implementation, we focus on one high-level model for accelerator programming, namely MetaFork. However, we believe that an adaptation to another high-level model for accelerator programming would not be difficult. One supporting reason for that claim is the fact that automatic code translation between the MetaFork and OpenMP languages can already be done within the MetaFork compilation framework, see [29].

The hardware characteristics of the GPGPU device can be the maximum number of registers to be allocated per thread in a thread-block and the maximum number of shared memory

words to be allocated per thread in a thread-block. Similarly, the program parameters can be the number of threads per thread-block and the granularity of threads. For the generated code to be valid, hardware characteristics and program parameters need to satisfy constraints in the form of polynomial equations and inequalities. Moreover, applying code transformation (like optimization techniques) requires a case distinction based on the values of those symbols, as we saw with the example in the introduction.

In Section 7.2.1, we specify the required properties of the input code fragment \mathcal{S} from the given MetaFork program, so that the comprehensive optimization algorithm, demonstrated in Section 7.1, can handle this MetaFork program. Section 7.2.2 discusses the procedure of *comprehensive translation* of the MetaFork program into parametric CUDA kernels, which yields the definition of *comprehensive parametric CUDA kernels*.

7.2.1 Input MetaFork code fragment

Consider a `meta_schedule` statement \mathcal{M} and its surrounding MetaFork program \mathcal{P} . In this process of code analysis and transformation, we focus on the `meta_schedule` statement \mathcal{M} and assume that the rest of the program \mathcal{P} is serial C code. Hence, our examples, like the matrix vector multiplication and matrix addition examples, consist simply of a `meta_schedule` statement \mathcal{M} together with a few (possibly none) statements located before \mathcal{M} and initializing variables used in \mathcal{M} .

Consider the `meta_schedule` statement \mathcal{M} , that is, a statement of the form

`meta_schedule A`

where A is a compound statement of the form $\{A_0 A_1 \cdots A_\ell\}$ and each of A_0, A_1, \dots, A_ℓ is a `for-loop` nest, such that:

1. each `for-loop` nest contains 2 or 4 `meta_for` loops; hence, it can be executed in a parallel fashion,
2. the body of the innermost loop can be any valid sequence of C statements; in particular, such a statement can be a `for-loop`, and

In practice, a parameter (in the sense of Definition 1) of the `meta_schedule` statement \mathcal{M} is either a *data parameter* (that is, related to data being processed, like a number of rows or columns in a matrix) or a *program parameter* (that is, related to the division of the work among the threads executing the parallel `for-loops`). In Example 4, the variable N is a data parameter, whereas the variables \dim and B are the program parameters.

Moreover, for the sake of clarity, we shall assume that the `meta_schedule` statement \mathcal{M} counts a single `meta_for` loop nest A . Extending the present section to the case where \mathcal{M} counts several `meta_for` loop nests can be done by existing techniques as we briefly explain now. Indeed, each `meta_for` loop nest can be handled separately. Then, “merging” the corresponding results can be done by techniques from symbolic computation, see [26, 27]. Therefore, we consider the serial elision (as defined in [29]) of A in \mathcal{M} as the code fragment \mathcal{S} . Turning our attention back to Example 4, Figure 7.6 shows the serial elision of the MetaFork program on the right-hand portion of Figure 7.4.

```

int dim = N / B, v, u;
// v is corresponding to the thread-block index
// u is corresponding to the thread index in a thread-block

// The following code is the serial elision
for (int i = 0; i < dim; ++i)
    for (int j = 0; j < B; ++j) {
        int p = v * B + u;
        int q = i * B + j;
        c[p] += a[p][q] * b[q];
    }

```

Figure 7.6: The serial elision of the MetaFork program for matrix vector multiplication

7.2.2 Comprehensive translation into parametric CUDA kernels

Now, applying the comprehensive optimization algorithm (described in Section 7.1) on the serial elision S of the `meta_schedule` statement \mathcal{M} (with prescribed resource evaluation functions, performance evaluation functions and optimization strategies), we obtain a sequence of processed quintuples of `meta_schedule` statements $Q_1(\mathcal{M}), Q_2(\mathcal{M}), \dots, Q_\ell(\mathcal{M})$, which forms a comprehensive optimization in the sense of Definition 2.

If, as mentioned in the introduction of this chapter, PTX is used as intermediate representation (IR) then, for each $i = 1, \dots, \ell$, under the constraints defined by the polynomial system associated with $Q_i(\mathcal{M})$, the IR code associated with $Q_i(\mathcal{M})$ is the translation in assembly language of a CUDA counterpart of \mathcal{M} . In our implementation, we also translate to CUDA source code the MetaFork code in each $Q_i(\mathcal{M})$, since this is easier to read for a human being.

Therefore, in broad terms, a *comprehensive translation* of the `meta_schedule` statement \mathcal{M} into parametric CUDA kernels is a decision tree, where each edge holds a Boolean expression (given by a polynomial constraint) and each leaf is either a CUDA program in PTX form, or the symbol \emptyset , such that for each leaf K , with $K \neq \emptyset$, we have:

1. K works correctly under the conjunction of the Boolean expressions located between the root node and the leaf, and
2. K is semantically equivalent to \mathcal{P} .

The symbol \emptyset is used to denote a situation (in fact, value ranges for the machine and program parameters) where no CUDA program equivalent to \mathcal{P} is provided.

7.3 Implementation details

In this section, we present a preliminary implementation of the comprehensive optimization algorithm demonstrated in Section 7.1. This implementation takes a given MetaFork program as input and is dedicated to the optimization of `meta_schedule` statements in view of generating parametric CUDA kernels.

For the algorithm stated in Section 7.1.7 to satisfy its specifications, one should use the PTX language for the IR. However, for simplicity, in our *proof-of-concept* implementation

here, we use the IR of the LLVM compiler infrastructure [75], since the MetaFork compilation framework is based on CLANG [30].

Two hardware resource counters are considered: register usage per thread and local/shared memory allocated per thread-block. No performance counters are specified; however, by design, the algorithm tries to minimize the usage of hardware resources. Four optimization strategies are used: (i) reducing register pressure, (ii) controlling thread granularity, (iii) common sub-expression elimination (CSE), and (iv) caching³ data in local/shared memory. Details are given hereafter.

Figure 7.7 gives an overview of the software tools that are used for our implementation. Appendix E shows the implemented algorithms with these two resource counters and these three optimization strategies.

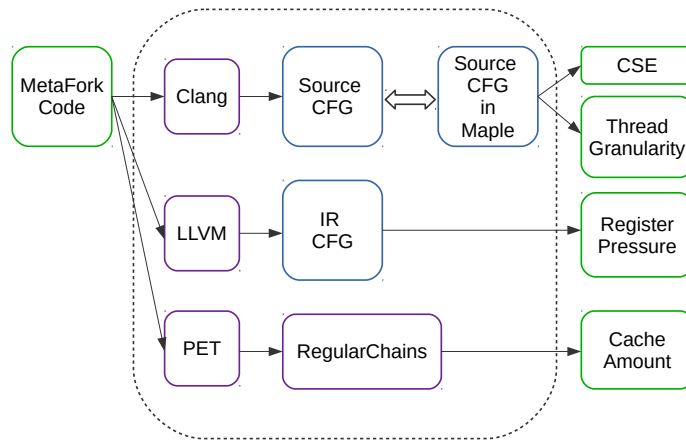


Figure 7.7: The software tools involved for the implementation

Conversion between source code and CFG. CLANG [30] is used as the front-end for parsing the MetaFork source code and generating the source CFG. This latter is converted into a MAPLE DAG in order to take advantage of MAPLE’s capabilities for algebraic computation. Conversely, given a MAPLE version of a CFG, we can generate the corresponding MetaFork code by traversing this CFG.

Register pressure. Given the MetaFork source code, we use LLVM to generate the low-level machine instructions in the intermediate representation (IR), which are in a *static single assignment* (SSA) form [40]. A benefit of using the SSA form is that one can calculate the *liveness sets*⁴ [123] without data flow analysis [123]. Once the lifetime information is computed, we use the classical linear scan algorithm [97] to estimate the register usage.

Thread granularity. A common method to improve arithmetic intensity and instruction level parallelism (ILP) is through controlling the granularity of threads [114], that is, each thread computing more than one of the output results. One can achieve this goal by adding a serial for loop within a thread. However, this method may increase the register usage as well as the amount of required shared memory (see the matrix multiplication example in Section 6.1).

³In the MetaFork language, the keyword `cache` is used to indicate that every thread accessing a specified array must copy in local/shared memory the data it accesses in a.

⁴https://en.wikipedia.org/wiki/Live_variable_analysis

In the case that adding the granularity loop causes the needed resources to exceed the hardware limits, our algorithm applies an optimized strategy, “Granularity set to 1,” to remove that serial loop from the generated kernel code. We implement this strategy during the translation phase from the CFG to the MetaFork code by not generating this loop.

Common sub-expression elimination (CSE). For each basic block in the CFG, built from the MetaFork source code, we consider the basic blocks with more than one statement. Then, we use the `codegen[optimize]` package of MAPLE, such that CSE is applied to those statements and a sequence of new statements is generated. Finally, we update each basic block with those new statements. Moreover, the optimization technique has two levels: one using MAPLE’s default CSE algorithm and the other using the `try-harder` option of `codegen[optimize]`.

Cache amount. We take advantage of the PET (polyhedral extraction tool) [116] to collect information related to the index expression of an array: occurring program parameters (defined as in Section 7.2), loop iteration counters and inequalities (that give the lower and upper bounds of those loop iteration counters). We now illustrate with an example for computing the amount of words that a thread-block requires. Consider the MetaFork program with the granularity for reversing a 1D array as shown on the left-hand portion of Figure 7.8. Note that w.r.t the MetaFork code, iteration counters i , j and k of `for`- (and `meta_for`-) loops are counted as neither program nor data parameters; thus, we shall consider them as bounded variables. However, in order to calculate the amount of words required per thread-block in the RegularChains library of MAPLE, we treat the iteration counter i , which indicates the thread-block index of the CUDA kernel, as a program parameter. The function call `ValueRangeWithConstraintsAndParameters` to the RegularChains library, as shown on the right-hand portion of Figure 7.8, is used to calculate the required words per thread-block for vector c . As a result, a range $[1, s*B+1]$ is returned, such that we can determine that vector c accesses $s*B$ words per thread-block.

<pre>// Data parameters: a, c, N // Program parameters: B, s int dim = N / (B * s); meta.schedule { meta_for (int i = 0; i < dim; i++) meta_for (int j = 0; j < B; j++) for (int k = 0; k < s; ++k) { int x = v * B * s + k * B + u; int y = N - 1 - x c[y] = a[x]; } }</pre>	<pre>lowerBound := []; upperBound := []; fixedVars := []; boundedVars := [j, k]; params := [i, s, B, N]; x := (((i)*(B))*(s))+((k)*(B)))+(j); y := ((N)-(1))-(x); S := [j >= 0, j <= -1 + B, k >= 0, k <= -1 + s]; i0 := y; bounds := ValueRangeWithConstraintsAndParameters (i0, S, lowerBound, upperBound, fixedVars, boundedVars, params);</pre>
--	---

(a) MetaFork program with the granularity (b) function call to RegularChains library

Figure 7.8: Computing the amount of words required per thread-block for reversing a 1D array

7.4 Experimentation

We present experimental results for the implementation described in Section 7.3. We consider six simple test cases: *array reversal*, *matrix vector multiplication*, *1D Jacobi*, *matrix addition*, *matrix transpose* and *matrix matrix multiplication*. Recall that we consider two machine

parameters: the amount Z_B of shared memory per streaming multiprocessor (SM) and the maximum number R_B of registers per thread in a thread-block.

Three scenarios of optimized MetaFork programs based on different systems of constraints are generated by our implementation of the comprehensive optimization algorithm. The first case of optimized MetaFork programs uses the shared memory and a granularity parameter s . The second case uses the shared memory but sets the granularity parameter s to 1. The third case removes the `cache`⁵ keyword and sets s to 1. In this latter case, the amount of words read and written per thread-block is more than the maximum amount Z_B of shared memory per SM. However, the `cache` keyword is not implemented in the MetaFork compilation framework yet, so that we manually process editing for this keyword. For each of these optimization strategies, we use a shortened code shown in Table 7.1.

Table 7.1: Optimization strategies with their codes

Strategy name	Its code	Strategy name	Its code
“Accept register pressure”	(1)	“CSE applied”	(2)
“No granularity reduction”	(3a)	“Granularity set to 1”	(3b)
“Accept caching”	(4a)	“Refuse caching”	(4b)

Array reversal. Our comprehensive optimization algorithm applies to the source code the following optimization strategy codes (1) (4a) (3a) (2) (2). For the first case, it generates the optimized MetaFork code shown in Figure 7.9.

Constraints:

$$\begin{cases} 2sB \leq Z_B \\ 4 \leq R_B \end{cases}$$

strategies (1) (4a) (3a) (2) (2) applied

```

int N, s, B, dim = N/(s*B);
int a[N], c[N];
meta_schedule cache(a, c) {
  meta_for (int i = 0; i < dim; i++)
    meta_for (int j = 0; j < B; j++)
      for (int k = 0; k < s; ++k) {
        int x = (i*s+k)*B+j;
        int y = N-1-x;
        c[y] = a[x];
      }
}

```

Figure 7.9: The first case of the optimized MetaFork code for array reversal

Applying optimization strategy codes in a sequence either (1) (3b) (4a) (3a) (2) (2) or (2) (2) (3b) (1) (4a) (3a), the second case generates the optimized MetaFork code shown in Figure 7.10.

Applying optimization strategy codes in a sequence either (1) (3b) (2) (2) (4b) or (2) (2) (3b) (1) (4b), the third case generates the optimized MetaFork code shown in Figure 7.11.

Matrix vector multiplication. Applying optimization strategy codes in a sequence either (1) (4a) (3a) (2) (2) or (2) (1) (4a) (3a) (2), the first case generates the optimized MetaFork code shown in Figure 7.12.

⁵ In the MetaFork language, the intention of using the keyword `cache` is to indicate that every thread accessing a specified array `a` must copy in local/shared memory the data it accesses in `a`.

<p>Constraints:</p> $\begin{cases} 2B \leq Z_B < 2sB \\ 3 \leq R_B \end{cases}$ <p>strategies (1) (3b) (4a) (3a) (2) (2) applied</p> <p style="text-align: center;">or</p> $\begin{cases} 2B \leq Z_B < 2sB \\ 3 \leq R_B < 4 \end{cases}$ <p>strategies (2) (2) (3b) (1) (4a) (3a) applied</p>	<pre> int N, s = 1, B, dim = N/(s*B); int a[N], c[N]; meta_schedule cache(a, c) { meta_for (int i = 0; i < dim; i++) meta_for (int j = 0; j < B; j++) { int x = i*B+j; int y = N-1-x; c[y] = a[x]; } } </pre>
---	---

Figure 7.10: The second case of the optimized MetaFork code for array reversal

<p>Constraints:</p> $\begin{cases} Z_B < 2B \\ 3 \leq R_B \end{cases}$ <p>strategies (1) (3b) (2) (2) (4b) applied</p> <p style="text-align: center;">or</p> $\begin{cases} Z_B < 2B \\ 3 \leq R_B < 4 \end{cases}$ <p>strategies (2) (2) (3b) (1) (4b) applied</p>	<pre> int N, s = 1, B, dim = N/(s*B); int a[N], c[N]; meta_schedule { meta_for (int i = 0; i < dim; i++) meta_for (int j = 0; j < B; j++) { int x = i*B+j; int y = N-1-x; c[y] = a[x]; } } </pre>
---	---

Figure 7.11: The third case of the optimized MetaFork code for array reversal

<p>Constraints:</p> $\begin{cases} sB^2 + sB + B \leq Z_B \\ 8 \leq R_B \end{cases}$ <p>strategies (1) (4a) (3a) (2) (2) applied</p> <p style="text-align: center;">or</p> $\begin{cases} sB^2 + sB + B \leq Z_B \\ 8 \leq R_B < 9 \end{cases}$ <p>strategies (2) (1) (4a) (3a) (2) applied</p>	<pre> int N, s, B, dim0 = N/(s*B), dim1 = N/B; int a[N][N], b[N], c[N]; meta_schedule cache(a, b, c) { meta_for (int v = 0; v < dim0; v++) for (int i = 0; i < dim1; i++) meta_for (int u = 0; u < B; u++) for (int j = 0; j < B; ++j) for (int k = 0; k < s; ++k) { int p = (v*s+k)*B+u; int q = i*B+j; c[p] = a[p][q]*b[q]+c[p]; } } </pre>
---	--

Figure 7.12: The first case of the optimized MetaFork code for matrix vector multiplication

Applying the optimization strategy codes in a sequence either (1) (3b) (4a) (3b) (2) (2), (2) (1) (3b) (4a) (3a) (2) or (2) (2) (3b) (1) (4a) (3a), the second case generates the optimized MetaFork code shown in Figure 7.13.

Applying the optimization strategy codes in a sequence either (1) (3b) (2) (2) (4b), (2) (1) (3b) (2) (4b) or (2) (2) (3b) (1) (4b), the third case generates the optimized MetaFork code shown in Figure 7.14.

1D Jacobi. Given 1D Jacobi source code written in MetaFork, shown in Figure 7.15, the CSE strategy is applied successfully for all cases of optimized MetaFork programs. This example requires post-processing for calculating the total amount of required shared memory

Constraints:

$\begin{cases} B^2 + 2B \leq Z_B < sB^2 + sB + B \\ 7 \leq R_B \end{cases}$ <p>strategies (1) (3b) (4a) (3a) (2) (2) applied or</p> $\begin{cases} B^2 + 2B \leq Z_B < sB^2 + sB + B \\ 7 \leq R_B < 9 \end{cases}$ <p>strategies (2) (1) (3b) (4a) (3a) (2) applied or</p> $\begin{cases} B^2 + 2B \leq Z_B < sB^2 + sB + B \\ 7 \leq R_B < 8 \end{cases}$ <p>strategies (2) (2) (3b) (1) (4a) (3a) applied</p>	<pre> int N, s = 1, B, dim0 = N/(s*B), dim1 = N/B; int a[N][N], b[N], c[N]; meta_schedule cache(a, b, c) { meta_for (int v = 0; v < dim0; v++) for (int i = 0; i < dim1; i++) meta_for (int u = 0; u < B; u++) for (int j = 0; j < B; ++j) { int p = v*B+u; int q = i*B+j; c[p] = a[p][q]*b[q]+c[p]; } } </pre>
--	---

Figure 7.13: The second case of the optimized MetaFork code for matrix vector multiplication

Constraints:

$\begin{cases} Z_B < B^2 + 2B \\ 7 \leq R_B \end{cases}$ <p>strategies (1) (3b) (2) (2) (4b) applied or</p> $\begin{cases} Z_B < B^2 + 2B \\ 7 \leq R_B < 9 \end{cases}$ <p>strategies (2) (1) (3b) (2) (4b) applied or</p> $\begin{cases} Z_B < B^2 + 2B \\ 7 \leq R_B < 8 \end{cases}$ <p>strategies (2) (2) (3b) (1) (4b) applied</p>	<pre> int N, s = 1, B, dim0 = N/(s*B), dim1 = N/B; int a[N][N], b[N], c[N]; meta_schedule { meta_for (int v = 0; v < dim0; v++) for (int i = 0; i < dim1; i++) meta_for (int u = 0; u < B; u++) for (int j = 0; j < B; ++j) { int p = v*B+u; int q = i*B+j; c[p] = a[p][q]*b[q]+c[p]; } } </pre>
--	--

Figure 7.14: The third case of the optimized MetaFork code for matrix vector multiplication

per thread-block, due to the fact that array *a* has multiple accesses and that each access has a different index.

Applying the optimization strategy codes in a sequence (1) (4a) (3a) (2) (2), the first case generates the optimized MetaFork code shown in Figure 7.16.

Applying the optimization strategy codes in a sequence (1) (3b) (4a) (3a) (2) (2), the second case generates the optimized MetaFork code shown in Figure 7.17.

Applying the optimization strategy codes in a sequence (1) (3b) (2) (2) (4b), the third case generates the optimized MetaFork code shown in Figure 7.18.

Matrix addition. Due to the limitation in the `codegen[optimize]` package of MAPLE, the CSE optimizer could not handle a two-dimensional array on the left-hand side of assignments. Thus, we use a one-dimensional array to represent the output matrix. Applying the optimization strategy codes in a sequence (1) (4a) (3a) (2) (2), the first case generates the optimized MetaFork code shown in Figure 7.19.

```

int T, N, s, B, dim = (N-2)/(s*B);
int a[2*N];
for (int t = 0; t < T; ++t)
  meta_schedule {
    meta_for (int i = 0; i < dim; i++)
      meta_for (int j = 0; j < B; j++)
        for (int k = 0; k < s; ++k) {
          int p = i * s * B + k * B + j;
          int p1 = p + 1;
          int p2 = p + 2;
          int np = N + p;
          int np1 = N + p + 1;
          int np2 = N + p + 2;
          if (t % 2)
            a[p1] = (a[np] + a[np1] + a[np2]) / 3;
          else
            a[np1] = (a[p] + a[p1] + a[p2]) / 3;
        }
  }
}

```

Figure 7.15: The MetaFork source code for 1D Jacobi

Constraints:

$$\begin{cases} 2sB + 2 \leq Z_B \\ 9 \leq R_B \end{cases}$$

strategies (1) (4a) (3a) (2) (2) applied

```

int T, N, s, B, dim = (N-2)/(s*B);
int a[2*N];
for (int t = 0; t < T; ++t)
  meta_schedule cache(a) {
    meta_for (int i = 0; i < dim; i++)
      meta_for (int j = 0; j < B; j++)
        for (int k = 0; k < s; ++k) {
          int p = j+(i*s+k)*B;
          int t16 = p+1;
          int t15 = p+2;
          int p1 = t16;
          int p2 = t15;
          int np = N+p;
          int np1 = N+t16;
          int np2 = N+t15;
          if (t % 2)
            a[p1] = (a[np]+a[np1]+a[np2])/3;
          else
            a[np1] = (a[p]+a[p1]+a[p2])/3;
        }
  }
}

```

Figure 7.16: The first case of the optimized MetaFork code for 1D Jacobi

Applying the optimization strategy codes in a sequence either (1) (3b) (4a) (3a) (2) (2) or (2) (2) (3b) (1) (4a) (3a), the second case generates the optimized MetaFork code shown in Figure 7.20.

Applying the optimization strategy codes in a sequence either (1) (3b) (2) (2) (4b) or (2) (2) (3b) (1) (4b), the third case generates the optimized MetaFork code shown in Figure 7.21.

Matrix transpose. Applying the optimization strategy codes in a sequence (1) (4a) (3a)

Constraints:

$$\begin{cases} 2B + 2 \leq Z_B < 2sB + 2 \\ 9 \leq R_B \end{cases}$$

strategies (1) (3b) (4a) (3a) (2) (2) applied

```

int T, N, s = 1, B, dim = (N-2)/(s*B);
int a[2*N];
for (int t = 0; t < T; ++t)
  meta_schedule cache(a) {
    meta_for (int i = 0; i < dim; i++)
      meta_for (int j = 0; j < B; j++) {
        int p = i*B+j;
        int t20 = p+1;
        int t19 = p+2;
        int p1 = t20;
        int p2 = t19;
        int np = N+p;
        int np2 = N+t19;
        int np1 = N+t20;
        if (t % 2)
          a[p1] = (a[np]+a[np1]+a[np2])/3;
        else
          a[np1] = (a[p]+a[p1]+a[p2])/3;
      }
  }

```

Figure 7.17: The second case of the optimized MetaFork code for 1D Jacobi

Constraints:

$$\begin{cases} Z_B < 2B + 2 \\ 9 \leq R_B \end{cases}$$

strategies (1) (3b) (2) (2) (4b) applied

```

int T, N, s = 1, B, dim = (N-2)/(s*B);
int a[2*N];
for (int t = 0; t < T; ++t)
  meta_schedule {
    meta_for (int i = 0; i < dim; i++)
      meta_for (int j = 0; j < B; j++) {
        int p = j+i*B;
        int t16 = p+1;
        int t15 = p+2;
        int p1 = t16;
        int p2 = t15;
        int np = N+p;
        int np1 = N+t16;
        int np2 = N+t15;
        if (t % 2)
          a[p1] = (a[np]+a[np1]+a[np2])/3;
        else
          a[np1] = (a[p]+a[p1]+a[p2])/3;
      }
  }

```

Figure 7.18: The third case of the optimized MetaFork code for 1D Jacobi

(2) (2), the first case generates the optimized MetaFork code shown in Figure 7.22.

Applying the optimization strategy codes in a sequence either (1) (3b) (4a) (3a) (2) (2) or (2) (2) (3b) (1) (4a) (3a), the second case generates the optimized MetaFork code shown in Figure 7.23.

Applying the optimization strategy codes in a sequence either (1) (3b) (2) (2) (4b) or (2) (2) (3b) (1) (4b), the third case generates the optimized MetaFork code shown in Figure 7.24.

Matrix matrix multiplication. Applying the optimization strategy codes in a sequence (1)

<p>Constraints:</p> $\begin{cases} 3sB_0B_1 \leq Z_B \\ 7 \leq R_B \end{cases}$ <p>strategies (1) (4a) (3a) (2) (2) applied</p>	<pre> int N, B0, B1, s, dim0 = N/B0, dim1 = N/(B1*s); int a[N][N], b[N][N], c[N*N]; meta_schedule cache(a, b, c) { meta_for (int v0 = 0; v0 < dim0; v0++) meta_for (int v1 = 0; v1 < dim1; v1++) meta_for (int u0 = 0; u0 < B0; u0++) meta_for (int u1 = 0; u1 < B1; u1++) for (int k = 0; k < s; ++k) { int i = v0*B0+u0; int j = (v1*s+k)*B1+u1; c[i*N+j] = a[i][j] + b[i][j]; } }</pre>
---	--

Figure 7.19: The first case of the optimized MetaFork code for matrix addition

<p>Constraints:</p> $\begin{cases} 3B_0B_1 \leq Z_B < 3sB_0B_1 \\ 6 \leq R_B \end{cases}$ <p>strategies (1) (3b) (4a) (3a) (2) (2) applied or</p> $\begin{cases} 3B_0B_1 \leq Z_B < 3sB_0B_1 \\ 6 \leq R_B < 7 \end{cases}$ <p>strategies (2) (2) (3b) (1) (4a) (3a) applied</p>	<pre> int N, B0, B1, s = 1, dim0 = N/B0, dim1 = N/(B1*s); int a[N][N], b[N][N], c[N*N]; meta_schedule cache(a, b, c) { meta_for (int v0 = 0; v0 < dim0; v0++) meta_for (int v1 = 0; v1 < dim1; v1++) meta_for (int u0 = 0; u0 < B0; u0++) meta_for (int u1 = 0; u1 < B1; u1++) { int i = v0*B0+u0; int j = v1*B1+u1; c[i*N+j] = a[i][j] + b[i][j]; } }</pre>
--	---

Figure 7.20: The second case of the optimized MetaFork code for matrix addition

<p>Constraints:</p> $\begin{cases} Z_B < 3B_0B_1 \\ 6 \leq R_B \end{cases}$ <p>strategies (1) (3b) (2) (2) (4b) applied or</p> $\begin{cases} Z_B < 3B_0B_1 \\ 6 \leq R_B < 7 \end{cases}$ <p>strategies (2) (2) (3b) (1) (4b) applied</p>	<pre> int N, B0, B1, s = 1, dim0 = N/B0, dim1 = N/(B1*s); int a[N][N], b[N][N], c[N*N]; meta_schedule { meta_for (int v0 = 0; v0 < dim0; v0++) meta_for (int v1 = 0; v1 < dim1; v1++) meta_for (int u0 = 0; u0 < B0; u0++) meta_for (int u1 = 0; u1 < B1; u1++) { int i = v0*B0+u0; int j = v1*B1+u1; c[i*N+j] = a[i][j] + b[i][j]; } }</pre>
--	--

Figure 7.21: The third case of the optimized MetaFork code for matrix addition

(4a) (3a) (2) (2), the first case generates the optimized MetaFork code shown in Figure 7.25.

