University of Thessaly

Master Thesis

Optimizing memory management on heterogeneous systems using polyhedral, compile-time techniques

Author:
Vassilis Vassiliadis

Supervisor:
Christos D. Antonopoulos,
Nikolaos Bellas, Panayiotis
Bozanis

A thesis submitted in fulfilment of the requirements for the degree of Master

in the

Department of Electrical and Computer Engineering
University of Thessaly

November 2013

Declaration of Authorship

I, Vassilis Vassiliadis, declare that this thesis titled, 'Optimizing memory management on heterogeneous systems using polyhedral, compile-time techniques' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Βελτιστοποίηση διαχείρισης μνήμης σε ετερογενή συστήματα με χρήση πολυεδρικών τεχνικών κατά το χρόνο μεταγλώττισης

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Πανεπιστήμιο Θεσσαλίας

Διπλωματική Διατριβή

Η μέθοδός μας αφορά τεχνικές που εκτελούνται κατά τη μεταγλώττιση βασισμένες στο πολυεδρικό μοντέλο με σκοπό την εκτίμηση του προτύπου προσπέλασης μνήμης εφαρμογών ανεπτυγμένες χρησιμοποιώντας το προγραμματιστικό μοντέλο OpenCL. Έτσι, κατά τη διάρκεια της μεταγλώττισης, συλλέγονται πληροφορίες σχετικά με την προσπέλαση πινάκων, και αναπαριστόνται σε πολυεδρική μορφή. Μετά από έναν αριθμό διαδικασιών επεξεργασίας που κάνουν χρήση του πολυεδρικού μοντέλου, παράγουμε μία εκτίμηση του προτύπου προσπέλασης στη μνήμη.

Πιο συγκεκριμένα, ο αλγόριθμός μας παρέχει πληροφορία σχετικά με τα άνω και κάτω όρια των προσπελάσεων σε στοιχεία πινάκων. Η γνώση αυτή, θα χρησιμοποιηθεί τόσο κατά τη διαδικασία μεταγλώττισης όσο κατά την εκτέλεση ενός OpenCL πυρήνα σε πολλαπλές υπολογιστικές μονάδες. Έχουμε έτσι, μία εκτίμηση του μεγέθους εργασίας και της μνήμη που απαιτείται σε buffer/caches για την εκτέλεση του πυρήνα. Τέλος, έχουμε τη δυνατότητα να ελαττώσουμε τις μεταφορές δεδομένων μεταξύ host και compute devices με το να μεταφέρονται μόνο όσα στοιχεία πινάκων βρίσκονται μέσα στα όρια που ενέδειξε ο αλγόριθμός μας.

Acknowledgements

I would like to thank my advisors Christos D. Antonopoulos, Nicolaos Bellas and Panayiotis Bozanis for their help and guidance throughout this process, their ideas and feedback have been absolutely invaluable.

This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: "Thales. Investing in knowledge society through the European Social Fund".

Thales Project MIS 379416: "Advanced mathematical methods and software platform for solving multi-physics, multi-domain problems on modern computer architectures: applications to environmental engineering and medical problems".

Contents

List of Figures

List of Tables

Abbreviations

CAD	**C**omputer **A**ided **D**esign
CU	**C**ompute **U**nit
HLC	**H**igh **L**evel **C**ompiler
IR	**I**ntermediate **R**epresentation
ISL	**I**nteger **S**et **L**ibrary
LLC	**L**ow **L**evel **C**ompiler
PDE	**P**artial **D**ifferential **E**quation
PET	**P**olyhedral **E**xtraction **T**ool
SCoP	**S**tatic **C**ontrol **P**art
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SoC	**S**ystem on **C**hip
SPMD	**S**ingle **P**rogram **M**ultiple **D**ata

Dedicated to my family and friends

Chapter 1

Introduction

Heterogeneous parallel systems are becoming increasingly popular, as they offer high performance with relatively low cost and power dissipation. These architectures typically include a host system and a number of accelerators, in the form of GPUs, DSPs, or even FPGAs. Memory on such platforms is inherently distributed. This necessitates explicit data distribution and movement between system components. Typically accelerators have limited memory capacity, in comparison to system RAM. Quite often the memory hierarchy within the accelerator is also software controlled. Furthermore, massive multicore chips in the future will essentially be NUCA, necessitating careful data placement close to the compute units that will use them. Similarly, in the case of FPGAs, the on-chip memory (BRAMs) is organized as many, relatively small memory islands distributed on different areas of the chip.