Applying the optimization strategy codes in a sequence either (1) (3b) (4a) (3a) (2) (2) or (2) (2) (3b) (1) (4a) (3a), the second case generates the optimized MetaFork code shown in Figure 7.26.

Applying the optimization strategy codes in a sequence either (1) (3b) (2) (2) (4b) or (2)

Constraints:

$$\begin{cases} 2sB_0B_1 \leq Z_B \\ 6 \leq R_B \end{cases}$$

strategies (1) (4a) (3a) (2) (2) applied

```

int N, B0, B1, s, dim0 = N/B0, dim1 = N/(B1*s);
int a[N][N], c[N*N];

meta_schedule cache(a, c) {
  meta_for (int v0 = 0; v0 < dim0; v0++)
    meta_for (int v1 = 0; v1 < dim1; v1++)
      meta_for (int u0 = 0; u0 < B0; u0++)
        meta_for (int u1 = 0; u1 < B1; u1++)
          for (int k = 0; k < s; ++k) {
            int i = v0*B0+u0;
            int j = (v1*s+k)*B1+u1;
            c[j*N+i] = a[i][j];
          }
}

```

Figure 7.22: The first case of the optimized MetaFork code for matrix transpose

Constraints:

$$\begin{cases} 2B_0B_1 \leq Z_B < 2sB_0B_1 \\ 5 \leq R_B \end{cases}$$

strategies (1) (3b) (4a) (3a) (2) (2) applied
or

$$\begin{cases} 2B_0B_1 \leq Z_B < 2sB_0B_1 \\ 5 \leq R_B < 6 \end{cases}$$

strategies (2) (2) (3b) (1) (4a) (3a) applied

```

int N, B0, B1, s = 1, dim0 = N/B0, dim1 = N/(B1*s);
int a[N][N], c[N*N];

meta_schedule cache(a, c) {
  meta_for (int v0 = 0; v0 < dim0; v0++)
    meta_for (int v1 = 0; v1 < dim1; v1++)
      meta_for (int u0 = 0; u0 < B0; u0++)
        meta_for (int u1 = 0; u1 < B1; u1++) {
          int i = v0*B0+u0;
          int j = v1*B1+u1;
          c[j*N+i] = a[i][j];
        }
}

```

Figure 7.23: The second case of the optimized MetaFork code for matrix transpose

Constraints:

$$\begin{cases} Z_B < 2B_0B_1 \\ 5 \leq R_B \end{cases}$$

strategies (1) (3b) (2) (2) (4b) applied
or

$$\begin{cases} Z_B < 2B_0B_1 \\ 5 \leq R_B < 6 \end{cases}$$

strategies (2) (2) (3b) (1) (4b) applied

```

int N, B0, B1, s = 1, dim0 = N/B0, dim1 = N/(B1*s);
int a[N][N], c[N*N];

meta_schedule {
  meta_for (int v0 = 0; v0 < dim0; v0++)
    meta_for (int v1 = 0; v1 < dim1; v1++)
      meta_for (int u0 = 0; u0 < B0; u0++)
        meta_for (int u1 = 0; u1 < B1; u1++) {
          int i = v0*B0+u0;
          int j = v1*B1+u1;
          c[j*N+i] = a[i][j];
        }
}

```

Figure 7.24: The third case of the optimized MetaFork code for matrix transpose

(2) (3b) (1) (4b), the third case generates the optimized MetaFork code shown in Figure 7.27.

Constraints:

$$\begin{cases} sB_0B_1 + sBB_1 + B_0B \leq Z_B \\ 9 \leq R_B \end{cases}$$

strategies (1) (4a) (3a) (2) (2) applied

```

int N, B0, B1, s, dim1 = N/(B1*s);
int dim0 = N/B0, B = min(B0, B1), dim = N/B;
int a[N][N], b[N][N], c[N*N];

meta_schedule cache(a, b, c) {
  meta_for (int v0 = 0; v0 < dim0; v0++)
    meta_for (int v1 = 0; v1 < dim1; v1++)
      for (int w = 0; w < dim; w++)
        meta_for (int u0 = 0; u0 < B0; u0++)
          meta_for (int u1 = 0; u1 < B1; u1++)
            for (int k = 0; k < s; ++k) {
              int i = v0*B0+u0;
              int j = (v1*s+k)*B1+u1;
              for (int z = 0; z < B; z++) {
                int p = B*w+z;
                c[i*N+j] = c[i*N+j] +
                  a[i][p] * b[p][j];
              }
            }
      }
}

```

Figure 7.25: The first case of the optimized MetaFork code for matrix matrix multiplication

Constraints:

$$\begin{cases} B_0B_1 + BB_1 + B_0B \leq Z_B < sB_0B_1 + sBB_1 + B_0B \\ 8 \leq R_B \end{cases}$$

strategies (1) (3b) (4a) (3a) (2) (2) applied
or

$$\begin{cases} B_0B_1 + BB_1 + B_0B \leq Z_B < sB_0B_1 + sBB_1 + B_0B \\ 8 \leq R_B < 9 \end{cases}$$

strategies (2) (2) (3b) (1) (4a) (3a) applied

```

int N, B0, B1, s = 1, dim1 = N/(B1*s);
int dim0 = N/B0, B = min(B0, B1), dim = N/B;
int a[N][N], b[N][N], c[N*N];

meta_schedule cache(a, b, c) {
  meta_for (int v0 = 0; v0 < dim0; v0++)
    meta_for (int v1 = 0; v1 < dim1; v1++)
      for (int w = 0; w < dim; w++)
        meta_for (int u0 = 0; u0 < B0; u0++)
          meta_for (int u1 = 0; u1 < B1; u1++)
            {
              int i = v0*B0+u0;
              int j = v1*B1+u1;
              for (int z = 0; z < B; z++) {
                int p = B*w+z;
                c[i*N+j] = c[i*N+j] +
                  a[i][p] * b[p][j];
              }
            }
      }
}

```

Figure 7.26: The second case of the optimized MetaFork code for matrix matrix multiplication

7.5 Conclusion

In this chapter, we proposed a *comprehensive optimization* algorithm that optimizes the input code fragment depending on unknown machine and program parameters; meanwhile, we realized a *proof-of-concept* implementation for generating *comprehensive* parametric MetaFork programs, in the form of a case distinction based on the possible values of the machine and program parameters.

<p style="text-align: center;">Constraints:</p> $\begin{cases} Z_B < B_0B_1 + BB_1 + B_0B \\ 8 \leq R_B \end{cases}$ <p style="text-align: center;">strategies (1) (3b) (2) (2) (4b) applied or</p> $\begin{cases} Z_B < B_0B_1 + BB_1 + B_0B \\ 8 \leq R_B < 9 \end{cases}$ <p style="text-align: center;">strategies (2) (2) (3b) (1) (4b) applied</p>	<pre> int N, B0, B1, s = 1, dim1 = N/(B1*s); int dim0 = N/B0, B = min(B0, B1), dim = N/B; int a[N][N], b[N][N], c[N*N]; meta_schedule { meta_for (int v0 = 0; v0 < dim0; v0++) meta_for (int v1 = 0; v1 < dim1; v1++) for (int w = 0; w < dim; w++) meta_for (int u0 = 0; u0 < B0; u0++) meta_for (int u1 = 0; u1 < B1; u1++) { int i = v0*B0+u0; int j = v1*B1+u1; for (int z = 0; z < B; z++) { int p = B*w+z; c[i*N+j] = c[i*N+j] + a[i][p] * b[p][j]; } } } } </pre>
--	---

Figure 7.27: The third case of the optimized MetaFork code for matrix matrix multiplication

The comprehensive optimization algorithm that we proposed takes optimization strategies, resource counters and performance counters into account; we implemented two resource counters and four optimization strategies in this comprehensive optimization algorithm.

With this preliminary implementation, experimentation shows that given a MetaFork program, three scenarios of optimized MetaFork programs are generated, each of them with a system of constraints specifying when the corresponding code is valid.

In addition, from the experimental results, we observe that different sequences of the optimization strategies yield the same optimized MetaFork program. However, since some optimization strategies are applied to the intermediate representation of the source code, the corresponding improvements are not shown in Section 7.4.

Chapter 8

Conclusion and Future Work

Our ultimate goal was to generate optimized CUDA kernels, where machine and program parameters are unknown symbols and can be, respectively, determined and optimized when the CUDA code is installed on the targeted hardware.

In our route to this goal, we developed a model of multithreaded computation (MCM) targeting many-core machines so as to measure the *work*, *span* and *parallelism overhead* of a CUDA-like program as well as to estimate the overall execution time of such program. Our experimentation shows that this model was able to determine the value ranges of some program parameters so that one can obtain a program with reduced parallelism overheads.

We acknowledge the fact that the MCM abstract machines admit a few simplifications and limitations with respect to actual many-core devices. Some factors, such as bank conflicts of shared memory and register spilling, increase memory access time because of the hardware limits of a streaming multiprocessor; however, these factors are ignored by our model. It is desirable to enhance our model with such features so as to obtain more accurate complexity estimates of a CUDA-like program. We leave this task for future work.

Taking advantage of the sophisticated CUDA code generator in PPCG [115], we realized a preliminary implementation of automatic code translator from MetaFork [29] programs to CUDA programs. In our automatic code generator, the number of threads per thread-block is generated as a symbol, whereas this number is generated as a numerical value by other tools, in particular PPCG.

In order to obtain optimized CUDA kernels, we also tuned a program parameter representing the granularity of a thread in a thread-block. Experimental results indicate that the CUDA kernels with these program parameters can lead to significant performance improvement and that the optimal choices for program parameters may depend on the problem size.

In addition, the non-linear expressions supported by our automatic code generator may involve parameters related to thread-block formats but not to the granularity of threads. Relaxing this limitation is a work in progress. On the other hand, our BPAS library provides efficient, parallel algorithms dealing with polynomial arithmetic operations; hence, it is desirable to integrate support from the BPAS library into our MetaFork-to-CUDA code generation framework.

Ultimately, we have proposed a *comprehensive optimization* algorithm that, given unknown machine and program parameters, optimizes the input code fragment in the form of a case discussion based on the possible values of the machine and program parameters. In the preliminary implementation, we considered optimizing MetaFork programs using LLVM as inter-

mediate representation. In the future, this algorithm should be implemented within the **Meta-Fork-to-CUDA** and should use PTX as intermediate representation.

Bibliography

- [1] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3–28, 1990.
- [2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Reading: Addison Wesley Publishing Company, 1986.
- [3] Ed Anderson, Zhaojun Bai, Jack Dongarra, Anne Greenbaum, A. McKenney, Jeremy Du Croz, Sven Hammarling, James Demmel, Christian H. Bischof, and Danny C. Sorensen. LAPACK: a portable linear algebra library for high-performance computers. In Joanne L. Martin, Daniel V. Pryor, and Gary Montry, editors, *Proceedings Supercomputing '90, New York, NY, USA, November 12-16, 1990*, pages 2–11. IEEE Computer Society, 1990.
- [4] Philippe Aubry, Daniel Lazard, and Marc Moreno Maza. On the theories of triangular sets. *Journal of Symbolic Computation*, 28(1):105–124, 1999.
- [5] László Babai and Endre Szemerédi. On the complexity of matrix group problems I. In *Foundations of Computer Science, 1984. 25th Annual Symposium on*, pages 229–240. IEEE, 1984.
- [6] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction, CC'10/ETAPS'10*, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag.
- [7] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, PACT '04*, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.
- [8] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in real algebraic geometry*, volume 10 of *Algorithms and Computations in Mathematics*. Springer-Verlag, 2006.
- [9] Carlo Bertolli, Samuel F. Antao, Alexandre E. Eichenberger, Kevin O'Brien, Zehra Sura, Arpith C. Jacob, Tong Chen, and Olivier Sallenave. Coordinating GPU threads for OpenMP 4.0 in LLVM. In *Proceedings of the 2014 LLVM Compiler Infrastructure*

- in *HPC*, LLVM-HPC '14, pages 12–21, Piscataway, NJ, USA, 2014. IEEE Computer Society.
- [10] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216. ACM, 1995.
- [11] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.
- [12] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.
- [13] OpenMP Architecture Review Board. OpenMP application program interface, version 4.0. <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>, 2013.
- [14] Marco Bodrato and Alberto Zanoni. Integer and polynomial multiplication: Towards optimal Toom-Cook matrices. In *Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, pages 17–24. ACM, 2007.
- [15] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Notices*, volume 43, pages 101–113. ACM, 2008.
- [16] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system I: The user language. *Journal of Symbolic Computation*, 24(3):235–265, 1997.
- [17] Richard P. Brent and H.T. Kung. Systolic VLSI arrays for polynomial GCD computation. *IEEE Transactions on Computers*, 33(8):731–736, 1984.
- [18] Christopher W. Brown. QEPCAD B: A program for computing with semi-algebraic sets using CADs. *ACM SIGSAM Bulletin*, 37(4):97–108, 2003.
- [19] Changbo Chen, Xiaohui Chen, Abdoul-Kader Keita, Marc Moreno Maza, and Ning Xie. MetaFork: A compilation framework for concurrency models targeting hardware accelerators and its application to the generation of parametric CUDA kernels. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, pages 70–79. IBM Corp., 2015.
- [20] Changbo Chen, Svyatoslav Covanov, Farnam Mansouri, Marc Moreno Maza, Ning Xie, and Yuzhen Xie. The basic polynomial algebra subprograms. In *International Congress on Mathematical Software*, pages 669–676. Springer, 2014.
- [21] Changbo Chen, Svyatoslav Covanov, Farnam Mansouri, Marc Moreno Maza, Ning Xie, and Yuzhen Xie. Parallel multiplication of dense polynomials with integer coefficient. *Accepted by 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'16)*, 2016.

- [22] Changbo Chen, James H. Davenport, John P. May, Marc Moreno Maza, Bican Xia, and Rong Xiao. Triangular decomposition of semi-algebraic systems. *Journal of Symbolic Computation*, 49:3–26, 2013.
- [23] Changbo Chen, James H. Davenport, Marc Moreno Maza, Bican Xia, and Rong Xiao. Computing with semi-algebraic sets represented by triangular decomposition. In *Proceedings of the 36th international symposium on Symbolic and algebraic computation*, pages 75–82. ACM, 2011.
- [24] Changbo Chen, Oleg Golubitsky, François Lemaire, Marc Moreno Maza, and Wei Pan. Comprehensive triangular decomposition. In *International Workshop on Computer Algebra in Scientific Computing*, pages 73–101. Springer, 2007.
- [25] Changbo Chen and Marc Moreno Maza. Algorithms for computing triangular decompositions of polynomial systems. In *Proceedings of the 36th international symposium on Symbolic and algebraic computation*, pages 83–90. ACM, 2011.
- [26] Changbo Chen and Marc Moreno Maza. An incremental algorithm for computing cylindrical algebraic decompositions. In *Computer Mathematics, 10th Asian Symposium (ASCM 2012)*, pages 199–221. Springer, 2012.
- [27] Changbo Chen and Marc Moreno Maza. Quantifier elimination by cylindrical algebraic decomposition based on regular chains. In *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 91–98, 2014.
- [28] Changbo Chen, Marc Moreno Maza, and Yuzhen Xie. Cache complexity and multicore implementation for univariate real root isolation. In *Journal of Physics: Conference Series*, volume 341, page 012026. IOP Publishing, 2012.
- [29] Xiaohui Chen, Marc Moreno Maza, Sushek Shekar, and Priya Unnikrishnan. MetaFork: A framework for concurrency platforms targeting multicores. In *Using and Improving OpenMP for Devices, Tasks, and More - 10th International Workshop on OpenMP, IWOMP 2014, Brazil, September 28-30, 2014. Proceedings*, pages 30–44, 2014.
- [30] Clang. A C language family frontend for LLVM. <http://clang.llvm.org/>, 2005.
- [31] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 20–23, 1975*, pages 134–183. Springer, 1975.
- [32] George E. Collins and Alkiviadis G. Akritas. Polynomial real root isolation using Descartes’s rule of signs. In *Proceedings of the third ACM symposium on Symbolic and algebraic computation*, pages 272–275. ACM, 1976.
- [33] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.
- [34] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press Cambridge, 2001.

- [35] Intel Corporation. Intel CilkPlus language extension specification, version 1.1. https://software.intel.com/sites/default/files/m/6/a/3/0/7/37679-Intel_Cilk_plus_lang_spec_2.htm, 2013.
- [36] NVIDIA Corporation. NVIDIA’s next generation CUDA compute architecture: Fermi. http://www.nvidia.com/object/IO_89570.html, 2009.
- [37] NVIDIA Corporation. NVIDIA next generation CUDA compute architecture: Kepler GK110. <http://www.nvidia.ca/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>, 2012.
- [38] NVIDIA Corporation. Compiler driver NVCC. [http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#options-for-steering-gpu-code-generation\(09.07.2013\)](http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#options-for-steering-gpu-code-generation(09.07.2013)), 2013.
- [39] NVIDIA Corporation. Parallel thread execution ISA: v4.3. <http://docs.nvidia.com/cuda/parallel-thread-execution/#axzz4HdnS1mUC>, 2015.
- [40] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [41] Leonardo Dagum and Ramesh Menon. OpenMP: An industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.
- [42] Xavier Dahan, Abdulilah Kadri, and Éric Schost. Bit-size estimates for triangular sets in positive dimension. *Journal of Complexity*, 28(1):109–135, 2012.
- [43] Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. FFPACK: finite field linear algebra package. In Jaime Gutierrez, editor, *Symbolic and Algebraic Computation, International Symposium ISSAC 2004, Santander, Spain, July 4-7, 2004, Proceedings*, pages 119–126. ACM, 2004.
- [44] Jean-Guillaume Dumas, Erich L. Kaltofen, and Clément Pernet, editors. *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation, PASCO 2015, Bath, United Kingdom, July 10-12, 2015*. ACM, 2015.
- [45] Paul Feautrier. Parametric integer programming. *Revue française d’automatique, d’informatique et de recherche opérationnelle*, 22(3):243–268, 1988.
- [46] Paul Feautrier. Automatic parallelization in the polytope model. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, volume 1132 of *Lecture Notes in Computer Science*, pages 79–103. Springer, 1996.
- [47] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

- [48] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.
- [49] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE Computer Society, 1999.
- [50] Mickaël Gastineau and Jacques Laskar. Highly scalable multiplication for distributed sparse multivariate polynomials on many-core systems. In *International Workshop on Computer Algebra in Scientific Computing*, pages 100–115. Springer, 2013.
- [51] Phillip B. Gibbons. A more practical PRAM model. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168. ACM, 1989.
- [52] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, pages 638–648, 1997.
- [53] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.
- [54] Armin Größlinger, Martin Griebel, and Christian Lengauer. Introducing non-linear parameters to the polyhedron model. In *Proceedings of 11th Workshop on Compilers for Parallel Computers (CPC 2004), Research Report Series*, pages 1–12, 2004.
- [55] Armin Größlinger, Martin Griebel, and Christian Lengauer. Quantifier elimination in automatic loop parallelization. *Journal of Symbolic Computation*, 41(11):1206–1221, 2006.
- [56] OpenACC Working Group et al. The OpenACC application programming interface. <http://www.openacc.org/>, 2011.
- [57] Tianyi D. Han and Tarek S. Abdelrahman. hiCUDA: A high-level directive based language for GPU programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2*, pages 52–61, New York, NY, USA, 2009.
- [58] Tianyi D. Han and Tarek S. Abdelrahman. Reducing branch divergence in GPU programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 3. ACM, 2011.
- [59] Sardar A. Haque, Xin Li, Farnam Mansouri, Marc Moreno Maza, Wei Pan, and Ning Xie. Dense arithmetic over finite fields with the CUMODP library. In *International Congress on Mathematical Software*, pages 725–732. Springer, 2014.
- [60] Sardar A. Haque and Marc Moreno Maza. Plain polynomial arithmetic on GPU. In *Journal of Physics: Conference Series*, volume 385, page 012014. IOP Publishing, 2012.

- [61] Sardar A. Haque, Marc Moreno Maza, and Ning Xie. A many-core machine model for designing algorithms with minimum parallelism overheads. In Gerhard R. Joubert, Hugh Leather, Mark Parsons, Frans J. Peters, and Mark Sawyer, editors, *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK*, volume 27 of *Advances in Parallel Computing*, pages 35–44. IOS Press, 2015.
- [62] William Hart, Fredrik Johansson, and Sebastian Pancratz. FLINT fast library for number theory version 2.3. 0. 2011.
- [63] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The Cilkview scalability analyzer. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 145–156. ACM, 2010.
- [64] Anthony C. Hearn. REDUCE: The first forty years. In *Invited paper presented at the A3L Conference in Honor of the 60th Birthday of Volker Weispfenning*, 2005.
- [65] Karin Högstedt, Larry Carter, and Jeanne Ferrante. Determining the idle time of a tiling. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 160–173, New York, NY, USA, 1997. ACM.
- [66] Justin Holewinski, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM.
- [67] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.
- [68] The LLVM Compiler Infrastructure. User guide for NVPTX back-end. <http://llvm.org/docs/NVPTXUsage.html>, 2013-2016.
- [69] Richard D Jenks and Robert S Sutor. *Axiom: The scientific computation system*. Springer, 1992.
- [70] Michael Kalkbrener. *Three contributions to elimination theory*. PhD thesis, 1991.
- [71] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
- [72] Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, and Jacqueline Chame. A script-based autotuning compiler system to generate high-performance CUDA code. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):31:1–31:25, January 2013.
- [73] David B. Kirk and W. Hwu Wen-mei. *Programming massively parallel processors: A hands-on approach*. Elsevier, 2013.

- [74] Donald E. Knuth. *The Art of Computer Programming, Vol. II: Seminumerical Algorithms*. Addison-Wesley, 1969.
- [75] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.
- [76] C. L. Lawson, Richard J. Hanson, D. R. Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.
- [77] Chuck L. Lawson, Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.
- [78] Daniel Lazard and Renaud Rioboo. Integration of rational functions: Rational computation of the logarithmic part. *Journal of Symbolic Computation*, 9(2):113–115, 1990.
- [79] Charles E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, 2010.
- [80] Xin Li, Marc Moreno Maza, Raqeeb Rasheed, and Éric Schost. The Modpn library: Bringing fast polynomial arithmetic into Maple. *ACM Communications in Computer Algebra*, 42(3):172–174, 2009.
- [81] Weiguo Liu, Wolfgang Muller-Wittig, and Bertil Schmidt. Performance predictions for general-purpose computation on GPUs. In *2007 International Conference on Parallel Processing (ICPP 2007)*, pages 50–50. IEEE Computer Society, 2007.
- [82] Yang Lu, Zhang Jingzhong, and Zeng Zhenbing. Searching dependency between algebraic equations: An algorithm applied to automated reasoning. In *Institute of Mathematics and its Applications Conference Series*, volume 51, pages 147–147. Oxford University Press, 1994.
- [83] Zhengyi Lu, Bi He, Yong Luo, and Lu Pan. An algorithm of real root isolation for polynomial systems. *Proceedings of Symbolic Numeric Computation*, pages 94–107, 2005.
- [84] David G. Luenberger and Yinyu Ye. *Linear and nonlinear programming*, volume 2. Springer, 1984.
- [85] Lin Ma, Kunal Agrawal, and Roger D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Computer Systems*, 30:202–215, 2014.
- [86] Lin Ma and Roger D. Chamberlain. A performance model for memory bandwidth constrained applications on graphics engines. In *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, pages 24–31. IEEE Computer Society, 2012.

- [87] Farnam Mansouri. On the parallelization of integer polynomial multiplication. Master's thesis, The University of Western Ontario, 2014.
- [88] Marc Moreno Maza and Jean-Louis Roch, editors. *Proceedings of the 4th International Workshop on Parallel Symbolic Computation, PASCO 2010, July 21-23, 2010, Grenoble, France*. ACM, 2010.
- [89] Marc Moreno Maza and Stephen M. Watt, editors. *Parallel Symbolic Computation, PASCO 2007, International Workshop, 27-28 July 2007, University of Western Ontario, London, Ontario, Canada*. ACM, 2007.
- [90] Leon Mirsky. A dual of Dilworth's decomposition theorem. *The American Mathematical Monthly*, 78(8):876–877, 1971.
- [91] Michael Monagan and Roman Pearce. Parallel sparse polynomial multiplication using heaps. In *Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, pages 263–270. ACM, 2009.
- [92] Marc Moreno Maza and Wei Pan. Fast polynomial arithmetic on a GPU. In *Journal of Physics: Conference Series*, volume 256, 2010.
- [93] Marc Moreno Maza and Yuzhen Xie. Balanced dense polynomial multiplication on multi-cores. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 1–9. IEEE Computer Society, 2009.
- [94] Marc Moreno Maza and Yuzhen Xie. FFT-based dense polynomial arithmetic on multi-cores. In *High Performance Computing Systems and Applications*, pages 378–399. Springer, 2010.
- [95] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.
- [96] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.
- [97] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, September 1999.
- [98] Markus Püschel, José MF Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *International Journal of High Performance Computing Applications*, 18(1):21–45, 2004.
- [99] Helge Rhodin. A PTX code generator for LLVM. Master's thesis, Saarland University, 2010.
- [100] Arch D. Robison. Composable parallel patterns with Intel CilkPlus. *Computing in Science and Engineering*, 15(2):66–71, 2013.

- [101] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. A programming language interface to describe transformations and code generation. In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, LCPC'10, pages 136–150, Berlin, Heidelberg, 2011. Springer-Verlag.
- [102] Greg Ruetsch and Paulius Micikevicius. Optimizing matrix transpose in CUDA. *Nvidia CUDA SDK Application Note*, 18, 2009.
- [103] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for many-core GPUs. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE Computer Society, 2009.
- [104] A. Schönhage and Volker Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7(3-4):281–292, 1971.
- [105] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.
- [106] Jaewook Shin. Introducing control flow into vectorized code. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 280–291. IEEE Computer Society, 2007.
- [107] Jun Shirako, Priya Unnikrishnan, Sanjay Chatterjee, Kelvin Li, and Vivek Sarkar. Expressing doacross loop dependences in OpenMP. In *IWOMP*, volume 8122 of *Lecture Notes in Computer Science*, pages 30–44. Springer, 2013.
- [108] Victor Shoup et al. NTL: A library for doing number theory. <http://www.shoup.net/ntl/>, 2001.
- [109] Thomas G. Stockham Jr. High-speed convolution and correlation. In *Proceedings of the April 26-28, 1966, Spring joint computer conference*, pages 229–233. ACM, 1966.
- [110] Larry Stockmeyer and Uzi Vishkin. Simulation of parallel random access machines by circuits. *SIAM Journal on Computing*, 13(2):409–422, 1984.
- [111] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.
- [112] Agnes Szanto. *Computation with polynomial systems*. PhD thesis, 1999.
- [113] Xiaonan Tian, Rengan Xu, Yonghong Yan, Zhifeng Yun, Sunita Chandrasekaran, and Barbara Chapman. Compiling a high-level directive-based programming model for GPGPUs. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 105–120. Springer, 2013.
- [114] Swapneela Unkule, Christopher Shaltz, and Apan Qasem. Automatic restructuring of GPU kernels for exploiting inter-thread data locality. In *International Conference on Compiler Construction*, pages 21–40. Springer, 2012.

- [115] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization TACO*, 9(4):54, 2013.
- [116] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*, 2012.
- [117] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. Schedule trees. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques. Vienna, Austria*, 2014.
- [118] Vasily Volkov. Better performance at lower occupancy. http://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf, 2010. Presentation at the GPU Technology Conference, GTC.
- [119] Joachim Von Zur Gathen and Jürgen Gerhard. Fast algorithms for Taylor shifts and certain difference equations. In *Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, pages 40–47. ACM, 1997.
- [120] Joachim Von Zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge university press, 2013.
- [121] R. Clint Whaley. ATLAS (automatically tuned linear algebra software). In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 95–101. Springer, 2011.
- [122] H Paul Williams. Fourier’s method of linear programming and its dual. *The American mathematical monthly*, 93(9):681–695, 1986.
- [123] Christian Wimmer and Michael Franz. Linear scan register allocation on SSA form. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '10*, pages 170–179, New York, NY, USA, 2010. ACM.
- [124] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. In *ACM Sigplan Notices*, volume 26, pages 30–44. ACM, 1991.
- [125] Stephen Wolfram. *Mathematica: A system for doing mathematics by computer*. Addison Wesley Longman Publishing Co., Inc., 1991.
- [126] Bican Xia and Ting Zhang. Real solution isolation using interval arithmetic. *Computers & Mathematics with Applications*, 52(6):853–860, 2006.

Appendix A

Sample Code in the BPAS Library

We collect sample code from the BPAS library, including two adaptive algorithms shown in Appendix A.1 and some user interface code for BPAS rings and polynomials in Appendix A.2.

A.1 Adaptive algorithms

```
/* *****
 * Internal multiplication function for integer univariate polynomials *
 * It is an adaptive algorithm based on input sizes and available cores. *
 * @param mul, a vector of coefficients of the product polynomial *
 * @param a, a vector of coefficients over Z representing a polynomial *
 * @param n, the size of the above vector *
 * @param b, a vector of coefficients over Z representing a polynomial *
 * @param m, the size of the above vector *
 * ***** */
void univariateMultiplication(mpz_class* mul, mpz_class* a, int n,
                             mpz_class* b, int m) {
    int size = n + m - 1;
    // UnivariateIntegerPolynomial: internal data structure
    // for integer univariate polynomials
    UnivariateIntegerPolynomial aPoly(n, a);
    UnivariateIntegerPolynomial bPoly(m, b);
    UnivariateIntegerPolynomial rPoly(size, mul);

    if (size < 64) { // Naive multiplication; serial
        MulNaive naive;
        naive.multiply(&aPoly, &bPoly, &rPoly);
    }
    else if (size < 2048) { // Kronecker substitution method; serial
        MulKS ks;
        ks.multiply(&aPoly, &bPoly, &rPoly);
    }
    else if (size < 4096) { // Toom Cook method; parallel
        int worker = __cilkrts_get_nworkers();
```

```

    if (worker <= 6) { // 4-way
        MulToom4 toom4;
        toom4.multiply(&aPoly, &bPoly, &rPoly, 4);
    }
    else { // 8-way
        MulToom8 toom8;
        toom8.multiply(&aPoly, &bPoly, &rPoly, 8);
    }
}
else { // Two convolution method; parallel
    MulSSA ssa;
    ssa.multiply(&aPoly, &bPoly, &rPoly);
}
}

/*****
 * The adaptive Taylor shift algorithm combining a divide and conquer method, *
 * aka Algorithm (E) in [119], with the algorithm in [28] as the base case. *
 * @param a, a vector of coefficients over Z representing a polynomial *
 * @param m, the size of the above vector, power of 2 *
 * @param bi, a vector of coefficients representing  $(x+1)^k$ , for all  $1 \leq k \leq m/2$  *
 * @param B, the size determining whether to switch to Algorithm (E) *
 *****/
void taylorShiftBasePower2(mpz_class* a, int m, mpz_class* bi, int B) {
    int d = (B > m)? m : B;

    // Base case, calling to the algorithm in [28]
    cilk_for (int i = 0; i < m/d; ++i)
        taylorShiftIncrementalCilkFor(&a[i*B], d, 16);

    mpz_class* mul = new mpz_class[m];
    for (int i = B; i < m; i <= 1) {
        cilk_for (int k = 0; k < m/(2*i); ++k)
            univariateMultiplication(&mul[2*k*i], &a[2*k*i+i], i, &bi[i-1], i);
        for (int k = 0; k < m; ++k)
            a[k] += mul[k];
    }
    delete [] mul;
}

```

A.2 User interfaces

```

/*****
 * A child class of abstract BPASRing implements a ring *
 * structure in the sense of commutative algebra. *
 *****/
class BPASRing {

```

```

    public:
        static int characteristic;
        static bool isPrimeField;
        static bool isComplexField;
        virtual bool isZero() = 0;
        virtual void zero() = 0;
        virtual bool isOne() = 0;
        virtual void one() = 0;
        virtual bool isNegativeOne() = 0;
        virtual void negativeOne() = 0;
        virtual int isConstant() = 0;
};

typedef int DataType; // Different data type for each concrete polynomial

/*****
 * A child class of abstract BPASPolynomial implements a polynomial ring. *
 * At this level, the type of the coefficients and the number of variables *
 * are unspecified. Hence, the only member functions deal with assignment, *
 * stream writing and arithmetic operations. Those latter include: *
 * addition, subtraction, multiplication, exact division, exponentiation *
 * and equality test. Note that for the first four arithmetic operations, *
 * in-place versions are provided. *
 *****/
class BPASPolynomial : public BPASRing {
    public:
        BPASPolynomial& operator= (BPASPolynomial&);
        BPASPolynomial& operator+ (BPASPolynomial&);
        BPASPolynomial& operator+= (BPASPolynomial&);
        BPASPolynomial& operator- (BPASPolynomial&);
        BPASPolynomial& operator- ();
        BPASPolynomial& operator-= (BPASPolynomial&);
        BPASPolynomial& operator* (BPASPolynomial&);
        BPASPolynomial& operator*= (BPASPolynomial&);
        BPASPolynomial& operator/ (BPASPolynomial&);
        BPASPolynomial& operator/= (BPASPolynomial&);
        BPASPolynomial& operator^ (int);
        bool operator== (BPASPolynomial&);
        bool operator!= (BPASPolynomial&);
        friend std::ostream& operator<< (std::ostream&, BPASPolynomial&);
};

/*****
 * A child class of abstract BPASUnivariatePolynomial inherits from *
 * BPASPolynomial. Moreover, it implements a univariate polynomial ring *
 * in which a notion of Greatest Common Divisor (GCD) makes sense. *
 * Arithmetic operations (addition, subtraction, multiplication, exact *

```

```

* division) can take a polynomial and a coefficient as input arguments. *
* Arithmetic operations also include shift left and shift right (that *
* is, multiplication or division by a power of the variable) as well as *
* divisions (monic, pseudo and lazy), differentiation, evaluation, GCD *
* computation and square-free factorization. One can query the degree, *
* the leading coefficient, the coefficient of a prescribed monomial, *
* the variable name, the content of a polynomial. *
*****/
class BPASUnivariatePolynomial : public BPASPolynomial {
public:
    BPASUnivariatePolynomial& operator+ (DataType);
    BPASUnivariatePolynomial& operator+= (DataType);
    BPASUnivariatePolynomial& operator- (DataType);
    BPASUnivariatePolynomial& operator-= (DataType);
    BPASUnivariatePolynomial& operator* (DataType);
    BPASUnivariatePolynomial& operator*= (DataType);
    BPASUnivariatePolynomial& operator/ (DataType);
    BPASUnivariatePolynomial& operator/= (DataType);
    BPASUnivariatePolynomial& operator<< (int);
    BPASUnivariatePolynomial& operator<=< (int);
    BPASUnivariatePolynomial& operator>> (int);
    BPASUnivariatePolynomial& operator>=> (int);

    BPASUnivariatePolynomial& monicDivide(BPASUnivariatePolynomial&);
    BPASUnivariatePolynomial& monicDivide(BPASUnivariatePolynomial&,
        BPASUnivariatePolynomial*);
    BPASUnivariatePolynomial& lazyPseudoDivide(BPASUnivariatePolynomial&,
        DataType*, DataType*);
    BPASUnivariatePolynomial& lazyPseudoDivide(BPASUnivariatePolynomial&,
        BPASUnivariatePolynomial*, DataType*, DataType*);
    BPASUnivariatePolynomial& pseudoDivide(BPASUnivariatePolynomial&,
        DataType*);
    BPASUnivariatePolynomial& pseudoDivide(BPASUnivariatePolynomial&,
        BPASUnivariatePolynomial*, DataType*);
    virtual void differentiate(int) = 0;
    DataType content();
    BPASUnivariatePolynomial& gcd(BPASUnivariatePolynomial&);
    std::vector<BPASUnivariatePolynomial&> squareFree();
    virtual int degree() = 0;
    DataType leadingCoefficient();
    DataType coefficient(int);
    void setCoefficient(int, DataType);
    virtual void setVariableName (std::string) = 0;
    virtual std::string variable() = 0;
    virtual bool isTrailingCoefficientZero() = 0;
    DataType evaluate(DataType);
};

```


Appendix B

Theoretical Analysis of Fundamental Algorithms Using the MCM Model

With respect to using our MCM model to analyze fundamental algorithms in Chapter 4, we attach the PDF versions of the executable MAPLE worksheets of all applications.

Euclidean

This document is to demonstrate the analysis of the Euclidean algorithm

Given two polynomials a and b over a finite field, with $\deg(a) = n - 1$, and $\deg(b) = m - 1$, where $n \geq m$, return the GCD of (b, r) , where r is the remainder in the division of a by b

- Computing 1 coefficient in the intermediate remainder (in one division step) needs 3 steps along the span

1. multiplication factor
2. multiplication
3. subtraction

- Assumptions:

1. At each step, the degree of a or b decreases by 1
2. $\text{GCD}(a, b) = 1$

```
> restart;
```

```
> terms_a := n;
```

```
terms_a := n
```

(1)

```
> terms_b := m;
```

```
terms_b := m
```

(2)

```
[ > steps_for_one_coefficient := 3;
      steps_for_one_coefficient:= 3 ] (1)
```

Optimized Algorithm

- Phase 1, for the degree of a and b will decrease by $(n-1, m-1)$, $(n-s-1, m)$, $(n-2s-1, m-1)$, ..., $(m-1, m-1)$, such that there are $\frac{n-m}{s}$ kernel calls, and each kernel call has $\frac{m}{l}$ blocks (l divides m , and l is even).
- Phase 2, for the first kernel call, we use $\frac{m}{l}$ blocks, for the second kernel call, we use $\left\lfloor \frac{m - \frac{s}{2}}{l} \right\rfloor$ blocks, for the third kernel call, we use $\frac{m-s}{l}$ blocks, and so on, until the last kernel call, we use 1 block with $\frac{s}{2}$ terms. There are $\frac{2m}{s}$ kernel calls.
- For each block, it performs s division steps, and we use l threads, computing 1 leading coefficient and 2 other coefficients

```
[ > optimized_param := s;
      optimized_param:= s ] (1.1)
```

```
[ > optimized_threads := l;
      optimized_threads:= l ] (1.2)
```

```
[ > optimized_euclidean_kernels_phase1 := (terms_a-terms_b)
      /optimized_param;
      optimized_euclidean_kernels_phase1:=  $\frac{n-m}{s}$  ] (1.3)
```

```
[ > optimized_euclidean_kernels_phase2 := 2*
      terms_b/optimized_param;
      optimized_euclidean_kernels_phase2:=  $\frac{2m}{s}$  ] (1.4)
```

```
[ > optimized_euclidean_kernels :=
      optimized_euclidean_kernels_phase1+
      optimized_euclidean_kernels_phase2;
      optimized_euclidean_kernels:=  $\frac{n-m}{s} + \frac{2m}{s}$  ] (1.5)
```

```
[ > optimized_euclidean_first_blocks := terms_b/optimized_threads;
      optimized_euclidean_first_blocks:=  $\frac{m}{l}$  ] (1.6)
```

Work

```
> optimized_euclidean_phase1_workperblock := normal((sum(i,i=
1..optimized_param)+2*optimized_threads*optimized_param) *
steps_for_one_coefficient);
```

$$\text{optimized_euclidean_phase1_workperblock} := \frac{3}{2} s^2 + \frac{3}{2} s + 6 l s \quad (1.1.1)$$

```
> optimized_euclidean_phase2_workperblock := normal((2*sum(i,
i=optimized_param/2..optimized_param) + 2*optimized_threads*
optimized_param) * steps_for_one_coefficient);
```

$$\text{optimized_euclidean_phase2_workperblock} := \frac{9}{4} s^2 + \frac{9}{2} s + 6 l s \quad (1.1.2)$$

Phase 1: Work per block is the same as optimized division algorithm

Phase 2: For the last kernel call, it uses $\frac{s}{2}$ threads to compute the leading coefficients and l threads to compute other coefficients

Phase 2: For the rest $\frac{2m}{s} - 1$ kernel calls, it uses l threads to

compute the leading coefficients $(s + s - 1 + s - 1 + \dots + \frac{s}{2} + \frac{s}{2}$ for 1

block) and l threads to compute other coefficients

```
> optimized_euclidean_work := normal
(optimized_euclidean_kernels_phase1*
optimized_euclidean_first_blocks*
optimized_euclidean_phase1_workperblock
+
2*sum((terms_b - i*optimized_param/2)/optimized_threads,i=0.
.(terms_b/optimized_param)-1)*
optimized_euclidean_phase2_workperblock);
```

$$\text{optimized_euclidean_work} := \quad (1.1.3)$$

$$\frac{3}{8} \frac{m(8lm + 16ln + 8ls + 5ms + 4ns + 3s^2 + 14m + 4n + 6s)}{l}$$

Span

```
> optimized_euclidean_span := normal(normal
(optimized_euclidean_kernels*steps_for_one_coefficient*
optimized_param));
```

$$\text{optimized_euclidean_span} := 3n + 3m \quad (1.2.1)$$

Overhead

For each kernel calls, each block needs s leading coefficients and

2 s other coefficients of a and same of b , then it takes 6 time to load data and 2 time to write back

$$\begin{aligned} &> \text{optimized_euclidean_overheadperblock} := 8*U; \\ &\quad \text{optimized_euclidean_overheadperblock} := 8\ U \end{aligned} \quad (1.3.1)$$

$$\begin{aligned} &> \text{optimized_euclidean_overhead_a} := \\ &\quad \text{optimized_euclidean_kernels_phase1} * \\ &\quad \text{optimized_euclidean_first_blocks} * \\ &\quad \text{optimized_euclidean_overheadperblock}; \\ &\quad \text{optimized_euclidean_overhead_a} := \frac{8\ (n-m)\ m\ U}{s\ l} \end{aligned} \quad (1.3.2)$$

$$\begin{aligned} &> \text{optimized_euclidean_overhead_b} := \text{normal}(2*\text{sum}((\text{terms_b} - i * \\ &\quad \text{optimized_param}/2)/\text{optimized_threads}, i=0.. \\ &\quad \text{terms_b}/\text{optimized_param}-1) * \\ &\quad \text{optimized_euclidean_overheadperblock}); \\ &\quad \text{optimized_euclidean_overhead_b} := \frac{4\ m\ (3\ m + s)\ U}{l\ s} \end{aligned} \quad (1.3.3)$$

$$\begin{aligned} &> \text{optimized_euclidean_overhead} := \text{normal} \\ &\quad (\text{optimized_euclidean_overhead_a} + \\ &\quad \text{optimized_euclidean_overhead_b}); \\ &\quad \text{optimized_euclidean_overhead} := \frac{4\ m\ U\ (2\ n + m + s)}{l\ s} \end{aligned} \quad (1.3.4)$$

▼ Tp

$$\begin{aligned} &> \text{optimized_euclidean_workperthread} := \\ &\quad \text{steps_for_one_coefficient} * \text{optimized_param}; \\ &\quad \text{optimized_euclidean_workperthread} := 3\ s \end{aligned} \quad (1.4.1)$$

$$\begin{aligned} &> \text{optimized_euclidean_overheadperthread} := 8*U; \\ &\quad \text{optimized_euclidean_overheadperthread} := 8\ U \end{aligned} \quad (1.4.2)$$

$$\begin{aligned} &> \text{optimized_euclidean_C} := \text{optimized_euclidean_workperthread} + \\ &\quad \text{optimized_euclidean_overheadperthread}; \\ &\quad \text{optimized_euclidean_C} := 3\ s + 8\ U \end{aligned} \quad (1.4.3)$$

$$\begin{aligned} &> \text{optimized_euclidean_N} := \text{normal} \\ &\quad (\text{optimized_euclidean_kernels_phase1} * \\ &\quad \text{optimized_euclidean_first_blocks} \\ &\quad + \\ &\quad 2*\text{sum}((\text{terms_b} - i*\text{optimized_param}/2)/\text{optimized_threads}, i=0. \\ &\quad \text{terms_b}/\text{optimized_param}-1) \\ &\quad) ; \\ &\quad \text{optimized_euclidean_N} := \frac{1}{2} \frac{m\ (2\ n + m + s)}{l\ s} \end{aligned} \quad (1.4.4)$$

$$\begin{aligned} &> \text{optimized_euclidean_L} := \text{normal}(\text{optimized_euclidean_kernels}) \\ &\quad ; \end{aligned} \quad (1.4.5)$$

$$\text{optimized_euclidean_L} := \frac{n+m}{s} \quad (1.4.5)$$

> optimized_euclidean_K := optimized_euclidean_first_blocks;

$$\text{optimized_euclidean_K} := \frac{m}{l} \quad (1.4.6)$$

**> optimized_euclidean_Tp := normal(
(optimized_euclidean_N/optimized_euclidean_K+
optimized_euclidean_L)*optimized_euclidean_C);**

$$\text{optimized_euclidean_Tp} := \frac{1}{2} \frac{(4n+3m+s)(3s+8U)}{s} \quad (1.4.7)$$

> naive_euclidean_work := eval(optimized_euclidean_work, [s=1]);

$$\text{naive_euclidean_work} := \frac{3}{8} \frac{m(8lm+16ln+8l+19m+8n+9)}{l} \quad (2)$$

> naive_euclidean_span := eval(optimized_euclidean_span, [s=1]);

$$\text{naive_euclidean_span} := 3n+3m \quad (3)$$

**> naive_euclidean_overhead := eval(optimized_euclidean_overhead,
[s=1]);**

$$\text{naive_euclidean_overhead} := \frac{4mU(2n+m+1)}{l} \quad (4)$$

> naive_euclidean_Tp := eval(optimized_euclidean_Tp, [s=1]);

$$\text{naive_euclidean_Tp} := \frac{1}{2} (4n+3m+1)(3+8U) \quad (5)$$

**> work_ratio := normal
(naive_euclidean_work/optimized_euclidean_work);**

$$\text{work_ratio} := \frac{8lm+16ln+8l+19m+8n+9}{8lm+16ln+8ls+5ms+4ns+3s^2+14m+4n+6s} \quad (6)$$

**> span_ratio := normal
(naive_euclidean_span/optimized_euclidean_span);**

$$\text{span_ratio} := 1 \quad (7)$$

**> overhead_ratio := normal
(naive_euclidean_overhead/optimized_euclidean_overhead);**

$$\text{overhead_ratio} := \frac{(2n+m+1)s}{2n+m+s} \quad (8)$$

> Tp_ratio := normal(naive_euclidean_Tp/optimized_euclidean_Tp);

$$\text{Tp_ratio} := \frac{(4n+3m+1)(3+8U)s}{(4n+3m+s)(3s+8U)} \quad (9)$$

$s = O(l)$ on the condition that $4s+2l$ must fit into the local memory

Fast Fourier Transform

This documentation is to demonstrate the analysis of Cooley-Tukey and Stockham FFT algorithm on GPUs

$$DFT_{-}\{2^k\} = \prod_{i=0}^{k-1} (DFT_{-}\{2\} \times L_{-}\{2^{k-1}\}) (D_{-}\{2, 2^{k-i-1}\} \times L_{-}\{2^i\}) (L_{-}\{2\}^{2^{k-i}} \times L_{-}\{2^i\})$$

▼ Cooley-Tukey FFT algorithm

Phase 1: $x \rightarrow \prod_{i=0}^{k-1} \{i = k-l-1\} (L_{-}\{2^i\} \times L_{-}^{2^{k-i}}\{2\}) x$

Phase 2: $x \rightarrow (L_{-}\{2^{k-l}\} \times DFT_{-}\{m\}) x$

Phase 3:

$x \rightarrow \prod_{i=0}^{k-l-1} \{i = 0\} (L_{-}\{2^i\} \times DFT_{-}\{2\} \times L_{-}\{2^{k-i-1}\}) (L_{-}\{2^i\} \times D_{-}\{2, 2^{k-i-1}\}) x$

```
> cooleytukey_param := m;
    cooleytukey_param:= m (1.1)
```

```
> cooleytukey_threadspersblock := l;
    cooleytukey_threadspersblock:= l (1.2)
```

```
> cooleytukey_size := n;
    cooleytukey_size:= n (1.3)
```

First data_shuffle:

In list_transpose2, each thread moves 2 data, and it executes 9 additions, 11 bit operations.

Further, list_transpose2 is called $\log(n) - \log(s)$ times

Second list_fft, each thread moves s data. For each data, it executes

$\frac{s}{2} + \dots + 1$ with 2 modular

additions (each 5 arithmetic operations), 2 modular multiplications (each 11 arithmetic operations).

Also for each data, it executes 30 bit operations.

Additionally, 1 modular multiplication.

Third list_butterfly, each thread moves 2 data and reads 1 data, and it executes 6 additions, 6 bit operations, 2 modular additions.