The above are indicators that sophisticated data distribution and management is necessary for optimizing locality, performance, bandwidth exploitation and power efficiency. Many popular programming models, however, do not fully expose the communication pattern. Typical examples are OpenMP [1], or even OpenCL [2] which makes it impossible to express communication between kernels. Moreover application programmers tend to be "clumsy" in data management; for example they prefer using global over local buffers in OpenCL.

Our approach involves performing compile-time analysis using polyhedral techniques in order to estimate the memory access pattern of applications developed using the OpenCL programming model. During compile time, we collect information regarding accesses to array elements, in the form of polyhedra. After a number of polyhedral transformations we produce an estimation of the application's memory access pattern. More specifically our algorithm provides us with the access bounds on each array. This information is used both during compile time and the execution of a kernel on multiple

compute devices. It provides an estimation of the kernel's working set at compile time and thus the required memory buffers or cache. Furthermore, it allows for minimizing the memory transfers by only transfering between the host and the compute devices the array elements bounded by the ranges produced during the polyhedral analysis pass. Finally, estimating the required memory buffers/cache is of great importance when FPGAs are taken into account due to their limited memory capabilities. We can deploy an OpenCL kernel on an FPGA using the SOpenCL [3] toolchain, which is a framework for automatic hardware synthesis starting from unmodified OpenCL kernels. Essentially, SOpenCL can use the information retrieved by our polyhedral analysis in order to design the memory hierarchy on the FGPA so that data can be closer to where they are consumed/produced.

Finally, we apply our methodology on a hybrid Partial Differential Equation (PDE) solver[1]. The application was developed to provide an alternative method of approaching multi-domain, multi-physics problems. Its approach is based on breaking down a single PDE problem to produce a set of intermediate problems. Therefore a heterogenenous problem is substituted for a set of smaller homogeneous ones. For each of those sub-problems a solution is computed in a way that when all solutions are combined they form the solution to the original problem. This process attempts to provide more accurate solutions compared to solving a single multi-domain, multi-physics problem. After slicing is performed, new values have to be calculated for every point which lies on the boundary of each sub-region. These values are generated using a monte carlo algorithm for a number of boundary points which is then used to fill in the values of the remaining points on the boundaries using interpolation. Essentially, points are picked on the boundaries on the smaller regions for which random paths are walked. Each path uses a formula to produce an estimation of the value at its origin point. After a number of random walks are performed the results are averaged in order to compute the estimation. This enhances the overall process in two ways. Firstly, the results are more accurate compared to solving a single heterogeneous PDE problem. Additionally, this process results in a run-time speedup, because solving a possibly large problem is more time consuming that solving a set of smaller ones. In order to use our tool we ported the source code to the OpenCL[2] programming model. Then we used our tool to split the workload on both CPU and GPU. After all steps of optimization were performed we managed to achieve an average speedup of 18.3x. This enabled us to solve a problem, which would take $\sim$ 11 days (259.5 hours) for the original implementation, in $\sim$ 10.5 hours just by executing the monte carlo algorithm on both computing devices.

[1] https://github.com/mvavalis/Hybrid-numerical-PDE-solvers/
[2] It was originally written using the pthreads API.

This thesis is organized as follows. We begin by introducing the polyhedral model in *chapter 2*. Then in *chapter 3* we provide some background for SOpenCL and OpenCL. Finally in *chapter 4* we present and evaluate our algorithm by applying it to a real-life application. We ported a hybrid numerical partial differential equation (PDE) solver multithreaded application to OpenCL and then used our framework to optimize its execution.