Or compute jumped powers:

move 2 data, 2 modular additions, $2 \cdot (\log(i))$ calls to

fourier_reduction, with $i = 1 \dots l$, 1 additional call to fourier_reduction

In fourier_reduction, 11 bit operations, 5 additions, 1 multiplications

> **cooleytukey_phase3_workperblock_powers := 17 * 2 * simplify(sum(log[2](i), i=1..cooleytukey_threadspblock)) + cooleytukey_threadspblock * 27;**

$$cooleytukey_phase3_workperblock_powers := \frac{34 \ln(\Gamma(l+1))}{\ln(2)} + 27 l \quad (1.4)$$

> **cooleytukey_phase3_spanperkernel_powers := 17 * 2 * log[2](cooleytukey_threadspblock) + 27;**

$$cooleytukey_phase3_spanperkernel_powers := \frac{34 \ln(l)}{\ln(2)} + 27 \quad (1.5)$$

> **cooleytukey_phase3_overheadperblock_powers := 4 * U;**

$$cooleytukey_phase3_overheadperblock_powers := 4 U \quad (1.6)$$

Work

> **cooleytukey_phase1_kernels := log[2](cooleytukey_size)-log[2](cooleytukey_param);**

$$cooleytukey_phase1_kernels := \frac{\ln(n)}{\ln(2)} - \frac{\ln(m)}{\ln(2)} \quad (1.1.1)$$

> **cooleytukey_phase1_blocks := cooleytukey_size / cooleytukey_threadspblock;**

$$cooleytukey_phase1_blocks := \frac{n}{l} \quad (1.1.2)$$

> **cooleytukey_phase1_workperblock := 20 * cooleytukey_threadspblock;**

$$cooleytukey_phase1_workperblock := 20 l \quad (1.1.3)$$

> **cooleytukey_phase1_work := cooleytukey_phase1_kernels * cooleytukey_phase1_blocks * cooleytukey_phase1_workperblock;**

$$cooleytukey_phase1_work := 20 \left(\frac{\ln(n)}{\ln(2)} - \frac{\ln(m)}{\ln(2)} \right) n \quad (1.1.4)$$

> **cooleytukey_phase2_blocks := cooleytukey_size / (cooleytukey_threadspblock * cooleytukey_param);**

$$cooleytukey_phase2_blocks := \frac{n}{lm} \quad (1.1.5)$$

> **cooleytukey_phase2_workperblock := cooleytukey_threadspblock * simplify(cooleytukey_param * (32 * sum(cooleytukey_param/2^i, i=1..log[2]**

$$\begin{aligned} & (\text{cooleytukey_param})) + 11) + 30 * \text{cooleytukey_param}); \\ & \text{cooleytukey_phase2_workperblock} := l(32 m^2 + 9 m) \end{aligned} \quad (1.1.6)$$

$$\begin{aligned} & > \text{cooleytukey_phase2_work} := \text{cooleytukey_phase2_blocks} * \\ & \text{cooleytukey_phase2_workperblock}; \\ & \text{cooleytukey_phase2_work} := \frac{n(32 m^2 + 9 m)}{m} \end{aligned} \quad (1.1.7)$$

$$\begin{aligned} & > \text{cooleytukey_phase3_kernels} := \log[2](\text{cooleytukey_size}) - \log \\ & [2](\text{cooleytukey_param}); \\ & \text{cooleytukey_phase3_kernels} := \frac{\ln(n)}{\ln(2)} - \frac{\ln(m)}{\ln(2)} \end{aligned} \quad (1.1.8)$$

$$\begin{aligned} & > \text{cooleytukey_phase3_blocks} := \text{cooleytukey_size} / \\ & \text{cooleytukey_threadspersblock}; \\ & \text{cooleytukey_phase3_blocks} := \frac{n}{l} \end{aligned} \quad (1.1.9)$$

$$\begin{aligned} & > \text{cooleytukey_phase3_workperblock} := 22 * \\ & \text{cooleytukey_threadspersblock}; \\ & \text{cooleytukey_phase3_workperblock} := 22 l \end{aligned} \quad (1.1.10)$$

$$\begin{aligned} & > \text{cooleytukey_phase3_work} := \text{cooleytukey_phase3_kernels} * \\ & \text{cooleytukey_phase3_blocks} * \text{cooleytukey_phase3_workperblock}; \\ & \text{cooleytukey_phase3_work} := 22 \left(\frac{\ln(n)}{\ln(2)} - \frac{\ln(m)}{\ln(2)} \right) n \end{aligned} \quad (1.1.11)$$

$$\begin{aligned} & > \text{cooleytukey_phase3_work_powers} := \text{simplify} \\ & (\text{cooleytukey_phase3_kernels} * \text{cooleytukey_phase3_blocks} * \\ & \text{cooleytukey_phase3_workperblock_powers}); \\ & \text{cooleytukey_phase3_work_powers} := \\ & \frac{(\ln(n) - \ln(m)) n (27 l \ln(2) + 34 \ln(\Gamma(l+1)))}{\ln(2)^2 l} \end{aligned} \quad (1.1.12)$$

$$\begin{aligned} & > \text{cooleytukey_work} := \text{normal}(\text{cooleytukey_phase1_work} + \\ & \text{cooleytukey_phase2_work} + \text{cooleytukey_phase3_work}); \\ & \text{cooleytukey_work} := \frac{n(32 \ln(2) m + 42 \ln(n) + 9 \ln(2) - 42 \ln(m))}{\ln(2)} \end{aligned} \quad (1.1.13)$$

$$\begin{aligned} & > \text{cooleytukey_work1} := \text{eval}(\text{cooleytukey_work}, [m=16]); \\ & \text{cooleytukey_work1} := \frac{n(353 \ln(2) + 42 \ln(n))}{\ln(2)} \end{aligned} \quad (1.1.14)$$

$$\begin{aligned} & > \text{cooleytukey_work_powers} := \text{normal}(\text{cooleytukey_phase1_work} + \\ & \text{cooleytukey_phase2_work} + \text{cooleytukey_phase3_work_powers}); \\ & \text{cooleytukey_work_powers} := \frac{1}{\ln(2)^2 l} (n(32 \ln(2)^2 m l + 47 l \ln(2) \ln(n) \\ & + 9 \ln(2)^2 l - 47 l \ln(2) \ln(m) + 34 \ln(n) \ln(\Gamma(l+1)) - 34 \ln(\Gamma(l \\ & + 1)) \ln(m))) \end{aligned} \quad (1.1.15)$$

$$> \text{cooleytukey_work1_powers} := \text{expand}(\text{eval}$$

$$\begin{aligned}
& (\text{cooleytukey_work_powers}, [m=16])); \\
\text{cooleytukey_work1_powers} &:= 333 n + \frac{47 n \ln(n)}{\ln(2)} + \frac{34 n \ln(n) \ln(\Gamma(l) l)}{\ln(2)^2 l} \\
& - \frac{136 n \ln(\Gamma(l) l)}{\ln(2) l}
\end{aligned} \quad (1.1.16)$$

Span

$$\begin{aligned}
& > \text{cooleytukey_phase1_span} := 20 * \text{cooleytukey_phase1_kernels}; \\
\text{cooleytukey_phase1_span} &:= \frac{20 \ln(n)}{\ln(2)} - \frac{20 \ln(m)}{\ln(2)}
\end{aligned} \quad (1.2.1)$$

$$\begin{aligned}
& > \text{cooleytukey_phase2_span} := \text{simplify}((32 * \log[2] \\
& (\text{cooleytukey_param}) * \text{sum}(\text{cooleytukey_param}/2^i, i=1..\log[2] \\
& (\text{cooleytukey_param})) + 30 * \text{cooleytukey_param}) + 11); \\
\text{cooleytukey_phase2_span} &:= \frac{32 \ln(m) (-1 + m)}{\ln(2)} + 30 m + 11
\end{aligned} \quad (1.2.2)$$

$$\begin{aligned}
& > \text{cooleytukey_phase3_span} := 22 * \text{cooleytukey_phase3_kernels}; \\
\text{cooleytukey_phase3_span} &:= \frac{22 \ln(n)}{\ln(2)} - \frac{22 \ln(m)}{\ln(2)}
\end{aligned} \quad (1.2.3)$$

$$\begin{aligned}
& > \text{cooleytukey_phase3_span_powers} := \text{simplify} \\
& (\text{cooleytukey_phase3_kernels} * \\
& \text{cooleytukey_phase3_spanperkernel_powers}); \\
\text{cooleytukey_phase3_span_powers} &:= \frac{(\ln(n) - \ln(m)) (34 \ln(l) + 27 \ln(2))}{\ln(2)^2}
\end{aligned} \quad (1.2.4)$$

$$\begin{aligned}
& > \text{cooleytukey_span} := \text{normal}(\text{cooleytukey_phase1_span} + \\
& \text{cooleytukey_phase2_span} + \text{cooleytukey_phase3_span}); \\
\text{cooleytukey_span} &:= \frac{30 \ln(2) m + 32 \ln(m) m + 42 \ln(n) + 11 \ln(2) - 74 \ln(m)}{\ln(2)}
\end{aligned} \quad (1.2.5)$$

$$\begin{aligned}
& > \text{cooleytukey_span1} := \text{eval}(\text{cooleytukey_span}, [m=16]); \\
\text{cooleytukey_span1} &:= \frac{2243 \ln(2) + 42 \ln(n)}{\ln(2)}
\end{aligned} \quad (1.2.6)$$

$$\begin{aligned}
& > \text{cooleytukey_span_powers} := \text{normal}(\text{cooleytukey_phase1_span} + \\
& \text{cooleytukey_phase2_span} + \text{cooleytukey_phase3_span_powers}); \\
\text{cooleytukey_span_powers} &:= \frac{1}{\ln(2)^2} (30 m \ln(2)^2 + 32 \ln(m) \ln(2) m \\
& + 34 \ln(n) \ln(l) - 34 \ln(m) \ln(l) + 47 \ln(n) \ln(2) + 11 \ln(2)^2 \\
& - 79 \ln(m) \ln(2))
\end{aligned} \quad (1.2.7)$$

$$\begin{aligned}
& > \text{cooleytukey_span1_powers} := \text{expand}(\text{eval} \\
& (\text{cooleytukey_span_powers}, [m=16]));
\end{aligned} \quad (1.2.8)$$

$$\begin{aligned} \text{coleytukey_span1_powers} := & 2223 + \frac{34 \ln(n) \ln(l)}{\ln(2)^2} - \frac{136 \ln(l)}{\ln(2)} \\ & + \frac{47 \ln(n)}{\ln(2)} \end{aligned} \quad (1.2.8)$$

▼ Overhead

$$\begin{aligned} & > \text{coleytukey_phase1_overheadperblocka} := 2 * U + 2 * U * \\ & \quad \text{coleytukey_threadperblock}; \\ & \quad \text{coleytukey_phase1_overheadperblocka} := 2 U l + 2 U \end{aligned} \quad (1.3.1)$$

$$\begin{aligned} & > \text{coleytukey_phase1_overheadperblockb} := 4 * U; \\ & \quad \text{coleytukey_phase1_overheadperblockb} := 4 U \end{aligned} \quad (1.3.2)$$

$$\begin{aligned} & > \text{coleytukey_phase1_overhead} := \text{simplify}(\log[2] \\ & \quad (\text{coleytukey_threadperblock}) * \text{coleytukey_phase1_blocks} * \\ & \quad \text{coleytukey_phase1_overheadperblocka} + \\ & \quad (\text{coleytukey_phase1_kernels} - \log[2] \\ & \quad (\text{coleytukey_threadperblock})) * \text{coleytukey_phase1_blocks} * \\ & \quad \text{coleytukey_phase1_overheadperblockb}); \\ & \text{coleytukey_phase1_overhead} := \frac{2 n U (\ln(l) l - \ln(l) + 2 \ln(n) - 2 \ln(m))}{l \ln(2)} \end{aligned} \quad (1.3.3)$$

$$\begin{aligned} & > \text{coleytukey_phase2_overheadperblock} := 2 * \text{coleytukey_param} \\ & \quad * U; \\ & \quad \text{coleytukey_phase2_overheadperblock} := 2 m U \end{aligned} \quad (1.3.4)$$

$$\begin{aligned} & > \text{coleytukey_phase2_overhead} := \text{coleytukey_phase2_blocks} * \\ & \quad \text{coleytukey_phase2_overheadperblock}; \\ & \quad \text{coleytukey_phase2_overhead} := \frac{2 n U}{l} \end{aligned} \quad (1.3.5)$$

$$\begin{aligned} & > \text{coleytukey_phase3_overheadperblock} := 5 * U; \\ & \quad \text{coleytukey_phase3_overheadperblock} := 5 U \end{aligned} \quad (1.3.6)$$

$$\begin{aligned} & > \text{coleytukey_phase3_overhead} := \text{coleytukey_phase3_kernels} * \\ & \quad \text{coleytukey_phase3_blocks} * \\ & \quad \text{coleytukey_phase3_overheadperblock}; \\ & \quad \text{coleytukey_phase3_overhead} := \frac{5 \left(\frac{\ln(n)}{\ln(2)} - \frac{\ln(m)}{\ln(2)} \right) n U}{l} \end{aligned} \quad (1.3.7)$$

$$\begin{aligned} & > \text{coleytukey_phase3_overhead_powers} := \\ & \quad \text{coleytukey_phase3_kernels} * \text{coleytukey_phase3_blocks} * \\ & \quad \text{coleytukey_phase3_overheadperblock_powers}; \\ & \quad \text{coleytukey_phase3_overhead_powers} := \frac{4 \left(\frac{\ln(n)}{\ln(2)} - \frac{\ln(m)}{\ln(2)} \right) n U}{l} \end{aligned} \quad (1.3.8)$$

$$> \text{coleytukey_overhead} := \text{normal}(\text{coleytukey_phase1_overhead} +$$

cooleytukey_phase2_overhead + cooleytukey_phase3_overhead);
cooleytukey_overhead:= (1.3.9)

$$\frac{n U (2 \ln(l) l - 2 \ln(l) + 9 \ln(n) + 2 \ln(2) - 9 \ln(m))}{l \ln(2)}$$

> cooleytukey_overhead1 := eval(cooleytukey_overhead, [m=16]);
cooleytukey_overhead1:= $\frac{n U (2 \ln(l) l - 2 \ln(l) + 9 \ln(n) - 34 \ln(2))}{l \ln(2)}$ (1.3.10)

> cooleytukey_overhead_powers := normal
(cooleytukey_phase1_overhead + cooleytukey_phase2_overhead +
cooleytukey_phase3_overhead_powers);
cooleytukey_overhead_powers:= (1.3.11)

$$\frac{2 n U (\ln(l) l - \ln(l) + 4 \ln(n) + \ln(2) - 4 \ln(m))}{l \ln(2)}$$

> cooleytukey_overhead1_powers := expand(eval
(cooleytukey_overhead_powers, [m=16]));
cooleytukey_overhead1_powers:= $\frac{2 n U \ln(l)}{\ln(2)} - \frac{2 n U \ln(l)}{l \ln(2)} + \frac{8 n U \ln(n)}{l \ln(2)}$ (1.3.12)

$$- \frac{30 n U}{l}$$

▼ Tp

**> cooleytukey_N := normal(eval(cooleytukey_phase1_kernels *
cooleytukey_phase1_blocks + cooleytukey_phase2_blocks +
cooleytukey_phase3_kernels * cooleytukey_phase3_blocks, [s=
16]));**
cooleytukey_N:= $\frac{n (2 m \ln(n) - 2 \ln(m) m + \ln(2))}{l \ln(2) m}$ (1.4.1)

**> cooleytukey_L := eval(cooleytukey_phase1_kernels + 1 +
cooleytukey_phase3_kernels, [s=16]);**
cooleytukey_L:= $\frac{2 \ln(n)}{\ln(2)} - \frac{2 \ln(m)}{\ln(2)} + 1$ (1.4.2)

**> cooleytukey_C := eval(cooleytukey_phase2_span +
cooleytukey_phase1_overheadperblocka, [s=16]);**
cooleytukey_C:= $\frac{32 \ln(m) (-1 + m)}{\ln(2)} + 30 m + 11 + 2 U l + 2 U$ (1.4.3)

> cooleytukey_K := cooleytukey_phase1_blocks;
cooleytukey_K:= $\frac{n}{l}$ (1.4.4)

**> cooleytukey_Tp := normal((cooleytukey_N/cooleytukey_K +
cooleytukey_L) * cooleytukey_C);**
(1.4.5)

$$\text{cooleytukey_Tp} := \frac{1}{m \ln(2)^2} ((4 m \ln(n) + \ln(2) m - 4 \ln(m) m + \ln(2)) (2 U \ln(2) + 2 U \ln(2) + 30 \ln(2) m + 32 \ln(m) m + 11 \ln(2) - 32 \ln(m))) \quad (1.4.5)$$

$$\begin{aligned} &> \text{cooleytukey_Tp1} := \text{eval}(\text{cooleytukey_Tp}, [m=16]); \\ \text{cooleytukey_Tp1} &:= \frac{1}{16} \frac{(64 \ln(n) - 239 \ln(2)) (2 U \ln(2) + 2 U \ln(2) + 2411 \ln(2))}{\ln(2)^2} \end{aligned} \quad (1.4.6)$$

$$\begin{aligned} &> \text{cooleytukey_C_powers} := \text{eval} \\ &(\text{cooleytukey_phase3_spanperkernel_powers} + \\ &\text{cooleytukey_phase1_overheadperblocka}, [s=16]); \\ \text{cooleytukey_C_powers} &:= \frac{34 \ln(l)}{\ln(2)} + 27 + 2 U \ln(2) + 2 U \end{aligned} \quad (1.4.7)$$

$$\begin{aligned} &> \text{cooleytukey_Tp_powers} := \text{normal}((\text{cooleytukey_N}/\text{cooleytukey_K} \\ &+ \text{cooleytukey_L}) * \text{cooleytukey_C_powers}); \\ \text{cooleytukey_Tp_powers} &:= \frac{1}{m \ln(2)^2} ((4 m \ln(n) + \ln(2) m - 4 \ln(m) m + \ln(2)) (2 U \ln(2) + 2 U \ln(2) + 34 \ln(l) + 27 \ln(2))) \end{aligned} \quad (1.4.8)$$

$$\begin{aligned} &> \text{cooleytukey_Tp1_powers} := \text{eval}(\text{cooleytukey_Tp_powers}, [m=16]) ; \\ \text{cooleytukey_Tp1_powers} &:= \frac{1}{16} \frac{1}{\ln(2)^2} ((64 \ln(n) - 239 \ln(2)) (2 U \ln(2) + 2 U \ln(2) + 34 \ln(l) + 27 \ln(2))) \end{aligned} \quad (1.4.9)$$

Cooley-Tukey with vs without pre-computed jumped powers

$$\begin{aligned} &> \text{Rwork} := \text{normal}(\text{cooleytukey_work1_powers} / \text{cooleytukey_work1}); \\ \text{Rwork} &:= \frac{333 \ln(2)^2 l + 47 l \ln(2) \ln(n) + 34 \ln(n) \ln(\Gamma(l) l) - 136 \ln(\Gamma(l) l) \ln(2)}{\ln(2) l (353 \ln(2) + 42 \ln(n))} \end{aligned} \quad (1)$$

$$\begin{aligned} &> \text{Rspan} := \text{normal}(\text{cooleytukey_span1_powers} / \text{cooleytukey_span1}); \\ \text{Rspan} &:= \frac{2223 \ln(2)^2 + 34 \ln(n) \ln(l) - 136 \ln(2) \ln(l) + 47 \ln(n) \ln(2)}{\ln(2) (2243 \ln(2) + 42 \ln(n))} \end{aligned} \quad (2)$$

$$\begin{aligned} &> \text{Roverhead} := \text{normal}(\text{cooleytukey_overhead1_powers} / \\ &\text{cooleytukey_overhead1}); \\ \text{Roverhead} &:= \frac{2 (\ln(l) l - \ln(l) + 4 \ln(n) - 15 \ln(2))}{2 \ln(l) l - 2 \ln(l) + 9 \ln(n) - 34 \ln(2)} \end{aligned} \quad (3)$$

$$\begin{aligned} &> \text{Rt} := \text{normal}(\text{cooleytukey_Tp1_powers} / \text{cooleytukey_Tp1}); \\ \text{Rt} &:= \frac{2 U \ln(2) + 2 U \ln(2) + 34 \ln(l) + 27 \ln(2)}{\ln(2) (2 U \ln(2) + 2 U + 2411)} \end{aligned} \quad (4)$$

Stockham FFT algorithm

For each fixed $0 \leq i < k$, there are three computational steps:

$$\text{Phase 1: } x \rightarrow \left(L_{-}\{2\}^{2^{k-i}} \times L_{-}\{2^i\} \right) x$$

$$\text{Phase 2: } x \rightarrow \left(D_{-}\{2, 2^{k-i-1}\} \times L_{-}\{2^i\} \right) x$$

$$\text{Phase 3: } x \rightarrow \left(DFT_{-}\{2\} \times L_{-}\{2^{k-1}\} \right) x$$

```
> stockham_threadspersblock := l;
    stockham_threadspersblock := l
```

(2.1)

```
> stockham_size := n;
    stockham_size := n
```

(2.2)

First compute phase 3 $DFT_{-}\{2\} \times L_{-}\{2^{k-1}\}$ once, a thread reading 2 data + writing back the same + executing 3 bit operations, 3 additions, 2 modular add operations (5 arithmetic operations)

Second, for each $0 \leq i < k$, compute phase 1, 2 & 3

In phase 1, if $2^i < l$, a thread reading 1 data and writing back the same + executing 14 bit operations, 10 additions, 1 multiplications.

If $2^i \geq l$, a thread reading 1 data and writing back the same + executing 11 bit operations, 8 additions.

In phase 2, a thread reading 1 data and writing back the same + executing 5 bit operations, 4 additions, 1 modular multiply operations (11 arithmetic operations)

Prefix sum to compute the powers of the n-th primitive root of unity

Each thread deals with 4 data items

```
> roots_blocks := stockham_size/(4*stockham_threadspersblock);
    roots_blocks :=  $\frac{1}{4} \frac{n}{l}$ 
```

(2.1.1)

Regard to the prefix sum, one block performs the up-sweep phase, with 3 instructions, and then performs the down-sweep phase, with 5 instructions, regard to Listing 2 in scan.pdf. (Except mediate step)

```
> roots_inclusive_workperblock := normal(simplify( 8*(sum
  (stockham_threadspersblock/2^(i-1),i=1..log[2]
  (stockham_threadspersblock))+2)+1 ));
  roots_inclusive_workperblock:= 1 + 16 l
```

(2.1.2)

```
> roots_inclusive_overheadperblock := (4+4)*U;
  roots_inclusive_overheadperblock:= 8 U
```

(2.1.3)

```
> roots_inclusive_work := roots_blocks*
  roots_inclusive_workperblock;
  roots_inclusive_work:=  $\frac{1}{4} \frac{n(1+16l)}{l}$ 
```

(2.1.4)

```
> roots_inclusive_span := simplify( 8*(log[2]
  (stockham_threadspersblock)+2)+1 );
  roots_inclusive_span:=  $\frac{8 \ln(l)}{\ln(2)} + 17$ 
```

(2.1.5)

```
> roots_inclusive_overhead := roots_blocks*
  roots_inclusive_overheadperblock;
  roots_inclusive_overhead:=  $\frac{2 n U}{l}$ 
```

(2.1.6)

```
> roots_mediate_overheadperblock := (4+4)*U;
  roots_mediate_overheadperblock:= 8 U
```

(2.1.7)

```
> roots_mediate_work := normal(simplify( 8*(sum
  (stockham_threadspersblock/2^(i-1),i=1..log[2]
  (stockham_threadspersblock))+2)+1));
  roots_mediate_work:= 1 + 16 l
```

(2.1.8)

```
> roots_mediate_span := 8*(log[2](stockham_threadspersblock)+2)
  +1;
  roots_mediate_span:=  $\frac{8 \ln(l)}{\ln(2)} + 17$ 
```

(2.1.9)

```
> roots_mediate_overhead := roots_mediate_overheadperblock;
  roots_mediate_overhead:= 8 U
```

(2.1.10)

```
# Scan Block Sums
```

```
> roots_exclusive_workperblock := 4*stockham_threadspersblock;
  roots_exclusive_workperblock:= 4 l
```

(2.1.11)

```
> roots_exclusive_overheadperblock := (4+1+4)*U;
  roots_exclusive_overheadperblock:= 9 U
```

(2.1.12)

```
> roots_exclusive_work := (roots_blocks-1)*
  roots_exclusive_workperblock;
  roots_exclusive_work:=  $4 \left( \frac{1}{4} \frac{n}{l} - 1 \right) l$ 
```

(2.1.13)

```
> roots_exclusive_span := 4;
  roots_exclusive_span:= 4
```

(2.1.14)

```
> roots_exclusive_overhead := (roots_blocks-1)*
```

```
roots_exclusive_overheadperblock;
```

$$roots_exclusive_overhead := 9 \left(\frac{1}{4} \frac{n}{l} - 1 \right) U \quad (2.1.15)$$

```
# Total work & span in phase 2
```

```
> roots_work := normal(roots_inclusive_work+
  roots_mediate_work+roots_exclusive_work);
```

$$roots_work := \frac{1}{4} \frac{48 l^2 + 20 l n + 4 l + n}{l} \quad (2.1.16)$$

```
> roots_span := simplify(roots_inclusive_span+
  roots_mediate_span+roots_exclusive_span);
```

$$roots_span := \frac{16 \ln(l)}{\ln(2)} + 38 \quad (2.1.17)$$

```
> roots_overhead := normal(roots_inclusive_overhead+
  roots_mediate_overhead+roots_exclusive_overhead);
```

$$roots_overhead := -\frac{1}{4} \frac{U(-17n + 4l)}{l} \quad (2.1.18)$$

Work

```
> stockham_phase1_blocks := stockham_size /
  stockham_threadperblock;
```

$$stockham_phase1_blocks := \frac{n}{l} \quad (2.2.1)$$

```
> stockham_phase1_workperblock := 25 *
  stockham_threadperblock;
```

$$stockham_phase1_workperblock := 25 l \quad (2.2.2)$$

```
> stockham_phase1_work := stockham_phase1_blocks *
  stockham_phase1_workperblock;
```

$$stockham_phase1_work := 25 n \quad (2.2.3)$$

```
> stockham_phase2_blocks := stockham_size / (2 *
  stockham_threadperblock);
```

$$stockham_phase2_blocks := \frac{1}{2} \frac{n}{l} \quad (2.2.4)$$

```
> stockham_phase2_workperblock := 20 *
  stockham_threadperblock;
```

$$stockham_phase2_workperblock := 20 l \quad (2.2.5)$$

```
> stockham_phase2_work := stockham_phase2_blocks *
  stockham_phase2_workperblock;
```

$$stockham_phase2_work := 10 n \quad (2.2.6)$$

```
> stockham_phase3_blocks := stockham_size / (2 *
  stockham_threadperblock);
```

$$(2.2.7)$$

$$\text{stockham_phase3_blocks} := \frac{1}{2} \frac{n}{l} \quad (2.2.7)$$

$$\begin{aligned} &> \text{stockham_phase3_workperblock} := 16 * \\ &\quad \text{stockham_threadsperblock}; \\ &\quad \text{stockham_phase3_workperblock} := 16 l \end{aligned} \quad (2.2.8)$$

$$\begin{aligned} &> \text{stockham_phase3_work} := \text{stockham_phase3_blocks} * \\ &\quad \text{stockham_phase3_workperblock}; \\ &\quad \text{stockham_phase3_work} := 8 n \end{aligned} \quad (2.2.9)$$

$$\begin{aligned} &> \text{stockham_work} := \text{normal}(\text{roots_work} + \text{sum} \\ &\quad (\text{stockham_phase1_work} + \text{stockham_phase2_work} + \\ &\quad \text{stockham_phase3_work}, i=0..\log[2](\text{stockham_size})-2) + \\ &\quad \text{stockham_phase3_work}); \\ &\quad \text{stockham_work} := \\ &\quad \frac{1}{4} \frac{172 n l \ln(n) + 48 \ln(2) l^2 - 120 n l \ln(2) + 4 l \ln(2) + n \ln(2)}{l \ln(2)} \end{aligned} \quad (2.2.10)$$

Span

$$\begin{aligned} &> \text{stockham_phase1_span} := 25; \\ &\quad \text{stockham_phase1_span} := 25 \end{aligned} \quad (2.3.1)$$

$$\begin{aligned} &> \text{stockham_phase2_span} := 10; \\ &\quad \text{stockham_phase2_span} := 10 \end{aligned} \quad (2.3.2)$$

$$\begin{aligned} &> \text{stockham_phase3_span} := 8; \\ &\quad \text{stockham_phase3_span} := 8 \end{aligned} \quad (2.3.3)$$

$$\begin{aligned} &> \text{stockham_span} := \text{normal}(\text{roots_span} + \text{sum} \\ &\quad (\text{stockham_phase1_span} + \text{stockham_phase2_span} + \\ &\quad \text{stockham_phase3_span}, i=0..\log[2](n)-2) + \\ &\quad \text{stockham_phase3_span}); \\ &\quad \text{stockham_span} := \frac{16 \ln(l) + 3 \ln(2) + 43 \ln(n)}{\ln(2)} \end{aligned} \quad (2.3.4)$$

Overhead

$$\begin{aligned} &> \text{stockham_phase1_overheadperblock} := 2 * U; \\ &\quad \text{stockham_phase1_overheadperblock} := 2 U \end{aligned} \quad (2.4.1)$$

$$\begin{aligned} &> \text{stockham_phase1_overhead} := \text{stockham_phase1_blocks} * \\ &\quad \text{stockham_phase1_overheadperblock}; \\ &\quad \text{stockham_phase1_overhead} := \frac{2 n U}{l} \end{aligned} \quad (2.4.2)$$

$$\begin{aligned} &> \text{stockham_phase2_overheadperblock} := 2 * U; \\ &\quad \text{stockham_phase2_overheadperblock} := 2 U \end{aligned} \quad (2.4.3)$$

$$\begin{aligned} &> \text{stockham_phase2_overhead} := \text{stockham_phase2_blocks} * \\ &\quad \text{stockham_phase2_overheadperblock}; \\ &\quad \text{stockham_phase2_overhead} := \frac{2 n U}{l} \end{aligned} \quad (2.4.4)$$

$$\text{stockham_phase2_overhead} := \frac{nU}{l} \quad (2.4.4)$$

$$\begin{aligned} &> \text{stockham_phase3_overheadperblock} := 4*U; \\ &\quad \text{stockham_phase3_overheadperblock} := 4U \end{aligned} \quad (2.4.5)$$

$$\begin{aligned} &> \text{stockham_phase3_overhead} := \text{stockham_phase3_blocks} * \\ &\quad \text{stockham_phase3_overheadperblock}; \\ &\quad \text{stockham_phase3_overhead} := \frac{2nU}{l} \end{aligned} \quad (2.4.6)$$

$$\begin{aligned} &> \text{stockham_overhead} := \text{normal}(\text{roots_overhead} + \text{sum} \\ &\quad (\text{stockham_phase1_overhead} + \text{stockham_phase2_overhead} + \\ &\quad \text{stockham_phase3_overhead}, i=0..\log[2](\text{stockham_size})-2) + \\ &\quad \text{stockham_phase3_overhead}); \\ &\quad \text{stockham_overhead} := \frac{1}{4} \frac{U(20n\ln(n) - 4l\ln(2) + 5n\ln(2))}{l\ln(2)} \end{aligned} \quad (2.4.7)$$

▼ Tp

$$\begin{aligned} &> \text{stockham_N} := \text{normal}(\text{roots_blocks} + \text{sum} \\ &\quad (\text{stockham_phase1_blocks} + \text{stockham_phase2_blocks} + \\ &\quad \text{stockham_phase3_blocks}, i=0..\log[2](\text{stockham_size})-2) + \\ &\quad \text{stockham_phase3_blocks}); \\ &\quad \text{stockham_N} := \frac{1}{4} \frac{n(-5\ln(2) + 8\ln(n))}{l\ln(2)} \end{aligned} \quad (2.5.1)$$

$$\begin{aligned} &> \text{stockham_L} := \text{normal}(3 + \text{sum}(3, i=0..\log[2](\text{stockham_size}) \\ &\quad -2) + 1); \\ &\quad \text{stockham_L} := \frac{\ln(2) + 3\ln(n)}{\ln(2)} \end{aligned} \quad (2.5.2)$$

$$\begin{aligned} &> \text{stockham_C} := \text{roots_inclusive_span} + \\ &\quad \text{stockham_phase3_overheadperblock}; \\ &\quad \text{stockham_C} := \frac{8\ln(l)}{\ln(2)} + 17 + 4U \end{aligned} \quad (2.5.3)$$

$$\begin{aligned} &> \text{stockham_K} := \text{stockham_phase1_blocks}; \\ &\quad \text{stockham_K} := \frac{n}{l} \end{aligned} \quad (2.5.4)$$

$$\begin{aligned} &> \text{stockham_Tp} := \text{normal}((\text{stockham_N}/\text{stockham_K} + \text{stockham_L}) * \\ &\quad \text{stockham_C}); \\ &\quad \text{stockham_Tp} := \frac{1}{4} \frac{(-\ln(2) + 20\ln(n)) (4U\ln(2) + 8\ln(l) + 17\ln(2))}{\ln(2)^2} \end{aligned} \quad (2.5.5)$$

Cooley-Tukey without pre-computed jumped powers vs Stockham

$$\begin{aligned} &> \text{work_ratio} := \text{normal}(\text{cooley_tukey_work1_powers} / \text{stockham_work}); \\ &\quad \text{work_ratio} := \\ &\quad (4n(333\ln(2)^2l + 47l\ln(2)\ln(n) + 34\ln(n)\ln(\Gamma(l)l)) \end{aligned} \quad (5)$$

$$\begin{aligned}
& -136 \ln(\Gamma(l) l) \ln(2)) / (\ln(2) (172 n l \ln(n) + 48 \ln(2) l^2 - 120 n l \ln(2) \\
& + 4 l \ln(2) + n \ln(2))) \\
& \text{> span_ratio := normal(cooley_tukey_span1_powers / stockham_span);} \\
& \text{span_ratio := } \frac{2223 \ln(2)^2 + 34 \ln(n) \ln(l) - 136 \ln(2) \ln(l) + 47 \ln(n) \ln(2)}{\ln(2) (16 \ln(l) + 3 \ln(2) + 43 \ln(n))} \quad (6) \\
& \text{> overhead_ratio := normal(cooley_tukey_overhead1_powers /} \\
& \text{stockham_overhead);} \\
& \text{overhead_ratio := } \frac{8 n (\ln(l) l - \ln(l) + 4 \ln(n) - 15 \ln(2))}{20 n \ln(n) - 4 l \ln(2) + 5 n \ln(2)} \quad (7) \\
& \text{> Tp_ratio := normal(cooley_tukey_Tp1_powers / stockham_Tp);} \\
& \text{Tp_ratio :=} \quad (8) \\
& \frac{1}{4} \frac{(64 \ln(n) - 239 \ln(2)) (2 U l \ln(2) + 2 U \ln(2) + 34 \ln(l) + 27 \ln(2))}{(-\ln(2) + 20 \ln(n)) (4 U \ln(2) + 8 \ln(l) + 17 \ln(2))}
\end{aligned}$$

Plain Polynomial Multiplication Algorithm

This document is to demonstrate the analysis of plain polynomial multiplication algorithm

Given two polynomials a and b over a finite field, with $\deg(a) = n - 1$, and $\deg(b) = m - 1$, where $n \geq m$, compute the product of $a \times b$

```
> restart;
> terms_a := n;
                                terms_a := n
```

(1)

```
> terms_b := m;
                                terms_b := m
```

(2)

```
> multiplication_param := s;
                                multiplication_param := s
```

(3)

```
> multiplication_threadsperblock := l;
                                multiplication_threadsperblock := l
```

(4)

```
> multiplication_length := terms_a + multiplication_param - 1;
                                multiplication_length := n + s - 1
```

(5)

```
> multiplication_rows := terms_b/multiplication_param;
                                multiplication_rows := m/s
```

(6)

Phase 1:

- The grid of thread blocks is 2-D. Principle: each thread block computes some of the coefficient products and some of the partial sums toward the polynomial product. Those partial sums are stored in an auxiliary array, which is later processed in Phase 2.
- One row of thread blocks contributes to $y = n + s - 1$ terms of the product of $a \times b$, while there are $x = \frac{m}{s}$ rows of blocks.
- If we use l threads per block, then we have $\frac{x \cdot y}{s \cdot l}$ thread-blocks.
- Each thread in each thread block performs s^2 coefficient products and $s \cdot (s - 1)$ additions.
- Each block needs s terms of b and $l \cdot s + s - 1$ terms of a . Then it computes $s \cdot l$ elements of the $x \times y$ matrix.

```
> multiplication_phase1_blocks := multiplication_length *
multiplication_rows/(multiplication_threadsperblock *
multiplication_param);
                                multiplication_phase1_blocks := (n + s - 1) m / (s^2 l)
```

(7)

Phase 2:

- The x rows of the auxiliary array M are added pairwise in $\log(x)$ parallel steps. At each step, the number of rows (to be added) is divided by two
- At a given parallel step, each thread reads s elements from one row and s from another row
- Therefore at the i -th parallel step (starting at $i=0$ and using l threads per block) one reads 2 times s times l coefficients per block
- The total amount of data (coefficients) to be added at the beginning at the i -th parallel is twice less than at the previous step, thus it is x times y divided by 2^i
- Therefore, at the i -th parallel step, using l threads per block, we need $\frac{x \cdot y}{s \cdot l \cdot 2^{i+1}}$ blocks and we have $\log_2 x$ parallel steps for the whole phase.
- Each thread to compute at most s addition needs s element of *one row* and s element from another row, then writes back s elements.

```
> multiplication_phase2_blocks := proc(i)
    return multiplication_length*multiplication_rows/
    (multiplication_param*multiplication_threadsperblock*2^(i));
end proc;
> multiplication_phase2_steps := simplify(log[2]
    (multiplication_rows));
```

$$\text{multiplication_phase2_steps} := \frac{\ln\left(\frac{m}{s}\right)}{\ln(2)} \quad (8)$$

Work

```
> multiplication_phase1_workperblock :=
    multiplication_threadsperblock*multiplication_param*
    (multiplication_param+multiplication_param-1);
    multiplication_phase1_workperblock := l s (2 s - 1) \quad (1.1)
```

```
> multiplication_phase1_work := multiplication_phase1_blocks*
    multiplication_phase1_workperblock;
    multiplication_phase1_work := \frac{(n + s - 1) m (2 s - 1)}{s} \quad (1.2)
```

```
> multiplication_phase2_workperblock :=
    multiplication_threadsperblock*multiplication_param;
    multiplication_phase2_workperblock := l s \quad (1.3)
```

```
> multiplication_phase2_work := simplify(sum
    (multiplication_phase2_blocks(i)*
```

$$\begin{aligned} & \text{multiplication_phase2_workperblock}, i=1.. \\ & \text{multiplication_phase2_steps}) + \text{multiplication_length}/2); \\ & \text{multiplication_phase2_work} := \frac{1}{2} \frac{2 m n + 2 m s - n s - s^2 - 2 m + s}{s} \end{aligned} \quad (1.4)$$

$$\begin{aligned} & > \text{multiplication_work} := \text{normal}(\text{multiplication_phase1_work} + \\ & \text{multiplication_phase2_work}); \\ & \text{multiplication_work} := 2 m n + 2 m s - 2 m - \frac{1}{2} n - \frac{1}{2} s + \frac{1}{2} \end{aligned} \quad (1.5)$$

$$\begin{aligned} & > \text{expand}((2*m - 1/2)*(n+s-1)); \\ & 2 m n + 2 m s - 2 m - \frac{1}{2} n - \frac{1}{2} s + \frac{1}{2} \end{aligned} \quad (1.6)$$

Span

$$\begin{aligned} & > \text{multiplication_phase1_span} := \text{multiplication_param} * \\ & (\text{multiplication_param} + \text{multiplication_param} - 1); \\ & \text{multiplication_phase1_span} := s (2 s - 1) \end{aligned} \quad (2.1)$$

$$\begin{aligned} & > \text{multiplication_phase2_span} := \text{multiplication_phase2_steps} * \\ & \text{multiplication_param}; \\ & \text{multiplication_phase2_span} := \frac{\ln\left(\frac{m}{s}\right) s}{\ln(2)} \end{aligned} \quad (2.2)$$

$$\begin{aligned} & > \text{multiplication_span} := \text{expand}(\text{simplify} \\ & (\text{multiplication_phase1_span} + \text{multiplication_phase2_span})); \\ & \text{multiplication_span} := 2 s^2 - s + \frac{\ln\left(\frac{m}{s}\right) s}{\ln(2)} \end{aligned} \quad (2.3)$$

Overhead

[s from b, l*s+s-1 from a

$s + l \cdot s + s - 1 \leq Z$

$$\begin{aligned} & > \text{multiplication_phase1_overheadperblock} := U * \\ & (\text{multiplication_param} + 1 + 1 + \text{multiplication_param}); \\ & \text{multiplication_phase1_overheadperblock} := U (2 s + 2) \end{aligned} \quad (3.1)$$

$$\begin{aligned} & > \text{multiplication_phase1_overhead} := \text{multiplication_phase1_blocks} * \\ & \text{multiplication_phase1_overheadperblock}; \\ & \text{multiplication_phase1_overhead} := \frac{(n + s - 1) m U (2 s + 2)}{s^2 l} \end{aligned} \quad (3.2)$$

$$\begin{aligned} & > \text{multiplication_phase2_overheadperblock} := U * 3 * \\ & \text{multiplication_param}; \\ & \text{multiplication_phase2_overheadperblock} := 3 U s \end{aligned} \quad (3.3)$$

```

> multiplication_phase2_overhead := simplify(sum
(multiplication_phase2_blocks(i),i=1..
multiplication_phase2_steps)*
multiplication_phase2_overheadperblock);
multiplication_phase2_overhead:=  $\frac{3(n+s-1)(-s+m)U}{s^2 l}$  (3.4)

```

```

> multiplication_overhead := normal
(multiplication_phase1_overhead+multiplication_phase2_overhead)
;
multiplication_overhead:=  $\frac{(n+s-1)U(5ms-3s^2+2m)}{s^2 l}$  (3.5)

```

▼ Tp

▼ Phase 1 N & L & C

```

> multiplication_phase1_workperthread := multiplication_param*
(multiplication_param+multiplication_param-1);
multiplication_phase1_workperthread:=  $s(2s-1)$  (4.1.1)

```

```

> multiplication_phase1_overheadperthread :=
multiplication_phase1_overheadperblock;
multiplication_phase1_overheadperthread:=  $U(2s+2)$  (4.1.2)

```

```

> multiplication_phase1_C :=
multiplication_phase1_workperthread+
multiplication_phase1_overheadperthread;
multiplication_phase1_C:=  $s(2s-1) + U(2s+2)$  (4.1.3)

```

```

> multiplication_phase1_N := multiplication_phase1_blocks;
multiplication_phase1_N:=  $\frac{(n+s-1)m}{s^2 l}$  (4.1.4)

```

```

> multiplication_phase1_L := 1;
multiplication_phase1_L:= 1 (4.1.5)

```

```

> Tp_phase_1 := simplify(eval((multiplication_phase1_N/P+
multiplication_phase1_L)*multiplication_phase1_C, [P=
multiplication_phase1_blocks]));
Tp_phase_1:=  $4Us + 4s^2 + 4U - 2s$  (4.1.6)

```

▼ Phase 2 N & L & C

```

> multiplication_phase2_workperthread := multiplication_param;
multiplication_phase2_workperthread:=  $s$  (4.2.1)

```

```

> multiplication_phase2_overheadperthread :=
multiplication_phase2_overheadperblock;
multiplication_phase2_overheadperthread:=  $3Us$  (4.2.2)

```

```

> multiplication_phase2_C :=

```

$$\begin{aligned} & \text{multiplication_phase2_workperthread} + \\ & \text{multiplication_phase2_overheadperthread;} \\ & \text{multiplication_phase2_C} := 3 Us + s \end{aligned} \quad (4.2.3)$$

$$\begin{aligned} & > \text{multiplication_phase2_N} := \text{simplify}(\text{sum} \\ & \quad (\text{multiplication_phase2_blocks}(i), i=1.. \\ & \quad \text{multiplication_phase2_steps})); \\ & \text{multiplication_phase2_N} := \frac{(n+s-1)(-s+m)}{s^2 l} \end{aligned} \quad (4.2.4)$$

$$\begin{aligned} & > \text{multiplication_phase2_L} := \text{multiplication_phase2_steps}; \\ & \text{multiplication_phase2_L} := \frac{\ln\left(\frac{m}{s}\right)}{\ln(2)} \end{aligned} \quad (4.2.5)$$

$$\begin{aligned} & > \text{Tp_phase_2} := \text{simplify}(\text{eval}((\text{multiplication_phase2_N}/P + \\ & \quad \text{multiplication_phase2_L}) * \text{multiplication_phase2_C}, [P = \\ & \quad \text{multiplication_phase2_blocks}(1)])); \\ & \text{Tp_phase_2} := \left(\frac{2(-s+m)}{m} + \frac{\ln\left(\frac{m}{s}\right)}{\ln(2)} \right) (3 Us + s) \end{aligned} \quad (4.2.6)$$

▼ Tp

$$\begin{aligned} & > \text{multiplication_N} := \text{simplify}(\text{multiplication_phase1_N} + \\ & \quad \text{multiplication_phase2_N}); \\ & \text{multiplication_N} := \frac{(n+s-1)(2m-s)}{s^2 l} \end{aligned} \quad (4.3.1)$$

$$\begin{aligned} & > \text{multiplication_L} := \text{multiplication_phase1_L} + \\ & \quad \text{multiplication_phase2_L}; \\ & \text{multiplication_L} := \frac{\ln\left(\frac{m}{s}\right)}{\ln(2)} + 1 \end{aligned} \quad (4.3.2)$$

$$\begin{aligned} & > \text{multiplication_C} := \text{multiplication_phase1_C}; \\ & \text{multiplication_C} := s(2s-1) + U(2s+2) \end{aligned} \quad (4.3.3)$$

$$\begin{aligned} & > \text{multiplication_Tp} := \text{simplify}(\text{eval}((\text{multiplication_N}/P + \\ & \quad \text{multiplication_L}) * \text{multiplication_C}, [P = \\ & \quad \text{multiplication_phase1_blocks}])); \\ & \text{multiplication_Tp} := \left(\frac{2m-s}{m} + \frac{\ln\left(\frac{m}{s}\right)}{\ln(2)} + 1 \right) (2Us + 2s^2 + 2U - s) \end{aligned} \quad (4.3.4)$$

▼ When s = 1

$$> \text{multiplication_work1} := \text{normal}(\text{eval}(\text{multiplication_work}, [s=1]))$$

) ;

$$\text{multiplication_work1} := 2 m n - \frac{1}{2} n \quad (5.1)$$

> **multiplication_span1 := expand(eval(multiplication_span, [s=1]))**
) ;

$$\text{multiplication_span1} := 1 + \frac{\ln(m)}{\ln(2)} \quad (5.2)$$

> **multiplication_overhead1 := normal(eval(multiplication_overhead, [s=1]))**;

$$\text{multiplication_overhead1} := \frac{n U (7 m - 3)}{l} \quad (5.3)$$

> **multiplication_Tp1 := normal(eval(multiplication_Tp, [s=1]))**;

$$\text{multiplication_Tp1} := \frac{(\ln(m) m + 3 m \ln(2) - \ln(2)) (4 U + 1)}{m \ln(2)} \quad (5.4)$$

> **work_ratio := normal(multiplication_work1 / multiplication_work)**;

$$\text{work_ratio} := \frac{n}{n + s - 1} \quad (9)$$

> **span_ratio := normal(multiplication_span1 / multiplication_span)**;

$$\text{span_ratio} := \frac{\ln(2) + \ln(m)}{s \left(2 \ln(2) s - \ln(2) + \ln\left(\frac{m}{s}\right) \right)} \quad (10)$$

> **overhead_ratio := normal(multiplication_overhead1 / multiplication_overhead)**;

$$\text{overhead_ratio} := \frac{n (7 m - 3) s^2}{(n + s - 1) (5 m s - 3 s^2 + 2 m)} \quad (11)$$

> **Tp_ratio := normal(multiplication_Tp1 / multiplication_Tp)**;

$$\text{Tp_ratio} := \frac{(\ln(m) m + 3 m \ln(2) - \ln(2)) (4 U + 1)}{\left(3 m \ln(2) - \ln(2) s + \ln\left(\frac{m}{s}\right) m \right) (2 U s + 2 s^2 + 2 U - s)} \quad (12)$$

$s = O(1)$ on the condition that $2 l s + 2 s$ must fit into the local memory

Polynomial Multiplication Algorithm

This document is to demonstrate the analysis of plain vs FFT-based polynomial multiplication algorithm

[Given two polynomials a and b over a finite field, with $\deg(a) = n - 1$, and $\deg(b) = m - 1$, where $n \geq m$, compute the product of $a \times b$

```
[ > restart;
  > terms_a := n;                                terms_a:= n                                (1)
```

```
  > terms_b := m;                                terms_b:= m                                (2)
```

[Plain multiplication

```
  > plain_param := s;                            plain_param:= s                            (3)
```

```
  > plain_threadspersblock := l;                 plain_threadspersblock:= l                 (4)
```

```
  > plain_length := terms_a + plain_param - 1;    plain_length:= n + s - 1                   (5)
```

```
  > plain_rows := terms_b/plain_param;           plain_rows:= m/s                           (6)
```

Phase 1 multiplication:

- The grid of thread blocks is 2-D. Principle: each thread block computes some of the coefficient products and some of the partial sums toward the polynomial product. Those partial sums are stored in an auxiliary array, which is later processed in Phase 2.
- One row of thread blocks contributes to $y = n + s - 1$ terms of the product of $a \times b$, while there are $x = \frac{m}{s}$ rows of blocks.
- If we use l threads per block, then we have $\frac{x \cdot y}{s \cdot l}$ thread-blocks.
- Each thread in each thread block performs s^2 coefficient products and $s \cdot (s - 1)$ additions.
- Each block needs s terms of b and $l \cdot s + s - 1$ terms of a . Then it computes $s \cdot l$ elements of the $x \times y$ matrix.

```
[ > plain_phase1_blocks := plain_length*plain_rows/
```

$$\begin{aligned}
& \text{(plain_threadspersblock*plain_param);} \\
& \text{plain_phase1_blocks} := \frac{(n+s-1) m}{s^2 l} \tag{7}
\end{aligned}$$

Phase 2 addition:

- The x rows of the auxiliary array M are added pairwise in $\log(x)$ parallel steps. At each step, the number of rows (to be added) is divided by two
- At a given parallel step, each thread reads s elements from one row and s from another row
- Therefore at the i -th parallel step (starting at $i=0$ and using l threads per block) one reads 2 times s times l coefficients per block
- The total amount of data (coefficients) to be added at the beginning at the i -th parallel is twice less than at the previous step, thus it is x times y divided by 2^i
- Therefore, at the i -th parallel step, using l threads per block, we need $\frac{x \cdot y}{s \cdot l \cdot 2^{i+1}}$ blocks and we have $\log_2 x$ parallel steps for the whole phase.
- Each thread to compute at most s addition needs s element of *one row* and s element from another row, then writes back s elements.

$$\begin{aligned}
& > \text{plain_phase2_blocks} := \text{proc}(i) \\
& \quad \text{return plain_length*plain_rows/(plain_param*} \\
& \quad \text{plain_threadspersblock*2}^i(i)); \\
& \text{end proc;} \\
& > \text{plain_phase2_steps} := \text{simplify}(\log[2](\text{plain_rows})); \\
& \text{plain_phase2_steps} := \frac{\ln\left(\frac{m}{s}\right)}{\ln(2)} \tag{8}
\end{aligned}$$

Work

$$\begin{aligned}
& > \text{plain_phase1_workpersblock} := \text{plain_threadspersblock*plain_param*} \\
& \quad (\text{plain_param+plain_param-1}); \\
& \text{plain_phase1_workpersblock} := l s (2 s - 1) \tag{1.1}
\end{aligned}$$

$$\begin{aligned}
& > \text{plain_phase1_work} := \text{plain_phase1_blocks*} \\
& \quad \text{plain_phase1_workpersblock}; \\
& \text{plain_phase1_work} := \frac{(n+s-1) m (2 s - 1)}{s} \tag{1.2}
\end{aligned}$$

$$\begin{aligned}
& > \text{plain_phase2_workpersblock} := \text{plain_threadspersblock*plain_param}; \\
& \text{plain_phase2_workpersblock} := l s \tag{1.3}
\end{aligned}$$

$$\begin{aligned}
& > \text{plain_phase2_work} := \text{simplify}(\text{sum}(\text{plain_phase2_blocks}(i)* \\
& \quad \text{plain_phase2_workpersblock}, i=1..\text{plain_phase2_steps})+
\end{aligned}$$

```
plain_length/2);
```

$$plain_phase2_work := \frac{1}{2} \frac{2mn + 2ms - ns - s^2 - 2m + s}{s} \quad (1.4)$$

```
> plain_work := normal(plain_phase1_work+plain_phase2_work);
```

$$plain_work := 2mn + 2ms - 2m - \frac{1}{2}n - \frac{1}{2}s + \frac{1}{2} \quad (1.5)$$

```
> expand((2*m- 1/2)*(n+s-1));
```

$$2mn + 2ms - 2m - \frac{1}{2}n - \frac{1}{2}s + \frac{1}{2} \quad (1.6)$$

Span

```
> plain_phase1_span := plain_param*(plain_param+plain_param-1);
plain_phase1_span := s(2s-1) \quad (2.1)
```

```
> plain_phase2_span := plain_phase2_steps*plain_param;
```

$$plain_phase2_span := \frac{\ln\left(\frac{m}{s}\right)s}{\ln(2)} \quad (2.2)$$

```
> plain_span := expand(simplify(plain_phase1_span+
plain_phase2_span));
```

$$plain_span := 2s^2 - s + \frac{\ln\left(\frac{m}{s}\right)s}{\ln(2)} \quad (2.3)$$

Overhead

```
[s from b, l*s+s-1 from a
```

```
# s + l*s + s - 1 ≤ Z
```

```
> plain_phase1_overheadperblock := U*(plain_param+1+1+
plain_param);
```

$$plain_phase1_overheadperblock := U(2s+2) \quad (3.1)$$

```
> plain_phase1_overhead := plain_phase1_blocks*
plain_phase1_overheadperblock;
```

$$plain_phase1_overhead := \frac{(n+s-1)mU(2s+2)}{s^2l} \quad (3.2)$$

```
> plain_phase2_overheadperblock := U*3*plain_param;
```

$$plain_phase2_overheadperblock := 3Us \quad (3.3)$$

```
> plain_phase2_overhead := simplify(sum(plain_phase2_blocks(i),i=
1..plain_phase2_steps)*plain_phase2_overheadperblock);
```

$$plain_phase2_overhead := \frac{3(n+s-1)(-s+m)U}{sl} \quad (3.4)$$

```
> plain_overhead := normal(plain_phase1_overhead+
plain_phase2_overhead);
```

$$plain_overhead := \frac{(n+s-1) U(5ms - 3s^2 + 2m)}{s^2 l} \quad (3.5)$$

▼ Tp

▼ Phase 1 N & L & C

$$\begin{aligned} &> \text{plain_phase1_workperthread} := \text{plain_param} * (\text{plain_param} + \text{plain_param} - 1); \\ &\quad \text{plain_phase1_workperthread} := s(2s - 1) \end{aligned} \quad (4.1.1)$$

$$\begin{aligned} &> \text{plain_phase1_overheadperthread} := \text{plain_phase1_overheadperblock}; \\ &\quad \text{plain_phase1_overheadperthread} := U(2s + 2) \end{aligned} \quad (4.1.2)$$

$$\begin{aligned} &> \text{plain_phase1_C} := \text{plain_phase1_workperthread} + \text{plain_phase1_overheadperthread}; \\ &\quad \text{plain_phase1_C} := s(2s - 1) + U(2s + 2) \end{aligned} \quad (4.1.3)$$

$$\begin{aligned} &> \text{plain_phase1_N} := \text{plain_phase1_blocks}; \\ &\quad \text{plain_phase1_N} := \frac{(n+s-1)m}{s^2 l} \end{aligned} \quad (4.1.4)$$

$$\begin{aligned} &> \text{plain_phase1_L} := 1; \\ &\quad \text{plain_phase1_L} := 1 \end{aligned} \quad (4.1.5)$$

$$\begin{aligned} &> \text{Tp_phase_1} := \text{simplify}(\text{eval}((\text{plain_phase1_N}/\text{P} + \text{plain_phase1_L}) * \text{plain_phase1_C}, [\text{P} = \text{plain_phase1_blocks}])); \\ &\quad \text{Tp_phase_1} := 4Us + 4s^2 + 4U - 2s \end{aligned} \quad (4.1.6)$$