Chapter 2

Background - Polyhedral Model

The polyhedral model (also called polytope model) is a mathematical framework mostly used for loop nest analysis in compiler optimization. The polyhedral method treats each loop iteration within nested loops as lattice points inside mathematical objects called polyhedra (polytopes), performs affine transformations or more general non-affine transformations such as tiling on the polyhedra, and then converts the transformed polyhedra into equivalent, but optimized – depending on the target optimization goal – loop nests through polyhedra scanning.

2.1 Polyhedron

In elementary geometry a polyhedron is a geometric object with flat sides, which exists in any general number of dimensions. When referring to an n-dimensional generalization, the term n-polyhedron is used.

A convex polyhedron has two representations [4] the *implicit* and *the generators* representation. The first describes a polyhedron as the intersection of a finite number of half spaces. The latter presents a polytope as a combination of vertices, rays and lines; every point in a polyhedral domain can be generated by a linear combination of its generators. Finally, Chernikova described a method to produce one representation from the other [5].

2.1.1 Implicit representation

A polyhedron domain (D) is defined as the intersection of a finite set of closed linear half-spaces. This representation is specified by a system of equalities and inequalities:

$$D: \quad x \quad \mathbb{Q}^n \; Ax = b, Cx \geq d$$

2.1.2 Generators representation

The Minkowski form [6] describes a polyhedron as a combination of lines, rays and vertices.

$$D: \quad x \quad \mathbb{Q}^n \; x = L\lambda + R\mu + V\nu, \textstyle\sum_i \nu_i$$

In other words, every point in Domain D is a linear combination of lines, a positive linear combination of unidirectional rays and a convex combination of vertices.

The definitions described above are equivalent. Furthermore, each can be constructed using their dual counterpart. However, they allow for different kinds of transformations which is why some polyhedral model libraries, such as PolyLib [7] keep both descriptions for a polyhedron at the same time, given that it can be quite costly, in terms of computation, to transform one representation to the other.

2.2 Parametric Polyhedron

A common representation of parametric (or parameterized) polyhedra is a linear function of the parameter vector p which is of dimension m:

$$D(p): \quad x \quad \mathbb{Q}^n, p \quad \mathbb{Q}^m \; Ax = Bp + b, Cx \geq Cp + d$$

or equivalently

$$D(p): \left\{ \begin{pmatrix} x \\ p \end{pmatrix} \quad \mathbb{Q}^{n+m} \; A' \begin{pmatrix} x \\ p \end{pmatrix} = b, C' \begin{pmatrix} x \\ p \end{pmatrix} \geq d \right\}$$

Where $A' = [\; A \; -B \;]$ and $C' = [\; C \; -D \;]$

In order to let the reader get a better understanding of polytopes we will now introduce the restrictions which come with the Polyhedral Model.

2.3 Polyhedral Analysis

Polyhedral analysis is the process of representing nested loop structures along with their program statements in the form of polyhedra. Upon analysis of the input source code, Static Control Parts (SCoPs) are identified. SCoPs are source code fragments that hold a set of characteristics which if not present polyhedral analysis cannot be applied. These characteristics are listed below:

- All variables present in the SCoP that are taken into account must fall into one of the following two categories:
  1. Iterator: The variable is used as an iterator in one of the enclosing nested loops.
  2. Parameter: The variable maintains the same value maybe throughout the execution of the SCoP.
- Each loop's bounds must be either a) constant, or b) a linear combination of loop iterators, parameters and/or constants (affine). This also applies to the conditions on if statements.
- Expressions used to index arrays should be computed as affine combinations of parameters, iterators and/or constants.
- Data flow between statements in the loop must be explicit. In other words statements may not communicate with each other using shared variables invisible to the compiler.

Provided that all rules above are met, polyhedral analysis can produce a polytope. Each integer point of said polyhedron is mapped to the execution of a statement or, in the context of our thesis, to the access of a variable. This allows for manipulation of the program structures through polyhedral transformations while maintaining the original functionality as well as the source code correctness of the SCoP.