▼ Phase 2 N & L & C

$$\begin{aligned} &> \text{plain_phase2_workperthread} := \text{plain_param}; \\ &\quad \text{plain_phase2_workperthread} := s \end{aligned} \quad (4.2.1)$$

$$\begin{aligned} &> \text{plain_phase2_overheadperthread} := \text{plain_phase2_overheadperblock}; \\ &\quad \text{plain_phase2_overheadperthread} := 3Us \end{aligned} \quad (4.2.2)$$

$$\begin{aligned} &> \text{plain_phase2_C} := \text{plain_phase2_workperthread} + \text{plain_phase2_overheadperthread}; \\ &\quad \text{plain_phase2_C} := 3Us + s \end{aligned} \quad (4.2.3)$$

$$\begin{aligned} &> \text{plain_phase2_N} := \text{simplify}(\text{sum}(\text{plain_phase2_blocks}(i), i=1.. \text{plain_phase2_steps})); \\ &\quad \text{plain_phase2_N} := \frac{(n+s-1)(-s+m)}{s^2 l} \end{aligned} \quad (4.2.4)$$

$$\begin{aligned} &> \text{plain_phase2_L} := \text{plain_phase2_steps}; \\ &\quad \text{plain_phase2_L} := \frac{\ln\left(\frac{m}{s}\right)}{\ln(2)} \end{aligned} \quad (4.2.5)$$

```
> Tp_phase_2:= simplify(eval((plain_phase2_N/P+plain_phase2_L)
*plain_phase2_C, [P=plain_phase2_blocks(1)]));
```

$$Tp_phase_2 := \left(\frac{2(-s+m)}{m} + \frac{\ln\left(\frac{m}{s}\right)}{\ln(2)} \right) (3Us + s) \quad (4.2.6)$$

▼ **Tp**

```
> plain_N := simplify(plain_phase1_N + plain_phase2_N);
```

$$plain_N := \frac{(n+s-1)(2m-s)}{s^2 l} \quad (4.3.1)$$

```
> plain_L := plain_phase1_L + plain_phase2_L;
```

$$plain_L := \frac{\ln\left(\frac{m}{s}\right)}{\ln(2)} + 1 \quad (4.3.2)$$

```
> plain_C := plain_phase1_C;
```

$$plain_C := s(2s-1) + U(2s+2) \quad (4.3.3)$$

```
> plain_Tp := simplify(eval((plain_N/P+plain_L)*plain_C, [P=
plain_phase1_blocks]));
```

$$plain_Tp := \left(\frac{2m-s}{m} + \frac{\ln\left(\frac{m}{s}\right)}{\ln(2)} + 1 \right) (2Us + 2s^2 + 2U - s) \quad (4.3.4)$$

▼ **When s = 4, m = n**

```
> plain_work1 := normal(eval(plain_work, [s=4, m=n]));
```

$$plain_work1 := 2n^2 + \frac{11}{2}n - \frac{3}{2} \quad (5.1)$$

```
> plain_span1 := expand(eval(plain_span, [s=4, m=n]));
```

$$plain_span1 := 20 + \frac{4 \ln(n)}{\ln(2)} \quad (5.2)$$

```
> plain_overhead1 := normal(eval(plain_overhead, [s=4, m=n]));
```

$$plain_overhead1 := \frac{1}{8} \frac{(n+3)U(11n-24)}{l} \quad (5.3)$$

```
> plain_K1 := eval(plain_phase1_blocks, [s=4, m=n]);
```

$$plain_K1 := \frac{1}{16} \frac{(n+3)n}{l} \quad (5.4)$$

```
> plain_Tp1 := simplify(eval(plain_Tp, [s=4, m=n]));
```

$$plain_Tp1 := \left(\frac{2(n-2)}{n} + \frac{-2 \ln(2) + \ln(n)}{\ln(2)} + 1 \right) (10U + 28) \quad (5.5)$$

FFT-based multiplication

Input $f, g \in \mathbb{R}[x]$ with degree less than $n = 2^k$ and
a primitive n -th root of unity $w \in \mathbb{R}$

compute $1, w, w^2, \dots, w^{n-1}$

$a := DFT_{-}\{w\}(f)$

$b := DFT_{-}\{w\}(g)$

$c := a b$

return $(DFT_{-}\{w\})^{-1}(c) = \frac{1}{n} DFT_{-}\{w^{-1}\}(c)$

Stockham FFT algorithm

$$DFT_{-}\{2^k\} = \prod_{i=0}^{k-1} (DFT_{-}\{2\} \times L_{-}\{2^{k-1}\}) (D_{-}\{2, 2^{k-i-1}\} \\ \times L_{-}\{2^i\}) (L_{-}\{2\}^{2^{k-i}} \times L_{-}\{2^i\})$$

For each fixed $0 \leq i < k$, there are three computational steps:

Phase 1: $x \rightarrow (L_{-}\{2\}^{2^{k-i}} \times L_{-}\{2^i\}) x$

Phase 2: $x \rightarrow (D_{-}\{2, 2^{k-i-1}\} \times L_{-}\{2^i\}) x$

Phase 3: $x \rightarrow (DFT_{-}\{2\} \times L_{-}\{2^{k-1}\}) x$

> FFT_threadspersblock := l;

FFT_threadspersblock := l

(6.1)

First compute phase 3 $DFT_{-}\{2\} \times L_{-}\{2^{k-1}\}$ once, a thread reading 2 data + writing back the same + executing 3 bit operations, 3 additions, 2 modular add operations (5 arithmetic operations)

Second, for each $0 \leq i < k$, compute phase 1, 2 & 3

In phase 1, if $2^i < l$, a thread reading 1 data and writing back the same + executing 14 bit operations, 10 additions, 1 multiplications.

If $2^i \geq l$, a thread reading 1 data and writing back the same + executing 11 bit operations, 8 additions.

In phase 2, a thread reading 1 data and writing back the same + executing 5 bit operations, 4 additions, 1 modular multiply operations (11 arithmetic operations)

Work

$$\begin{aligned} &> \text{FFT_phase1_blocks} := n / \text{FFT_threadsperblock}; \\ &\quad \text{FFT_phase1_blocks} := \frac{n}{l} \end{aligned} \quad (6.1.1)$$

$$\begin{aligned} &> \text{FFT_phase1_workperblock} := 25 * \text{FFT_threadsperblock}; \\ &\quad \text{FFT_phase1_workperblock} := 25 l \end{aligned} \quad (6.1.2)$$

$$\begin{aligned} &> \text{FFT_phase1_work} := \text{FFT_phase1_blocks} * \\ &\quad \text{FFT_phase1_workperblock}; \\ &\quad \text{FFT_phase1_work} := 25 n \end{aligned} \quad (6.1.3)$$

$$\begin{aligned} &> \text{FFT_phase2_blocks} := n / (2 * \text{FFT_threadsperblock}); \\ &\quad \text{FFT_phase2_blocks} := \frac{1}{2} \frac{n}{l} \end{aligned} \quad (6.1.4)$$

$$\begin{aligned} &> \text{FFT_phase2_workperblock} := 20 * \text{FFT_threadsperblock}; \\ &\quad \text{FFT_phase2_workperblock} := 20 l \end{aligned} \quad (6.1.5)$$

$$\begin{aligned} &> \text{FFT_phase2_work} := \text{FFT_phase2_blocks} * \\ &\quad \text{FFT_phase2_workperblock}; \\ &\quad \text{FFT_phase2_work} := 10 n \end{aligned} \quad (6.1.6)$$

$$\begin{aligned} &> \text{FFT_phase3_blocks} := n / (2 * \text{FFT_threadsperblock}); \\ &\quad \text{FFT_phase3_blocks} := \frac{1}{2} \frac{n}{l} \end{aligned} \quad (6.1.7)$$

$$\begin{aligned} &> \text{FFT_phase3_workperblock} := 16 * \text{FFT_threadsperblock}; \\ &\quad \text{FFT_phase3_workperblock} := 16 l \end{aligned} \quad (6.1.8)$$

$$\begin{aligned} &> \text{FFT_phase3_work} := \text{FFT_phase3_blocks} * \\ &\quad \text{FFT_phase3_workperblock}; \\ &\quad \text{FFT_phase3_work} := 8 n \end{aligned} \quad (6.1.9)$$

$$\begin{aligned} &> \text{FFT_work} := \text{normal}(\text{sum}(\text{FFT_phase1_work} + \text{FFT_phase2_work} + \\ &\quad \text{FFT_phase3_work}, i=0..\log[2](n)-2) + \text{FFT_phase3_work}); \\ &\quad \text{FFT_work} := \frac{n(-35 \ln(2) + 43 \ln(n))}{\ln(2)} \end{aligned} \quad (6.1.10)$$

Span

$$\begin{aligned} &> \text{FFT_phase1_span} := 25; \\ &\quad \text{FFT_phase1_span} := 25 \end{aligned} \quad (6.2.1)$$

$$\begin{aligned} &> \text{FFT_phase2_span} := 10; \\ &\quad \text{FFT_phase2_span} := 10 \end{aligned} \quad (6.2.2)$$

$$\begin{aligned} &> \text{FFT_phase3_span} := 8; \\ &\quad \text{FFT_phase3_span} := 8 \end{aligned} \quad (6.2.3)$$

$$\begin{aligned} &> \text{FFT_span} := \text{normal}(\text{sum}(\text{FFT_phase1_span} + \text{FFT_phase2_span} + \\ &\quad \text{FFT_phase3_span}, i=0..\log[2](n)-2) + \text{FFT_phase3_span}); \\ &\quad \text{FFT_span} := \frac{-35 \ln(2) + 43 \ln(n)}{\ln(2)} \end{aligned} \quad (6.2.4)$$

▼ Overhead

$$\begin{aligned} &> \text{FFT_phase1_overheadperblock} := 2*U; \\ &\quad \text{FFT_phase1_overheadperblock} := 2 U \end{aligned} \quad (6.3.1)$$

$$\begin{aligned} &> \text{FFT_phase1_overhead} := \text{FFT_phase1_blocks} * \\ &\quad \text{FFT_phase1_overheadperblock}; \\ &\quad \text{FFT_phase1_overhead} := \frac{2 n U}{l} \end{aligned} \quad (6.3.2)$$

$$\begin{aligned} &> \text{FFT_phase2_overheadperblock} := 2*U; \\ &\quad \text{FFT_phase2_overheadperblock} := 2 U \end{aligned} \quad (6.3.3)$$

$$\begin{aligned} &> \text{FFT_phase2_overhead} := \text{FFT_phase2_blocks} * \\ &\quad \text{FFT_phase2_overheadperblock}; \\ &\quad \text{FFT_phase2_overhead} := \frac{n U}{l} \end{aligned} \quad (6.3.4)$$

$$\begin{aligned} &> \text{FFT_phase3_overheadperblock} := 4*U; \\ &\quad \text{FFT_phase3_overheadperblock} := 4 U \end{aligned} \quad (6.3.5)$$

$$\begin{aligned} &> \text{FFT_phase3_overhead} := \text{FFT_phase3_blocks} * \\ &\quad \text{FFT_phase3_overheadperblock}; \\ &\quad \text{FFT_phase3_overhead} := \frac{2 n U}{l} \end{aligned} \quad (6.3.6)$$

$$\begin{aligned} &> \text{FFT_overhead} := \text{normal}(\text{sum}(\text{FFT_phase1_overhead} + \\ &\quad \text{FFT_phase2_overhead} + \text{FFT_phase3_overhead}, i=0..\log[2](n)-2) + \\ &\quad \text{FFT_phase3_overhead}); \\ &\quad \text{FFT_overhead} := \frac{n U (-3 \ln(2) + 5 \ln(n))}{\ln(2) l} \end{aligned} \quad (6.3.7)$$

▼ Tp

$$\begin{aligned} &> \text{FFT_N} := \text{normal}(\text{sum}(\text{FFT_phase1_blocks} + \text{FFT_phase2_blocks} + \\ &\quad \text{FFT_phase3_blocks}, i=0..\log[2](n)-2) + \text{FFT_phase3_blocks}); \\ &\quad \text{FFT_N} := \frac{1}{2} \frac{n (-3 \ln(2) + 4 \ln(n))}{\ln(2) l} \end{aligned} \quad (6.4.1)$$

$$\begin{aligned} &> \text{FFT_L} := \text{normal}(\text{sum}(3, i=0..\log[2](n)-2) + 1); \\ &\quad \text{FFT_L} := \frac{-2 \ln(2) + 3 \ln(n)}{\ln(2)} \end{aligned} \quad (6.4.2)$$

$$\begin{aligned} &> \text{FFT_C} := \text{FFT_phase1_span} + \text{FFT_phase3_overheadperblock}; \\ &\quad \text{FFT_C} := 25 + 4 U \end{aligned} \quad (6.4.3)$$

$$\begin{aligned} &> \text{FFT_Tp} := \text{normal}((\text{FFT_N}/\text{FFT_phase1_blocks} + \text{FFT_L}) * \text{FFT_C}); \\ &\quad \text{FFT_Tp} := \frac{1}{2} \frac{(-7 \ln(2) + 10 \ln(n)) (25 + 4 U)}{\ln(2)} \end{aligned} \quad (6.4.4)$$

▼ Point wise multiplication

$$\begin{aligned} &> \text{PWM_threads} := l; \\ &\qquad \text{PWM_threads} := l \end{aligned} \tag{7.1}$$

$$\begin{aligned} &> \text{PWM_blocks} := n / \text{PWM_threads}; \\ &\qquad \text{PWM_blocks} := \frac{n}{l} \end{aligned} \tag{7.2}$$

$$\begin{aligned} &> \text{PWM_workperblock} := 6 * \text{PWM_threads}; \\ &\qquad \text{PWM_workperblock} := 6 l \end{aligned} \tag{7.3}$$

$$\begin{aligned} &> \text{PWM_work} := \text{PWM_blocks} * \text{PWM_workperblock}; \\ &\qquad \text{PWM_work} := 6 n \end{aligned} \tag{7.4}$$

$$\begin{aligned} &> \text{PWM_span} := 6; \\ &\qquad \text{PWM_span} := 6 \end{aligned} \tag{7.5}$$

$$\begin{aligned} &> \text{PWM_overheadperblock} := 3 * U; \\ &\qquad \text{PWM_overheadperblock} := 3 U \end{aligned} \tag{7.6}$$

$$\begin{aligned} &> \text{PWM_overhead} := \text{PWM_blocks} * \text{PWM_overheadperblock}; \\ &\qquad \text{PWM_overhead} := \frac{3 n U}{l} \end{aligned} \tag{7.7}$$

$$\begin{aligned} &> \text{PWM_N} := \text{PWM_blocks}; \\ &\qquad \text{PWM_N} := \frac{n}{l} \end{aligned} \tag{7.8}$$

$$\begin{aligned} &> \text{PWM_L} := 1; \\ &\qquad \text{PWM_L} := 1 \end{aligned} \tag{7.9}$$

$$\begin{aligned} &> \text{PWM_C} := 6 + \text{PWM_overheadperblock}; \\ &\qquad \text{PWM_C} := 6 + 3 U \end{aligned} \tag{7.10}$$

▼ Scale the vector

$$\begin{aligned} &> \text{SV_threads} := l; \\ &\qquad \text{SV_threads} := l \end{aligned} \tag{8.1}$$

$$\begin{aligned} &> \text{SV_blocks} := n / \text{SV_threads}; \\ &\qquad \text{SV_blocks} := \frac{n}{l} \end{aligned} \tag{8.2}$$

$$\begin{aligned} &> \text{SV_workperblock} := 5 * \text{SV_threads}; \\ &\qquad \text{SV_workperblock} := 5 l \end{aligned} \tag{8.3}$$

$$\begin{aligned} &> \text{SV_work} := \text{SV_blocks} * \text{SV_workperblock}; \\ &\qquad \text{SV_work} := 5 n \end{aligned} \tag{8.4}$$

$$\begin{aligned} &> \text{SV_span} := 5; \\ &\qquad \text{SV_span} := 5 \end{aligned} \tag{8.5}$$

$$\begin{aligned} &> \text{SV_overheadperblock} := 2 * U; \\ &\qquad \text{SV_overheadperblock} := 2 U \end{aligned} \tag{8.6}$$

$$\begin{aligned} & \text{SV_overhead} := \text{SV_blocks} * \text{SV_overheadperblock}; \\ & \text{SV_overhead} := \frac{2 n U}{l} \end{aligned} \quad (8.7)$$

$$\begin{aligned} & \text{SV_N} := \text{SV_blocks}; \\ & \text{SV_N} := \frac{n}{l} \end{aligned} \quad (8.8)$$

$$\begin{aligned} & \text{SV_L} := 1; \\ & \text{SV_L} := 1 \end{aligned} \quad (8.9)$$

$$\begin{aligned} & \text{SV_C} := 5 + \text{SV_overheadperblock}; \\ & \text{SV_C} := 5 + 2 U \end{aligned} \quad (8.10)$$

▼ Work, Span & Overhead

$$\begin{aligned} & \text{FFTbased_work} := 3 * \text{FFT_work} + \text{PWM_work} + \text{SV_work}; \\ & \text{FFTbased_work} := \frac{3 n (-35 \ln(2) + 43 \ln(n))}{\ln(2)} + 11 n \end{aligned} \quad (9.1)$$

$$\begin{aligned} & \text{FFTbased_span} := 3 * \text{FFT_span} + \text{PWM_span} + \text{SV_span}; \\ & \text{FFTbased_span} := \frac{3 (-35 \ln(2) + 43 \ln(n))}{\ln(2)} + 11 \end{aligned} \quad (9.2)$$

$$\begin{aligned} & \text{FFTbased_overhead} := 3 * \text{FFT_overhead} + \text{PWM_overhead} + \text{SV_overhead}; \\ & \text{FFTbased_overhead} := \frac{3 n U (-3 \ln(2) + 5 \ln(n))}{\ln(2) l} + \frac{5 n U}{l} \end{aligned} \quad (9.3)$$

▼ Tp

$$\begin{aligned} & \text{FFTbased_K} := \text{FFT_phase1_blocks}; \\ & \text{FFTbased_K} := \frac{n}{l} \end{aligned} \quad (10.1)$$

$$\begin{aligned} & \text{FFTbased_N} := \text{normal}(3 * \text{FFT_N} + \text{PWM_N} + \text{SV_N}); \\ & \text{FFTbased_N} := \frac{1}{2} \frac{n (-5 \ln(2) + 12 \ln(n))}{\ln(2) l} \end{aligned} \quad (10.2)$$

$$\begin{aligned} & \text{FFTbased_L} := \text{normal}(3 * \text{FFT_L} + \text{PWM_L} + \text{SV_L}); \\ & \text{FFTbased_L} := \frac{-4 \ln(2) + 9 \ln(n)}{\ln(2)} \end{aligned} \quad (10.3)$$

$$\begin{aligned} & \text{FFTbased_C} := \text{FFT_C}; \\ & \text{FFTbased_C} := 25 + 4 U \end{aligned} \quad (10.4)$$

$$\text{FFTbased_Tp} := \text{normal}((\text{FFTbased_N} / \text{FFTbased_K} + \text{FFTbased_L}) * \text{FFTbased_C});$$

$$\left[\left[\begin{aligned} & \text{FFTbased_Tp} := \frac{1}{2} \frac{(-13 \ln(2) + 30 \ln(n)) (25 + 4 U)}{\ln(2)} \end{aligned} \right. \right. \quad (10.5)$$

Plain vs FFT-based

> **Rwork := normal(plain_work1 / FFTbased_work);**

$$Rwork := \frac{1}{2} \frac{(4 n^2 + 11 n - 3) \ln(2)}{n (-94 \ln(2) + 129 \ln(n))} \quad (9)$$

> **Rspan := normal(plain_span1 / FFTbased_span);**

$$Rspan := \frac{4 (5 \ln(2) + \ln(n))}{-94 \ln(2) + 129 \ln(n)} \quad (10)$$

> **Roverhead := normal(plain_overhead1 / FFTbased_overhead);**

$$Roverhead := \frac{1}{8} \frac{(n + 3) (11 n - 24) \ln(2)}{n (-4 \ln(2) + 15 \ln(n))} \quad (11)$$

> **Rt := normal(plain_Tp1 / FFTbased_Tp);**

$$Rt := \frac{4 (n \ln(n) + n \ln(2) - 4 \ln(2)) (5 U + 14)}{n (-13 \ln(2) + 30 \ln(n)) (25 + 4 U)} \quad (12)$$

> **plain_Tp1 := simplify(eval(eval((plain_N/P+plain_L)*plain_C, [P=FFTbased_K]), [s=4, m=n]));**

$$plain_Tp1 := \left(\frac{1}{8} \frac{(n + 3) (n - 2)}{n} + \frac{-2 \ln(2) + \ln(n)}{\ln(2)} + 1 \right) (10 U + 28) \quad (13)$$

> **Rt := normal(plain_Tp1 / FFTbased_Tp);**

$$Rt := \frac{1}{2} \frac{(\ln(2) n^2 + 8 n \ln(n) - 7 n \ln(2) - 6 \ln(2)) (5 U + 14)}{n (-13 \ln(2) + 30 \ln(n)) (25 + 4 U)} \quad (14)$$

Sorting

This document is to demonstrate the analysis of radix sort.

[> **restart;**

- For a fixed key size c , radix sort on input size n is computed by $\frac{c}{s}$ passes, and each pass sort based on s ([1..8]) bits.
- We denote l is the number of threads per block.

[> radix_input := n;	<i>radix_input := n</i>	(1)
[> radix_keysize := c;	<i>radix_keysize := c</i>	(2)
[> radix_threads := l;	<i>radix_threads := l</i>	(3)
[> radix_iterations := s;	<i>radix_iterations := s</i>	(4)
[> radix_blocks := radix_input / (4 * radix_threads);	<i>radix_blocks := $\frac{1}{4} \frac{n}{l}$</i>	(5)
[> radix_buckets := 2^radix_iterations;	<i>radix_buckets := 2^s</i>	(6)

▼ **Phase 1: Each block loads and sorts its tile using s iterations of 1-bit split, and write back its 2^s -entry digit histogram and sorted data**

- Each of *the* t threads deals with 4 elements, so that each thread block needs $4l$ elements to load, and writes back $4l$ sorted elements and 2^s entry elements.
- Each thread block computes s iterations (c bits), where $1 \leq s \leq c$
- In total, we have $\frac{n}{4l}$ thread blocks.

For each block dealing with one bit, it needs 5 steps to compute the correct position of each input element:

- # 1) Set a "1" in each "0" input;
- # 2) Scan the 1s (using prefix sum algorithm)

```

# 3) Compute the total false only once per block
# 4) Compute the position by 2 additions and 1 comparison
# 5) Scatter input to the correct position
# (Refer to http://cudpp.googlecode.com/svn/trunk/doc/CUDPP\_slides.pdf)

# To create a  $2^s$  bucket, it needs 3 steps
# 1) Set the bucket to 0
# 2) Each thread checks 5 elements and update the bucket at most 5 times
# 3) Each bucket updates its value to the number of elements inside the bucket

# Load  $4l$  elements and write back sorted  $4l$  elements and  $2^s$  values of the bucket

> radix_phase1_workperblock := normal(radix_iterations*(6*
radix_threads+8*sum(radix_threads/2^(i-1),i=1..log[2]
(radix_threads))+17)+12);


$$radix\_phase1\_workperblock := -32 \left( \frac{1}{2} \right)^{\frac{\ln(l) + \ln(2)}{\ln(2)}} ls + 22ls + 17s + 12 \quad (1.1)$$


> radix_phase1_overheadperblock := (4+4+1)*U;
radix_phase1_overheadperblock := 9 U \quad (1.2)

> radix_phase1_work := simplify(radix_blocks*
radix_phase1_workperblock);

$$radix\_phase1\_work := \frac{1}{4} \frac{n(22ls + s + 12)}{l} \quad (1.3)$$


> radix_phase1_span := radix_iterations*(4+(log[2](radix_threads)
+2)*3+(log[2](radix_threads)+2)*5+1+3*4+2*4)+5+5+1+1;

$$radix\_phase1\_span := s \left( 41 + \frac{8 \ln(l)}{\ln(2)} \right) + 12 \quad (1.4)$$


> radix_phase1_overhead := normal(radix_blocks*
radix_phase1_overheadperblock);

$$radix\_phase1\_overhead := \frac{9}{4} \frac{nU}{l} \quad (1.5)$$


```

Phase 2: Perform a prefix sum over the histogram table, stored in column-major order

- We consider 2^s buckets, each having $\frac{n}{4l}$, the results from phase 1, and perform a prefix sum over the histogram table to compute the

output positions.