In other words the polyhedral model can describe the schedule of a program, however it can also be used for a wide class of problems as long as they are adhering to the above rule set. For example, our algorithm constructs polyhedra for array accesses: in this context each integer point of our polyhedra corresponds to accessing an array element.

Polyhedral analysis is the basis for powerful methods which are present in popular compilers like LLVM/CLANG [8], and GCC [9]. By constructing and transforming polyhedra the compiler can identify dependencies between statements as well as discover parallelism

which is present in the input source code. This information can be exploited in order to produce optimized program schedules as well as perform automatic parallelization [8–11].

2.3.1 Polyhedral analysis example

We show a simple example of applying polyhedral analysis to a small SCoP which consists of two nested loops:

```
1 for ( i=2; i<=N; ++i ) {
2    for ( j=2; j<=min(M,-1+N+2); ++j ) {
3       S1(i,j);
4    }
5 }
```

Listing 2.1: Simple SCoP

This SCoP is translated to the following inequalities (constraints).

- $2 \leq i \leq N$
- $2 \leq j \leq min(M, -1 + N + 2)$

Which can be further refined to the following:

- $2 \leq i \leq N$
- $2 \leq j \leq M$
- $2 \leq j \leq N + 1$

The resulting polyhedron description after expanding the above relations into trivial constraints is presented below using the Implicit form of *section 2.1.1*:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & -2 \\ 1 & 0 & 1 & 0 & 0 & -2 \\ 1 & -1 & 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 1 & 0 & 1 \end{bmatrix}$$

or equivalently

$$P_{\left(\begin{smallmatrix} N \\ P \end{smallmatrix}\right)} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix}, \begin{pmatrix} N \\ P \end{pmatrix} \quad \mathbb{Z}^2 \begin{pmatrix} i \\ j \end{pmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \\ 0 & -1 \end{bmatrix} + \begin{pmatrix} N \\ P \end{pmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \geq \begin{bmatrix} 2 \\ 2 \\ 0 \\ 0 \\ -1 \end{bmatrix} \right\}$$

where i, j are the loop iterators and N, P are parameters; the latter maintain their values throughout the extent of the SCoP.

From now on the **implicit representation** will be used to describe polyhedra. It consists of a matrix with $(1 + Number\_of\_Iterators + Number\_of\_Parameters + 1)$ columns and as many rows as the number of constraints which form the polyhedron. *Figure 2.1* is a visual breakdown of the Implicit Representation. In the following lines we further explain the Implicit Representation form.

- The first column indicates whether the i_{th} row describes an equality or inequality by setting the column vector's i_{th} value equal to 0 or 1 respectively.
- Following the first column there exist a "group" of column vectors, one for each iterator present in the polyhedron domain. We will refer to this submatrix's values by $I[row][column]$. The value $I[i][j]$ denotes the coefficient of iterator j in constraint i.
- Using the same idea the next group of column vectors represent the parameters' coefficients. We will refer to the values of this submatrix using the notation $P[row][column]$.
- The last column vector is the constant component of the affine constraints.

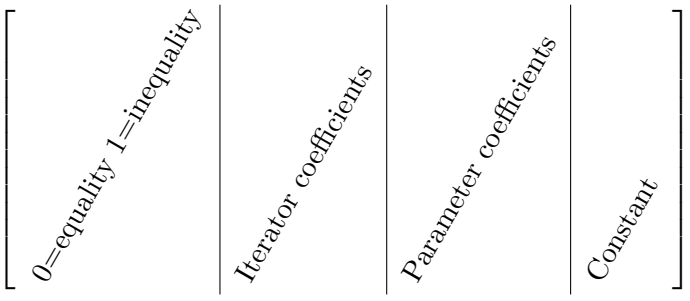

FIGURE 2.1: Visual breakdown of the implicit representation

If we wanted to add the additional constraint $5*j < 2*i+N*3+4*M$ we would have to extend the above matrix with the following row (constraint) $\begin{bmatrix} 1 & 2 & -5 & 3 & 4 & -1 \end{bmatrix}$ which represents the inequality $2*i-5*j+3*N+4*M-1 \geq 0$.