- Further reference: Multiscan - <http://www.moderngpu.com/intro/scan.html>

$$\begin{aligned} &> \text{radix_phase2_elements} := \text{radix_blocks} * \text{radix_buckets}; \\ &\quad \text{radix_phase2_elements} := \frac{1}{4} \frac{n 2^s}{l} \end{aligned} \quad (2.1)$$

$$\begin{aligned} &> \text{radix_phase2_blocks} := \text{radix_phase2_elements} / (4 * \text{radix_threads}); \\ &\quad \text{radix_phase2_blocks} := \frac{1}{16} \frac{n 2^s}{l^2} \end{aligned} \quad (2.2)$$

Regard to the prefix sum, one block performs the up-sweep phase, with 3 instructions, and then performs the down-sweep phase, with 5 instructions, regard to Listing 2 in scan.pdf. (Except mediate step)

$$\begin{aligned} &> \text{radix_phase2_inclusive_workperblock} := \text{normal}(\text{simplify}(8 * (\text{sum} \\ &\quad (\text{radix_threads} / 2^{(i-1)}, i=1.. \log[2](\text{radix_threads}) + 2) + 1)); \\ &\quad \text{radix_phase2_inclusive_workperblock} := 1 + 16 l \end{aligned} \quad (2.3)$$

$$\begin{aligned} &> \text{radix_phase2_inclusive_overheadperblock} := (4+4)*U; \\ &\quad \text{radix_phase2_inclusive_overheadperblock} := 8 U \end{aligned} \quad (2.4)$$

$$\begin{aligned} &> \text{radix_phase2_inclusive_work} := \text{radix_phase2_blocks} * \\ &\quad \text{radix_phase2_inclusive_workperblock}; \\ &\quad \text{radix_phase2_inclusive_work} := \frac{1}{16} \frac{n 2^s (1 + 16 l)}{l^2} \end{aligned} \quad (2.5)$$

$$\begin{aligned} &> \text{radix_phase2_inclusive_span} := \text{simplify}(8 * (\log[2] \\ &\quad (\text{radix_threads}) + 2) + 1); \\ &\quad \text{radix_phase2_inclusive_span} := \frac{8 \ln(l)}{\ln(2)} + 17 \end{aligned} \quad (2.6)$$

$$\begin{aligned} &> \text{radix_phase2_inclusive_overhead} := \text{radix_phase2_blocks} * \\ &\quad \text{radix_phase2_inclusive_overheadperblock}; \\ &\quad \text{radix_phase2_inclusive_overhead} := \frac{1}{2} \frac{n 2^s U}{l^2} \end{aligned} \quad (2.7)$$

Store Block Sum to Auxiliary Array, here assume the number of blocks can fit in one shared memory, say $\frac{1}{16} \frac{n 2^s}{l^2} \leq Z$.

$$\begin{aligned} &> \text{radix_phase2_mediate_overheadperblock} := (4+4)*U; \\ &\quad \text{radix_phase2_mediate_overheadperblock} := 8 U \end{aligned} \quad (2.8)$$

$$\begin{aligned} &> \text{radix_phase2_mediate_work} := \text{normal}(\text{simplify}(8 * (\text{sum} \\ &\quad (\text{radix_threads} / 2^{(i-1)}, i=1.. \log[2](\text{radix_threads}) + 2) + 1)); \\ &\quad \text{radix_phase2_mediate_work} := 1 + 16 l \end{aligned} \quad (2.9)$$

$$\begin{aligned} &> \text{radix_phase2_mediate_span} := 8 * (\log[2](\text{radix_threads}) + 2) + 1; \end{aligned} \quad (2.10)$$

$$radix_phase2_mediate_span := \frac{8 \ln(l)}{\ln(2)} + 17 \quad (2.10)$$

$$\begin{aligned} &> radix_phase2_mediate_overhead := \\ &radix_phase2_mediate_overheadperblock; \\ &radix_phase2_mediate_overhead := 8 U \end{aligned} \quad (2.11)$$

Scan Block Sums

$$\begin{aligned} &> radix_phase2_exclusive_workperblock := 4 * radix_threads; \\ &radix_phase2_exclusive_workperblock := 4 l \end{aligned} \quad (2.12)$$

$$\begin{aligned} &> radix_phase2_exclusive_overheadperblock := (4+1+4)*U; \\ &radix_phase2_exclusive_overheadperblock := 9 U \end{aligned} \quad (2.13)$$

$$\begin{aligned} &> radix_phase2_exclusive_work := (radix_phase2_blocks-1)* \\ &radix_phase2_exclusive_workperblock; \\ &radix_phase2_exclusive_work := 4 \left(\frac{1}{16} \frac{n 2^s}{l^2} - 1 \right) l \end{aligned} \quad (2.14)$$

$$\begin{aligned} &> radix_phase2_exclusive_span := 4; \\ &radix_phase2_exclusive_span := 4 \end{aligned} \quad (2.15)$$

$$\begin{aligned} &> radix_phase2_exclusive_overhead := (radix_phase2_blocks-1)* \\ &radix_phase2_exclusive_overheadperblock; \\ &radix_phase2_exclusive_overhead := 9 \left(\frac{1}{16} \frac{n 2^s}{l^2} - 1 \right) U \end{aligned} \quad (2.16)$$

Total work & span in phase 2

$$\begin{aligned} &> radix_phase2_work := normal(radix_phase2_inclusive_work+ \\ &radix_phase2_mediate_work+radix_phase2_exclusive_work); \\ &radix_phase2_work := \frac{1}{16} \frac{20 n 2^s l + 192 l^3 + n 2^s + 16 l^2}{l^2} \end{aligned} \quad (2.17)$$

$$\begin{aligned} &> radix_phase2_span := simplify(radix_phase2_inclusive_span+ \\ &radix_phase2_mediate_span+radix_phase2_exclusive_span); \\ &radix_phase2_span := \frac{16 \ln(l)}{\ln(2)} + 38 \end{aligned} \quad (2.18)$$

$$\begin{aligned} &> radix_phase2_overhead := normal \\ &(radix_phase2_inclusive_overhead+radix_phase2_mediate_overhead+ \\ &radix_phase2_exclusive_overhead); \\ &radix_phase2_overhead := \frac{1}{16} \frac{U(17 n 2^s - 16 l^2)}{l^2} \end{aligned} \quad (2.19)$$

▼ Phase 3: Each block copies its elements to the correct output position

- We have $\frac{n}{4 l}$ blocks, and each block copies $4 l$ elements to the correct output position

Each thread needs the update bucket to compute the position of the first element

$$\begin{aligned} &> \text{radix_phase3_workperblock} := 4 * \text{radix_threads}; \\ &\quad \text{radix_phase3_workperblock} := 4 \, l \end{aligned} \quad (3.1)$$

$$\begin{aligned} &> \text{radix_phase3_overheadperblock} := (4+1+4) * U; \\ &\quad \text{radix_phase3_overheadperblock} := 9 \, U \end{aligned} \quad (3.2)$$

$$\begin{aligned} &> \text{radix_phase3_work} := \text{radix_blocks} * \text{radix_phase3_workperblock}; \\ &\quad \text{radix_phase3_work} := n \end{aligned} \quad (3.3)$$

$$\begin{aligned} &> \text{radix_phase3_span} := 4; \\ &\quad \text{radix_phase3_span} := 4 \end{aligned} \quad (3.4)$$

$$\begin{aligned} &> \text{radix_phase3_overhead} := \text{radix_blocks} * \\ &\quad \text{radix_phase3_overheadperblock}; \\ &\quad \text{radix_phase3_overhead} := \frac{9}{4} \frac{n \, U}{l} \end{aligned} \quad (3.5)$$

Work & Span & Overhead & N & L & C

$$\begin{aligned} &> \text{radix_work} := \text{collect}(\text{radix_keysize}/\text{radix_iterations} * \\ &\quad (\text{radix_phase1_work} + \text{radix_phase2_work} + \text{radix_phase3_work}), n); \\ &\quad \text{radix_work} := \frac{c \left(\frac{1}{4} \frac{22 \, l \, s + s + 12}{l} + \frac{1}{16} \frac{20 \, 2^s \, l + 2^s}{l^2} + 1 \right) n}{s} \end{aligned} \quad (4.1)$$

$$+ \frac{1}{16} \frac{c (192 \, l^3 + 16 \, l^2)}{s \, l^2}$$

$$\begin{aligned} &> \text{radix_span} := \text{normal}(\text{radix_keysize}/\text{radix_iterations} * \\ &\quad (\text{radix_phase1_span} + \text{radix_phase2_span} + \text{radix_phase3_span})); \\ &\quad \text{radix_span} := \frac{c (8 \, s \ln(l) + 41 \, s \ln(2) + 16 \ln(l) + 54 \ln(2))}{s \ln(2)} \end{aligned} \quad (4.2)$$

$$\begin{aligned} &> \text{radix_overhead} := \text{normal}(\text{radix_keysize}/\text{radix_iterations} * \\ &\quad (\text{radix_phase1_overhead} + \text{radix_phase2_overhead} + \\ &\quad \text{radix_phase3_overhead})); \\ &\quad \text{radix_overhead} := \frac{1}{16} \frac{c \, U (17 \, n \, 2^s - 16 \, l^2 + 72 \, n \, l)}{s \, l^2} \end{aligned} \quad (4.3)$$

$$\begin{aligned} &> \text{radix_N} := \text{radix_keysize}/\text{radix_iterations} * (\text{radix_blocks} + 2 * \\ &\quad \text{radix_phase2_blocks} + \text{radix_blocks}); \\ &\quad \text{radix_N} := \frac{c \left(\frac{1}{2} \frac{n}{l} + \frac{1}{8} \frac{n \, 2^s}{l^2} \right)}{s} \end{aligned} \quad (4.4)$$

$$\begin{aligned} &> \text{radix_L} := \text{radix_keysize}/\text{radix_iterations} * (1 + 3 + 1); \\ &\quad \text{radix_L} := \frac{c \, U}{s} \end{aligned} \quad (4.5)$$

$$radix_L := \frac{5c}{s} \quad (4.5)$$

$$\begin{aligned} &> \text{radix_C} := \text{radix_phase1_span} + 9*U; \\ &\quad radix_C := s \left(41 + \frac{8 \ln(l)}{\ln(2)} \right) + 12 + 9U \end{aligned} \quad (4.6)$$

$$\begin{aligned} &> \text{radix_Tp} := \text{simplify}((\text{radix_N}/\text{radix_blocks} + \text{radix_L}) * \text{radix_C}); \\ &\quad radix_Tp := \frac{1}{2} \frac{c(2^s + 14l)(8s \ln(l) + 9U \ln(2) + 41s \ln(2) + 12 \ln(2))}{l s \ln(2)} \end{aligned} \quad (4.7)$$

▼ When $s = 1$

$$\begin{aligned} &> \text{radix_work_1} := \text{simplify}(\text{eval}(\text{radix_work}, [s=1])); \\ &\quad radix_work_1 := \frac{1}{8} \frac{c(96l^3 + 52l^2n + 8l^2 + 46ln + n)}{l^2} \end{aligned} \quad (5.1)$$

$$\begin{aligned} &> \text{radix_overhead_1} := \text{normal}(\text{eval}(\text{radix_overhead}, [s=1])); \\ &\quad radix_overhead_1 := -\frac{1}{8} \frac{cU(8l^2 - 36ln - 17n)}{l^2} \end{aligned} \quad (5.2)$$

$$\begin{aligned} &> \text{radix_span_1} := \text{normal}(\text{eval}(\text{radix_span}, [s=1])); \\ &\quad radix_span_1 := \frac{c(24 \ln(l) + 95 \ln(2))}{\ln(2)} \end{aligned} \quad (5.3)$$

$$\begin{aligned} &> \text{radix_Tp_1} := \text{simplify}(\text{eval}(\text{radix_Tp}, [s=1])); \\ &\quad radix_Tp_1 := \frac{c(1 + 7l)(8 \ln(l) + 9U \ln(2) + 53 \ln(2))}{l \ln(2)} \end{aligned} \quad (5.4)$$

▼ Comparison

$$[8l + 2^s \leq Z$$

$$\begin{aligned} &> \text{radix_work_ratio} := \text{simplify}(\text{radix_work_1}/\text{radix_work}); \\ &\quad radix_work_ratio := \frac{2(96l^3 + 52l^2n + 8l^2 + 46ln + n)s}{88sl^2n + 20n2^sl + 192l^3 + 16l^2n + 4lsn + n2^s + 16l^2 + 48nl} \end{aligned} \quad (6.1)$$

$$\begin{aligned} &> \text{radix_overhead_ratio} := \text{collect} \\ &\quad (\text{radix_overhead_1}/\text{radix_overhead}, n); \\ &\quad radix_overhead_ratio := -\frac{2((-36l - 17)n + 8l^2)s}{(172^s + 72l)n - 16l^2} \end{aligned} \quad (6.2)$$

$$\begin{aligned} &> \text{radix_span_ratio} := \text{simplify}(\text{radix_span_1}/\text{radix_span}); \\ &\quad radix_span_ratio := \frac{(24 \ln(l) + 95 \ln(2))s}{8s \ln(l) + 41s \ln(2) + 16 \ln(l) + 54 \ln(2)} \end{aligned} \quad (6.3)$$

Assume $2^s = O(l)$, we can reduce overhead by s

$$\begin{aligned} &> \text{radix_Tp_R} := \text{simplify}(\text{radix_Tp_1}/\text{radix_Tp}); \\ &\text{radix_Tp_R} := \frac{2 (1 + 7 l) (8 \ln(l) + 9 U \ln(2) + 53 \ln(2)) s}{(2^s + 14 l) (8 s \ln(l) + 9 U \ln(2) + 41 s \ln(2) + 12 \ln(2))} \end{aligned} \quad (6.4)$$

$$\begin{aligned} &> \text{simplify}(\text{eval}(\text{radix_Tp_R}, [\text{s}=\log[2](l)])); \\ &\frac{2}{15} \frac{(1 + 7 l) (8 \ln(l) + 9 U \ln(2) + 53 \ln(2)) \ln(l)}{l (9 U \ln(2)^2 + 8 \ln(l)^2 + 41 \ln(l) \ln(2) + 12 \ln(2)^2)} \end{aligned} \quad (6.5)$$

$$\begin{aligned} &> \text{lcN} := \text{collect}(\text{expand}(2*(1+7*l)*(8*\ln(l)+9*U+53)*\ln(l)), l); \\ &\text{lcN} := (112 \ln(l)^2 + 126 \ln(l) U + 742 \ln(l)) l + 16 \ln(l)^2 + 18 \ln(l) U \\ &\quad + 106 \ln(l) \end{aligned} \quad (6.6)$$

$$\begin{aligned} &> \text{lcD} := 15*l*(9*U+8*\ln(l)^2+41*\ln(l)+12); \\ &\text{lcD} := 15 l (9 U + 8 \ln(l)^2 + 41 \ln(l) + 12) \end{aligned} \quad (6.7)$$

$$\begin{aligned} &> \text{lcN} := 112*\ln(l)^2+126*\ln(l)*U+742*\ln(l); \\ &\text{lcN} := 112 \ln(l)^2 + 126 \ln(l) U + 742 \ln(l) \end{aligned} \quad (6.8)$$

$$\begin{aligned} &> \text{lcD} := \text{expand}(15*(9*U+8*\ln(l)^2+41*\ln(l)+12)); \\ &\text{lcD} := 135 U + 120 \ln(l)^2 + 615 \ln(l) + 180 \end{aligned} \quad (6.9)$$

$$\begin{aligned} &> \text{simplify}(\text{lcN}/\text{lcD}); \\ &\frac{14}{15} \frac{\ln(l) (8 \ln(l) + 9 U + 53)}{9 U + 8 \ln(l)^2 + 41 \ln(l) + 12} \end{aligned} \quad (6.10)$$

$$\begin{aligned} &> \text{expand}(14*(8*\ln(l)+9*U+53)); \\ &112 \ln(l) + 126 U + 742 \end{aligned} \quad (6.11)$$

$$\begin{aligned} &> \text{expand}(15*(8*\ln(l)+41)); \\ &120 \ln(l) + 615 \end{aligned} \quad (6.12)$$

Appendix C

Documentation for MetaFork-to-CUDA Code Generator

Our MetaFork-to-CUDA code generator is based on the version 0.04 of PPCG¹. The code generation follows two algorithms, depending on whether one intends to use the shared memory or not in the kernel code. In the sequel, we refer to these algorithms as the *shared memory* and *global memory* modes of the code generator. These two algorithms are available to the user as two different targets of the Makefile for compiling our code generator. By default, the compilation of MetaFork programs, with our extended version of PPCG, uses the global memory only. This mode is compiled by running `make` at the root of the source tree of PPCG. To enable the use of the shared memory, one should compile the code generator by issuing `make mem=mlocal`.

In this documentation, we report the assumptions within our MetaFork-to-CUDA code generator in Appendix C.1, and the schedule tree used as a mathematical representation for MetaFork and parametric CUDA code in Appendix C.2.

C.1 Assumptions on the syntax of MetaFork statements

In order to use MetaFork-to-CUDA code generator without post-processing, we make assumptions on the syntax of MetaFork statements as below.

```
meta_schedule { ... }
```

This statement indicates its body will be launched to hardware accelerators, i.e. NVIDIA GPUs. It also transfers the data from CPU to GPU before launching kernels and transfers the data back from GPU to CPU after executing kernels. Furthermore, data transfer between CPU and GPU is automatically detected by this statement.

Statements supported within the body of the `meta_schedule` statement are a sequence of nested for loops. Thus, each nested for loops consist of `parallel for` loops (only 2 or 4) and/or serial `for` loops, and will be translated into a kernel call. In the case of `parallel for` loops, it is identified with the ‘`meta_for`’ keyword.

¹PPCG’s original code is available at <https://www.openhub.net/p/ppcg>.

```

    meta_for (initialize; condition; increment) { ... }
initialize: 0
condition: < (a variable)
increment: ++ or += 1
For instance, meta_for (int i = 0; i < upper_bound; i++) { ... }

```

The upper bound in the `condition` indicates either the number of threads per thread-block or the number of thread-blocks per grid. Thus, for launching a one-dimension kernel, it requires one outer `meta_for` loop specifying the grid size and one inner `meta_for` loop specifying the thread-block size. For a two-dimension kernel, two (immediately) consecutive outer `meta_for` loops are used to specify the grid sizes, and two (immediately) consecutive inner `meta_for` loops are used to specify the thread-block sizes. Moreover, the iterators in the first and second outer (resp. inner) `meta_for` loops correspond to `blockIdx.y` and `blockIdx.x` (resp. `threadIdx.y` and `threadIdx.x`), respectively, in the generated kernel code. Figure C.1 shows an example of a `meta_schedule` statement with a 1-D kernel and a 2-D kernel.

```

meta_schedule {
    // only for loops are supported here
    meta_for (int i = 0; i < gridDim.x; i++)
        // only for loops are supported here
        meta_for (int j = 0; j < blockDim.x; j++) {
            ... // nested for-loop body
        }

    // only for loops are supported here
    meta_for (int u = 0; u < gridDim.y; u++)
        meta_for (int i = 0; i < gridDim.x; i++)
            // only for loops are supported here
            meta_for (int v = 0; v < blockDim.y; v++)
                meta_for (int j = 0; j < blockDim.x; j++) {
                    ... // nested for-loop body
                }
}

```

Figure C.1: An example of the `meta_schedule` statement

```

for (initialize; condition; increment) { ... }

```

For those serial for loops, the upper or lower bound of `condition` should be a linear expression and `increment` should be increased or decreased by a constant. This observation is based on the work reported in [116].

```

array[expression] or array[expression][expression]

```

In the global memory mode, one can only use linear expressions as the indices (`expression`) of 1D or 2D arrays. However, one can hide non-linear expressions by using a separate statement, which yields a linear expression that can be used as the indices. For instance, `int p = i * B + j * s + k; array[p] = ...`. Note that non-linear expressions cannot be analyzed by PPCG, such that whether `array[p]` is reused or coalesced accessed is unknown to PPCG.

In the shared memory mode, due to the lack of analysis of non-linear expressions, we make the following assumptions.

1. If `expression` is a linear expression, all its variables must refer to the variable counters in the serial `for` loops of the current loop nest. In this case, we rely on PPCG to analyze the access patterns of the corresponding array.
2. If `expression` contains one and only one non-linear term, that is, one variable multiplying by another, say `i * B`, then one variable must refer to the current thread-block index and the other must refer to the corresponding thread-block size. One shall hide this non-linear expression by using a separate statement, while this statement should add a third variable referring to the thread index. For instance, `int p = i * B + j; array[p] = ...`. Furthermore, adding a constant to the above format of the index of `array[]` is supported, say `array[p] = array[p+1]`. For this format, we extend the analysis calculating the total amount of required shared memory per thread-block. Particularly, for a 2D array, the first (resp. second) `expression` refers to the first (resp. second) dimension of grids and thread-blocks, defined by the first (resp. second) outer and inner `meta_for` loops, respectively.
3. No other forms of non-linear expressions are accepted in `expression`.

Moreover, in the shared memory mode, not all arrays occurring in the MetaFork source code will necessarily use the shared memory in the generated CUDA code. In fact, in the addition to the syntax constraints described before, one of the following conditions must also hold:

1. If `array[]` (resp. `array[][]`) is written more than once and threads access it in a coalesced fashion, then a shared memory counterpart of `array[]` (resp. `array[][]`) will be generated.
2. If `array[]` (resp. `array[][]`) is read and threads access it in a coalesced fashion, then a shared memory counterpart of `array[]` (resp. `array[][]`) will be generated.

When none of those conditions is satisfied, then shared memory counterpart of `array[]` (resp. `array[][]`) is not generated.

C.2 Schedule tree for MetaFork and parametric CUDA code

Sven Verdoolaege, et al [117] proposed to explicitly represent schedules in the polyhedral model as a tree structure. We take advantage of this structure with minor modifications, such that it can represent the execution order of a MetaFork program as well as be converted to the AST for parametric CUDA kernel code generation.

For each schedule tree, there are three catalogues of node types, including core, external and convenience nodes. For core nodes, a *band* node indicates multi-dimensional piecewise quasi-affine partial schedule, a *filter* node selects statement instances that are executed by descendants, a *sequence* node defines its children executed in a given order, and a *set* node has its

children executed in an arbitrary order. For those external node types, a *domain* node includes a set of statement instances to be scheduled, and a *context* node specifies external constraints on symbolic constants. For convenience node types, one can attach additional information to subtrees as a *mark* node.

```
int ub_v = N / B;
meta_schedule {
  meta_for (int v = 0; v < ub_v; v++)
    meta_for (int u = 0; u < B; u++) {
      int inoffset = v * B + u;          // S_1
      int outoffset = N - 1 - inoffset;  // S_4
      Out[outoffset] = In[inoffset];     // S_6
    }
}
```

Consider the above MetaFork code for one-dimensional array reversal. Within the nested for loops, there are three statements referred as S_1, S_4, S_6, respectively, in the schedule tree below. This schedule tree is initialized based on the original MetaFork code. Two *schedule* nodes correspond to two *meta_for* loops, respectively, while we set “permutable” and “coincident” of each of these two schedule nodes to be 1s. Our goal is to extend this schedule tree so as to obtain another “rich” schedule tree for its corresponding CUDA program.

```
domain: "[N, ub_v, B] -> { S_6[v, u] : B >= 0 and v >= 0 and v <= -1 + ub_v and
u >= 0; S_1[v, u] : B >= 0 and v >= 0 and v <= -1 + ub_v and u >= 0;
S_4[v, u] : B >= 0 and v >= 0 and v <= -1 + ub_v and u >= 0 }"
child:
  schedule: "[N, ub_v, B] -> L_0[{ S_6[v, u] -> [(v)]; S_4[v, u] -> [(v)];
S_1[v, u] -> [(v)] }]"
  permutable: 1
  coincident: [ 1 ]
  child:
    schedule: "[N, ub_v, B] -> L_1[{ S_6[v, u] -> [(u)]; S_4[v, u] -> [(u)];
S_1[v, u] -> [(u)] }]"
    permutable: 1
    coincident: [ 1 ]
    child:
      sequence:
        - filter: "[N, ub_v, B] -> { S_1[v, u] }"
        - filter: "[N, ub_v, B] -> { S_4[v, u] }"
        - filter: "[N, ub_v, B] -> { S_6[v, u] }"
```

The schedule tree shown below is extended from the above schedule tree for generating both the host and device codes based on our MetaFork code. We first insert a mark node, named “kernel,” as on Line 16 below that indicates where the kernel call starts. Then, for each thread-block, we refer variable `b0` to the thread-block index `blockIdx.x` for a one-dimensional kernel, while for a two-dimensional kernel, we refer variable `b0` (resp. `b1`) to `blockIdx.y` (resp. `blockIdx.x`), which corresponds to the first (resp. second) outer *meta_for* loop. Due to

the fact that the number of thread-blocks per grid depends on the problem size, which is an arbitrary number, but that the maximum number of thread-blocks supported by *streaming multiprocessors* (SMs) is a fixed number, we shall generate a serial loop for each thread-block to execute in sequence, in case the number of required thread-blocks exceeds the hardware limit. Thus, we insert a filter node as on Line 21 of the schedule tree below, which means that there exists an instance for each thread-block starting from b_0 and incrementing by 32768 (that is, the hardware limit).

```

1  domain: "[N, ub_v, B] -> { kernel0[] : ub_v >= 1 and B >= 0 }"
2  child:
3    context: "[N, ub_v, B] -> { [] : N <= 2147483647 and N >= 0 and
4    ub_v <= 2147483647 and ub_v >= 1 and B <= 2147483647 and B >= 0 }"
5    child:
6      guard: "[N, ub_v, B] -> { [] : ub_v >= 1 and B >= 0 and N <= 2147483647
7      and N >= 0 and ub_v <= 2147483647 and B <= 2147483647 }"
8      child:
9        contraction: "[N, ub_v, B] -> { S_6[v, u] -> kernel0[];
10        S_1[v, u] -> kernel0[]; S_4[v, u] -> kernel0[] }"
11        expansion: "[N, ub_v, B] -> { kernel0[] -> S_6[v, u] : B >= 0 and
12        v >= 0 and v <= -1 + ub_v and u >= 0;
13        kernel0[] -> S_1[v, u] : B >= 0 and v >= 0 and v <= -1 + ub_v and u >= 0;
14        kernel0[] -> S_4[v, u] : B >= 0 and v >= 0 and v <= -1 + ub_v and u >= 0 }"
15        child:
16          mark: "kernel"
17          child:
18            context: "[N, ub_v, B, b0, t0] -> { [] : b0 <= -1 + ub_v and
19            b0 <= 32767 and b0 >= 0 and t0 >= 0 and t0 <= 511 }"
20            child:
21              filter: "[N, ub_v, B, b0] -> { S_1[v, u] : exists
22              (e0 = floor((-b0 + v)/32768): 32768e0 = -b0 + v and b0 >= 0
23              and b0 <= 32767); S_4[v, u] : exists (e0 = floor((-b0 + v)/32768):
24              32768e0 = -b0 + v and b0 >= 0 and b0 <= 32767);
25              S_6[v, u] : exists (e0 = floor((-b0 + v)/32768): 32768e0 = -b0 + v
26              and b0 >= 0 and b0 <= 32767) }"
27              child:
28                schedule: "[N, ub_v, B] -> L_0[{ S_6[v, u] -> [(v)];
29                S_4[v, u] -> [(v)]; S_1[v, u] -> [(v)] }]"
30                permutable: 1
31                coincident: [ 1 ]
32                child:
33                  extension: "[N, ub_v, B, b0] -> { [i0] -> sync0[];
34                  [i0] -> read[[i0] -> In[o1]] : o1 >= B + 1024i0 and
35                  o1 <= 1023 + B + 1024i0 and N >= 1 and ub_v >= 1 and B >= 0
36                  and o1 <= -1 + N and o1 >= 0; [i0] -> sync1[] }"
37                  child:
38                    sequence:
39                      - filter: "[N, ub_v, B, b0, t0] -> { read[[i0] ->
40                      In[B + t0 + 1024i0]] }"

```

```

41         child:
42             schedule: "[N, ub_v, B, b0] -> shared_In[{ read[[i0] ->
43                 In[i1]] -> [(-B - 1024i0 + i1)] }]"
44         - filter: "{ sync1[] }"
45         - filter: "[N, ub_v, B, b0] -> { S_1[v, u]; S_4[v, u];
46             S_6[v, u] }"
47         child:
48             filter: "[N, ub_v, B, t0] -> { S_4[v, t0]; S_6[v, t0];
49                 S_1[v, t0] }"
50         child:
51             schedule: "[N, ub_v, B] -> L_1[{ S_6[v, u] -> [(u)];
52                 S_4[v, u] -> [(u)]; S_1[v, u] -> [(u)] }]"
53             permutable: 1
54             coincident: [ 1 ]
55         child:
56             sequence:
57                 - filter: "[N, ub_v, B] -> { S_1[v, u] }"
58                 - filter: "[N, ub_v, B] -> { S_4[v, u] }"
59                 - filter: "[N, ub_v, B] -> { S_6[v, u] }"
60         - filter: "{ sync0[] }"

```

Regarding the thread index variables, we refer variable `t0` to `threadIdx.x` for a one-dimensional kernel, while for a two-dimensional kernel, we refer variable `t0` (resp. `t1`) to `threadIdx.y` (resp. `threadIdx.x`), which corresponds to the first (resp. second) inner `meta_for` loop. The maximum number of threads per thread-block is also limited by the hardware architecture. However, when the value of the program parameter `B`, which indicates the thread-block format, exceeds the hardware limit, the compilation fails when one intends to launch the kernel from the host code. Thus, although in the schedule tree, we explicitly say that `t0` is bounded by 512 (that is, the hardware limit) on the context node as shown on Line 18, this information is ignored by the subtree of that node. The task to control the value of `B` within the hardware limit is left to the user.

For this array reversal example, since we would like to use the shared memory for the input array `In` in the kernel, we need to insert relevant nodes in the schedule tree corresponding to copying data statements from the global memory to the shared memory. As shown on Lines 39-40, we insert a filter node that indicates the new statement reading from array `In[B + t0 + 1024i0]`. However, this is a trick that we use to encode the non-linear expression `t0 + B * i0` here, but we later replace it during the phase of translating the AST to the generated code. Then, a schedule node inserted on Lines 42-43 specifies an array `shared_In` to be allocated in the shared memory. In addition, the allocation unit size of this array `shared_In` is defined by `o1` on the extension node as on Line 33, while it is bounded by 1024 but later replaced by `BLOCK_0` during the kernel code generation phase. Note that `BLOCK_0` is predefined as a macro and specified its value at compile time.

Within the kernel mark node, a synchronization filter node is inserted on Line 44 after copy statements as well as on Line 60 after the computation statements. By the end, we insert several nodes, such as domain and guard, to provide the kernel information for the host code. The automatically generated CUDA kernel code for array reversal is shown in Figure 5.6 of

Chapter 5.

Turning our attention to the 1D Jacobi example as shown in Figure 5.3 of Chapter 5, we observe that each thread-block of the first kernel shall read $B+2$ elements of the input array, compute the average and update the result to an intermediate array. In the kernel code, we intend to use the shared memory for the input, while PPCG uses the global memory. Thereby, we extend the analysis for using the shared memory, such that when the total amount of required shared memory is based on the number of threads per thread-block and a constant (if applicable), we can allocate a corresponding array in the shared memory. Consequently, the following nodes for copy statements are inserted into the schedule tree for the first CUDA kernel of 1D Jacobi.

```
- filter: "[N, T, ub_v, B, b0, t0] -> { read[[i0, i1] -> a[i2]] : exists
(e0 = floor((-B - t0 + i2)/1024): 1024e0 = -B - t0 + i2 and t0 >= 0 and
t0 <= 1023) }"
child:
  schedule: "[N, T, ub_v, B, b0] -> shared_a[{ read[[i0, i1] -> a[i2]]
-> [(-B - 1024i1 + i2)] }]"
```

The above filter node indicates that a for loop will be generated to read for each thread, since the total amount of required shared memory per thread-block is more than the number of threads per thread-block. Moreover, the upper bound of this for loop and the allocation unit size of the array that is declared in the shared memory are defined by $o2$ on the extension node below. Note that there is one B used to replace 1024 so as to form the non-linear expression $B * i1$. One can easily calculate from the upper and lower bounds of $o2$ the amount $B+2$, which is used as the upper bound of the for loop for the copy statements and is replaced by $BLOCK_0+2$ for allocating the array in the shared memory. The automatically generated CUDA kernel code for 1D Jacobi is shown in Figure 5.4 of Chapter 5.

```
extension: "[N, T, ub_v, B, b0] -> { [i0, i1] -> sync1[];
[i0, i1] -> sync0[]; [i0, i1] -> read[[i0, i1] -> a[o2]] :
o2 >= B + 1024i1 and o2 <= 1 + 2B + 1024i1 and N >= 1 and
T >= 1 and ub_v >= 1 and B >= 0 and o2 <= -1 + N and o2 >= 0 }"
```

Appendix D

Examples Generated by PPCG

We present PPCG code with generated CUDA kernels for eight examples: *array reversal* (Figure D.1), *1D Jacobi* (Figure D.3), *2D Jacobi* (Figure D.4), *LU decomposition* (Figure D.5), *matrix transposition* (Figure D.7), *matrix addition* (Figure D.2), *matrix vector multiplication* (Figure D.6), and *matrix matrix multiplication* (Figure D.8).

```
#pragma scop
for (int i = 0; i < N; i++)
    Out[N - 1 - i] = In[i];
#pragma endscop

__global__ void kernel0(int *In, int *Out, int N) {
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    __shared__ int shared_Out[32];

    #define floord(n,d) (((n)<0) ? -((-n)+(d)-1)/(d)) : (n)/(d))
    for (int c0 = 32 * b0; c0 < N; c0 += 1048576) {
        __syncthreads();
        if (N >= t0 + c0 + 1)
            shared_Out[-t0 + 31] = In[t0 + c0];
        __syncthreads();
        if (N + t0 >= c0 + 32)
            Out[N + t0 - c0 - 32] = shared_Out[t0];
    }
}
```

Figure D.1: PPCG code and generated CUDA kernel for array reversal

```
#pragma scop
for (int v0 = 0; v0 < n; v0++)
    for (int v1 = 0; v1 < n; v1++)
        c[v0][v1] = a[v0][v1] + b[v0][v1];
#pragma endscop

__global__ void kernel0(int *a, int *b, int *c, int n)
{
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;

    #define floord(n,d) (((n)<0) ? -((-n)+(d)-1)/(d)) : (n)/(d))
    #define min(x,y) ((x) < (y) ? (x) : (y))
    for (int c0 = 32 * b0; c0 < n; c0 += 8192)
        if (n >= t0 + c0 + 1)
            for (int c1 = 32 * b1; c1 < n; c1 += 8192)
                for (int c3 = t1; c3 <= min(31, n - c1 - 1); c3 += 16)
                    c[(t0 + c0) * n + (c1 + c3)] =
                        (a[(t0 + c0) * n + (c1 + c3)] +
                         b[(t0 + c0) * n + (c1 + c3)]);
}
```

Figure D.2: PPCG code and generated CUDA kernel for matrix addition

```

__global__ void kernel0(int *a, int *b, int N, int T, int c0)
{
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;

    #define floord(n,d) (((n)<0) ? -((-n)+(d)-1)/(d)) : (n)/(d)
    for (int c1 = 32 * b0; c1 < N - 1; c1 += 1048576)
        if (N >= t0 + c1 + 2 && t0 + c1 >= 1)
            b[t0 + c1] = (((a[t0 + c1 - 1] + a[t0 + c1]) +
                a[t0 + c1 + 1]) / 3);
}

#pragma scop
for (int t = 0; t < T; ++t) {
    for (int i = 1; i < N-1; ++i)
        b[i] = (a[i-1] + a[i] + a[i+1]) / 3;
    for (int i = 1; i < N-1; ++i)
        a[i] = b[i];
}
#pragma endscop

__global__ void kernel1(int *a, int *b, int N, int T, int c0)
{
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;

    #define floord(n,d) (((n)<0) ? -((-n)+(d)-1)/(d)) : (n)/(d)
    for (int c1 = 32 * b0; c1 < N - 1; c1 += 1048576)
        if (N >= t0 + c1 + 2 && t0 + c1 >= 1)
            a[t0 + c1] = b[t0 + c1];
}

```

Figure D.3: PPCG code and generated CUDA kernel for 1D Jacobi

```

__global__ void kernel0(int *a, int *b, int N, int T, int c0)
{
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;

    #define floord(n,d) (((n)<0) ? -((-n)+(d)-1)/(d)) : (n)/(d)
    #define min(x,y) ((x) < (y) ? (x) : (y))
    for (int c1 = 32 * b0; c1 < N - 2; c1 += 8192)
        if (N >= t0 + c1 + 3)
            for (int c2 = 32 * b1; c2 < N - 2; c2 += 8192)
                for (int c4 = t1; c4 <= min(31, N - c2 - 3); c4 += 16)
                    b[(t0 + c1 + 1) * N + (c2 + c4 + 1)] =
                        (((a[(t0 + c1) * N + (c2 + c4 + 1)] +
                            a[(t0 + c1 + 2) * N + (c2 + c4 + 1)]) +
                            a[(t0 + c1 + 1) * N + (c2 + c4)]) +
                            a[(t0 + c1 + 1) * N + (c2 + c4 + 2)]) / 4);
}

#pragma scop
for (int t = 0; t < T; ++t) {
    for (int i = 0; i < N-2; ++i)
        for (int j = 0; j < N-2; ++j)
            b[i+1][j+1] = (a[i][j+1] +
                a[i+2][j+1] + a[i+1][j] +
                a[i+1][j+2]) / 4;
    for (int i = 0; i < N-2; ++i)
        for (int j = 0; j < N-2; ++j)
            a[i+1][j+1] = b[i+1][j+1];
}
#pragma endscop

__global__ void kernel1(int *a, int *b, int N, int T, int c0)
{
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;

    #define floord(n,d) (((n)<0) ? -((-n)+(d)-1)/(d)) : (n)/(d)
    #define min(x,y) ((x) < (y) ? (x) : (y))
    for (int c1 = 32 * b0; c1 < N - 2; c1 += 8192)
        if (N >= t0 + c1 + 3)
            for (int c2 = 32 * b1; c2 < N - 2; c2 += 8192)
                for (int c4 = t1; c4 <= min(31, N - c2 - 3); c4 += 16)
                    a[(t0 + c1 + 1) * N + (c2 + c4 + 1)] =
                        b[(t0 + c1 + 1) * N + (c2 + c4 + 1)];
}

```

Figure D.4: PPCG code and generated CUDA kernel for 2D Jacobi

```

__global__ void kernel0(double *L, double *U, int n, int c0)
{
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    __shared__ double shared_U_1[1][1];

#define floord(n,d) (((n)<0) ? -((-n)+(d)-1)/(d)) : (n)/(d))
{
    if (t0 == 0)
        shared_U_1[0][0] = U[c0 * n + c0];
    __syncthreads();
    for (int c1 = 32 * b0; c1 < n - c0 - 1; c1 += 1048576)
        if (n >= t0 + c0 + c1 + 2)
            L[c0 * n + (t0 + c0 + c1 + 1)] =
                (U[c0 * n + (t0 + c0 + c1 + 1)] / shared_U_1[0][0]);
}
}

#pragma scop
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n-k-1; i++) {
        // column major representation
        // of L and U
        int p = i + k + 1;
        L[k][p] = U[k][p] / U[k][k];
        for (int j = k; j < n; j++)
            U[j][p] -= L[k][p] * U[j][k];
    }
}
#pragma endscop

__global__ void kernel1(double *L, double *U, int n, int c0)
{
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    __shared__ double shared_L[1][32];
    __shared__ double shared_U_1[32][1];

#define floord(n,d) (((n)<0) ? -((-n)+(d)-1)/(d)) : (n)/(d))
#define min(x,y) ((x) < (y) ? (x) : (y))
#define max(x,y) ((x) > (y) ? (x) : (y))
if (n + 30 >= ((32 * b1 + 8191 * c0 + 31) \% 8192) + c0)
    for (int c1 = 32 * b0; c1 < n - c0 - 1; c1 += 8192) {
        if (t0 == 0)
            for (int c3 = t1; c3 <= min(31, n - c0 - c1 - 2); c3 += 16)
                shared_L[0][c3] = L[c0 * n + (c0 + c1 + c3 + 1)];
        __syncthreads();
        for (int c2 = 32 * b1 + 8192 * ((-32 * b1 + c0 + 8160)
            / 8192); c2 < n; c2 += 8192) {
            if (t1 == 0 && n >= t0 + c2 + 1)
                shared_U_1[t0][0] = U[(t0 + c2) * n + c0];
            __syncthreads();
            if (n >= t0 + c0 + c1 + 2)
                for (int c4 = max(t1, t1 + 16 * floord(-t1 + c0 - c2 - 1,
                    16) + 16); c4 <= min(31, n - c2 - 1); c4 += 16)
                    U[(c2 + c4) * n + (t0 + c0 + c1 + 1)] -=
                        (shared_L[0][t0] * shared_U_1[c4][0]);
            __syncthreads();
        }
        __syncthreads();
    }
}
}

```

Figure D.5: PPCG code and generated CUDA kernel for LU decomposition

```

__global__ void kernel0(int *a, int *b, int *c, int n)
{
    int b0 = blockIdx.x;
    int t0 = threadIdx.x;
    __shared__ int shared_a[32][32];
    __shared__ int shared_b[32];
    int private_c[1];

#define floord(n,d) (((n)<0) ? -((-n)+(d)-1)/(d)) : (n)/(d))
#define min(x,y) ((x) < (y) ? (x) : (y))
    for (int c0 = 32 * b0; c0 < n; c0 += 1048576) {
        for (int c1 = 0; c1 < n; c1 += 32) {
            if (n >= t0 + c1 + 1) {
                for (int c2 = 0; c2 <= min(31, n - c0 - 1); c2 += 1)
                    shared_a[c2][t0] = a[(c0 + c2) * n + (t0 + c1)];
                shared_b[t0] = b[t0 + c1];
            }
            __syncthreads();
            if (n >= t0 + c0 + 1 && c1 == 0)
                private_c[0] = 0;
            if (n >= t0 + c0 + 1)
                for (int c3 = 0; c3 <= min(31, n - c1 - 1); c3 += 1)
                    private_c[0] += (shared_a[t0][c3] * shared_b[c3]);
            __syncthreads();
        }
        if (n >= t0 + c0 + 1)
            c[t0 + c0] = private_c[0];
        __syncthreads();
    }
}

```

```

#pragma scop
for (int i = 0; i < n; i++) {
    c[i] = 0;
    for (int j = 0; j < n; j++)
        c[i] += a[i][j] * b[j];
}
#pragma endscop

```

Figure D.6: PPCG code and generated CUDA kernel for matrix vector multiplication

```

__global__ void kernel0(int *a, int *c, int n)
{
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    __shared__ int shared_a[32][32];

#define floord(n,d) (((n)<0) ? -((-n)+(d)-1)/(d)) : (n)/(d))
#define min(x,y) ((x) < (y) ? (x) : (y))
    for (int c0 = 32 * b0; c0 < n; c0 += 8192)
        for (int c1 = 32 * b1; c1 < n; c1 += 8192) {
            if (n >= t0 + c1 + 1)
                for (int c3 = t1; c3 <= min(31, n - c0 - 1); c3 += 16)
                    shared_a[t0][c3] = a[(t0 + c1) * n + (c0 + c3)];
            __syncthreads();
            if (n >= t0 + c0 + 1)
                for (int c3 = t1; c3 <= min(31, n - c1 - 1); c3 += 16)
                    c[(t0 + c0) * n + (c1 + c3)] = shared_a[c3][t0];
            __syncthreads();
        }
    }
}

```

```

#pragma scop
for (int v0 = 0; v0 < n; v0++)
    for (int v1 = 0; v1 < n; v1++)
        c[v0][v1] = a[v1][v0];
#pragma endscop

```

Figure D.7: PPCG code and generated CUDA kernel for matrix transpose

```

__global__ void kernel0(int *a, int *b, int *c, int n)
{
    int b0 = blockIdx.y, b1 = blockIdx.x;
    int t0 = threadIdx.y, t1 = threadIdx.x;
    __shared__ int shared_a[32][32];
    __shared__ int shared_b[32][32];
    int private_c[1][2];

    #define floord(n,d) (((n)<0) ? -((-n)+(d)-1)/(d)) : (n)/(d))
    #define min(x,y) ((x) < (y) ? (x) : (y))
    for (int c0 = 32 * b0; c0 < n; c0 += 8192)
        for (int c1 = 32 * b1; c1 < n; c1 += 8192) {
            if (n >= t0 + c0 + 1 && n >= t1 + c1 + 1) {
                private_c[0][0] = c[(t0 + c0) * n + (t1 + c1)];
                if (n >= t1 + c1 + 17)
                    private_c[0][1] = c[(t0 + c0) * n + (t1 + c1 + 16)];
            }
            for (int c2 = 0; c2 < n; c2 += 32) {
                if (n >= t0 + c0 + 1)
                    for (int c4 = t1; c4 <= min(31, n - c2 - 1); c4 += 16)
                        shared_a[t0][c4] = a[(t0 + c0) * n + (c2 + c4)];
                if (n >= t0 + c2 + 1)
                    for (int c4 = t1; c4 <= min(31, n - c1 - 1); c4 += 16)
                        shared_b[t0][c4] = b[(t0 + c2) * n + (c1 + c4)];
                __syncthreads();
                if (n >= t0 + c0 + 1 && n >= t1 + c1 + 1)
                    for (int c3 = 0; c3 <= min(31, n - c2 - 1); c3 += 1) {
                        private_c[0][0] +=
                            (shared_a[t0][c3] * shared_b[c3][t1]);
                        if (n >= t1 + c1 + 17)
                            private_c[0][1] +=
                                (shared_a[t0][c3] * shared_b[c3][t1 + 16]);
                    }
                __syncthreads();
            }
            if (n >= t0 + c0 + 1 && n >= t1 + c1 + 1) {
                c[(t0 + c0) * n + (t1 + c1)] = private_c[0][0];
                if (n >= t1 + c1 + 17)
                    c[(t0 + c0) * n + (t1 + c1 + 16)] = private_c[0][1];
            }
            __syncthreads();
        }
    }
}

```

```

#pragma scop
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; ++k)
            c[i][j] += a[i][k] * b[k][j];
#pragma endscop

```

Figure D.8: PPCG code and generated CUDA kernel for matrix matrix multiplication

Appendix E

The Implementation for Generating Comprehensive MetaFork Programs

In this appendix, we exhibit the pseudocode of the preliminary implementation of the *comprehensive optimization* algorithm demonstrated in Chapter 7. Algorithm 7 is the implemented algorithm for generating *comprehensive* MetaFork programs from a given MetaFork program, while Algorithm 8 and Algorithm 9 comprise the implemented *Optimize* procedure. In this implementation, we consider two resource counters: register usage per thread and data amount per thread-block to be cached in the shared memory; meanwhile, we apply three optimization strategies, including reducing register pressure, controlling thread granularity, and common sub-expression elimination.

Algorithm 7: MultiParametricCodeOptimizer(*fileName*)

Input: *fileName*, giving the location of the input program

Output: optimized versions of the input program in the form of a case discussion (depending on the hardware resource limits)

```
1 plans := [Create_Optimization_Plan(fileName)];
2 results := [];
3 while the number of plans <> 0 do
4   plan := plans[1]; plans := plans[2..-1];
5   task := ExtractTask(plan) [1];
6   new_plans := Optimize(plan, task);
7   for new_plan in new_plans do
8     if IsCompleted(new_plan) then
9       results := [new_plan, op(results)];
10    else
11      plans := [new_plan, op(plans)];
12 return results;
```

Algorithm 8: Optimize(*plan*, *task*)**Input:** *plan*, encoding a program being optimized, and *task*, an optimization task of plan**Output:** A list of new plans obtained by optimizing plan according to task

```

1 local caching_task, granularity_task, register_task, new_plans, optimized_plans, current_vars, alternative;
2 new_plans := [];
3 if task[NAME] = "Register_Pressure_Control" and task[CURRENTLEVEL] <= task[FINALLEVEL] then
4   alternative := Copy_Optimization_Plan(plan);
5   r := RegisterPressure(plan, task[CURRENTLEVEL]);
6   /* Accept case */
7   plan[CONSTRAINTS] := [ '<='(r, R.B), op(plan[CONSTRAINTS]) ];
8   if IsConsistent(plan) then
9     register_task := FindTask(plan, "Register_Pressure_Control");
10    register_task[CURRENTLEVEL] := register_task[FINALLEVEL] + 1;
11    plan[LOG] := ["Accept register pressure", op(plan[LOG])];
12    caching_task := FindTask(plan, "Caching");
13    if caching_task[CURRENTLEVEL] > caching_task[FINALLEVEL] then
14      granularity_task := FindTask(plan, "Granularity_Control");
15      granularity_task[CURRENTLEVEL] := granularity_task[FINALLEVEL] + 1;
16      plan[LOG] := ["No granularity reduction", op(plan[LOG])];
17    new_plans := [plan, op(new_plans)];
18  /* Refuse case */
19  alternative[CONSTRAINTS] := [ '<'(R.B, r), op(alternative[CONSTRAINTS]) ];
20  if IsConsistent(alternative) then
21    register_task := FindTask(alternative, "Register_Pressure_Control");
22    if (register_task[CURRENTLEVEL] < register_task[FINALLEVEL]) then
23      register_task[CURRENTLEVEL] := register_task[CURRENTLEVEL] + 1;
24      new_plans := [alternative, op(new_plans)];
25  else
26    optimized_plans := Optimize(alternative, FindTask(alternative, "CSE"));
27    if evalb(nops(optimized_plans) <> 0) then
28      new_plans := [op(optimized_plans), op(new_plans)];
29  else
30    optimized_plans := Optimize(alternative, FindTask(alternative, "Granularity_Control"));
31    if evalb(nops(optimized_plans) <> 0) then
32      new_plans := [op(optimized_plans), op(new_plans)];
33    /* No "else" case since we tried everything we could */
34    /* to reduce register pressure and we failed! */
35  /* To be continued in Algorithm 9 */

```

Algorithm 9: Optimize(*plan*, *task*)

Input: *plan*, encoding a program being optimized, and *task*, an optimization task of plan

Output: A list of new plans obtained by optimizing plan according to task

```

/* continuing Algorithm 8 */
1  else if task[NAME] = "Caching" and task[CURRENTLEVEL] <= task[FINALLEVEL] then
2      alternative := Copy_Optimization_Plan(plan);
3      z := CacheAmount(plan);
/* Accept case */
4      plan[CONSTRAINTS] := [ '<='(z, Z_B), op(plan[CONSTRAINTS]) ];
5      current_vars := { op((plan[RING])[variables]) };
6      current_vars := (current_vars union indets(z)) minus R_B, Z_B;
7      plan[RING] := RegularChains:~PolynomialRing([R_B, Z_B, op(current_vars)]);
8      if IsConsistent(plan) then
9          plan[LOG] := ["Accept caching", op(plan[LOG])];
10         task[CURRENTLEVEL] := task[FINALLEVEL] + 1;
11         register_task := FindTask(plan, "Register_Pressure_Control");
12         if register_task[CURRENTLEVEL] > register_task[FINALLEVEL] then
13             granularity_task := FindTask(plan, "Granularity_Control");
14             granularity_task[CURRENTLEVEL] := granularity_task[FINALLEVEL] + 1;
15             plan[LOG] := ["No granularity reduction", op(plan[LOG])];
16         new_plans := [plan, op(new_plans)];
/* Refuse case */
17         alternative[CONSTRAINTS] := [ '<'(Z_B, z), op(alternative[CONSTRAINTS]) ];
18         alternative[RING] := plan[RING];
19         if IsConsistent(alternative) then
20             optimized_plans := Optimize(alternative, FindTask(alternative, "Granularity_Control"));
21             if evalb(nops(optimized_plans) <> 0) then
22                 new_plans := [op(optimized_plans), op(new_plans)];
23             else
24                 optimized_plans := Optimize(alternative, FindTask(alternative, "CSE"));
25                 if evalb(nops(optimized_plans) <> 0) then
26                     new_plans := [op(optimized_plans), op(new_plans)];
27                 else
28                     task := FindTask(alternative, "Caching");
29                     task[CURRENTLEVEL] := task[FINALLEVEL] + 1;
30                     new_plan := AbandonCaching(alternative);
31                     new_plan[LOG] := ["Refuse caching", op(new_plan[LOG])];
32                     new_plans := [new_plan, op(new_plans)];
33     else if task[NAME] = "CSE" and task[CURRENTLEVEL] <= task[FINALLEVEL] then
34         task[CURRENTLEVEL] := task[CURRENTLEVEL] + 1;
35         new_plan := ApplyCSE(plan, task[CURRENTLEVEL]);
36         new_plan[LOG] := ["CSE applied", op(new_plan[LOG])];
37         new_plans := [new_plan, op(new_plans)];
38     else if task[NAME] = "Granularity_Control" and task[CURRENTLEVEL] <= task[FINALLEVEL] then
39         task[CURRENTLEVEL] := task[FINALLEVEL] + 1;
40         new_plan := SetGranularityToOne(plan);
41         new_plan[LOG] := ["Granularity set to 1", op(new_plan[LOG])];
42         new_plans := [new_plan, op(new_plans)];
43     return (new_plans);

```

Curriculum Vitae

Name: Ning Xie

Education

Degrees: Doctor of Philosophy in Computer Science
University of Western Ontario, 2012.09 - 2016.09

Master of Science in Information Technology
Hong Kong University of Science and Technology, 2010.09 - 2011.08

Bachelor of Engineering in Computer Science and Technology
Harbin Institute of Technology, 2006.09 - 2010.07

Related Work

Experience: Lecturer of CS3350B - Computer Architecture
University of Western Ontario, 2016.01 - 2016.04

Research Intern, funded by MITACS
Maplesoft, 2013.07 - 2013.10

Publications:

- Changbo Chen, Svyatoslav Covanov, Farnam Mansouri, Marc Moreno Maza, Ning Xie and Yuzhen Xie. “Parallel Integer Polynomial Multiplication”. Accepted by *SYNASC '16*. 2016
- Changbo Chen, Xiaohui Chen, Abdoul-Kader Keita, Marc Moreno Maza and Ning Xie. “MetaFork: A Compilation Framework for Concurrency Models Targeting Hardware Accelerators and Its Application to the Generation of Parametric CUDA Kernels”. In *Proceedings of the 25th Annual International Conference on Computer Science and Software (CASCOS '15)*. IBM Corp., 2015.11, p70-79.
- Sardar Anisul Haque, Marc Moreno Maza and Ning Xie. “A Many-core Machine Model for Designing Algorithms with Minimum Parallelism Overheads”. In *Proceedings of International Conference on Parallel Computing (ParCo'15)*. IOS press, May 2016, vol.27, p35-44.

- Changbo Chen, Farnam Mansouri, Marc Moreno Maza, Ning Xie and Yuzhen Xie. “The Basic Polynomial Algebra Subprograms”. In *Proceedings of the 4th International Congress on Mathematical Software (ICMS’14)*. Springer Berlin Heidelberg, 2014.08, vol.8592, p669-676.
- Sardar Anisul Haque, Xin Li, Farnam Mansouri, Marc Moreno Maza, Wei Pan and Ning Xie. “Dense Arithmetic over Finite Fields with the CUMODP Library”. In *Proceedings of the 4th International Congress on Mathematical Software (ICMS’14)*. Springer Berlin Heidelberg, 2014.08, vol.8592, p725-732.