2.4 Polyhedral optimization pass methodology

All compiler optimization techniques which are based on the polyhedral model roughly follow the following four step process:

1. **SCoP identification:** Polyhedral analysis can only be performed when certain conditions are met; refer to *section 2.3* for a summary of the requirements. It is the compiler's responsibility to discover such code fragments in order to apply polyhedral analysis on them.

2. **Polyhedra formation:** For every SCoP identified during the previous step, a number of polyhedra are produced. These represent the statements and data as well as control flow dependencies present in the SCoP.

3. **Optimization through transformation:** During this step the polyhedra undergo a series of polyhedral transformations in order to produce new polyhedra, which describe an optimized version of the original source code.

4. **Code generation:** The new set of polyhedra are now used as input to produce one or more source code fragments which can, under certain circumstances, execute in parallel. In any case the resulting source code is an optimized version of the original one.

Most of the optimization techniques present in modern compilers use the above skeleton. When blocks of statements are proven to be parallelizable source code transformations take place in order to produce parallel code segments. These perform the original computation in parallel, resulting to a much more efficient usage of hardware as well as improved application performance [10, 12, 13].

Even if the optimization pass fails to identify parallel blocks of statements, performance gains can still be accomplished by modifying the sequential schedule of the source code in order to remove overhead cost introduced by if/while/for conditions. This may increase the code size of applications but in return it removes a number of branches which decreases the execution time of the SCoP. This is done by producing source code which *scans* the integer points of the polyhedron. Scanning means that each integer point in the polyhedral domain specified by a polytope is visited in *order*. Since a point in the polyhedra formed by the the analysis is mapped to the execution of a set of statements, this *scan* is in fact a program schedule for the original algorithm.

Polyhedral analysis enforces quite strict rules regarding the source code on which it can be performed. However, it allows for many loop transformations which are common in

compiler optimization passes to be applied with ease. We will present a few examples of loop transformation techniques In more detail:

- **Loop tiling:** The process of transforming nested loops by splitting their iteration space into smaller subdomains called blocks. This transformation is intended to force the enclosed statements to operate on fewer data per iteration. If applied properly, cache efficiency increases because data are being reused in the inner nested loop. Since exploiting the cache mechanism can lead to a huge improvement in application performance, loop tiling is a quite popular loop transformation in the high performance computing community.
- **Loop splitting:** Breaks the iteration space of a loop into a set of contiguous subsets which can be then executed in sequence. The goal here is to simplify loops by removing condition checking and/or data/control flow dependencies statically, at the cost of increasing code size.
  - A special case of *loop splitting* is **loop peeling** which removes some of the first (or last) iterations and executes them separately before (or after) the loop body.
- **Loop fission:** Loop fission is similar to the above two transformations. A given loop structure is split into multiple smaller ones which have the same iteration space but each one executes a susbet of the original block of statements. The reverse transformation is called **loop fusion**. From a set of loops with the same iteration domain a new loop structure is created whose body is a combination of the bodies of the individual ones. If applied properly both transformations can achieve better *locality of references*, which can improve the application performance.
- **Loop unrolling:** Is the process of rewritting a loop's body and stride in order to have one iteration execute the workload of multiple ones provided they are sequential. This optimization attempts to improve performance by a) reducing end-of-loop tests and branches, b) reducing memory delays especially in the case of memory read accesses, c) pipelining and so on. However this comes at the expense of code size because the original loop's body statements are replicated a number of times.

2.4.1 Polyhedral analysis optimization example

A popular algorithm based on polyhedral analysis is Quillere's algorithm [13]. It produces an optimized code which scans the input polyhedra. Because polyhedra can be

used to represent SCoPs, Quillere's algorithm may be used as a compiler optimization pass. The final output is the optimized schedule for the original source code. In certain cases where parallelism can be discovered the algorithm's output will actually consist of schedules of source code that can be executed concurrently.

The algorithm recursively iterates the dimensions of the input polyhedron. In each iteration of the process the existing polyhedra are separated into a new list of disjoint polyhedra. Afterwards the lexicographic ordering graph is created, an edge from polyhedron P to polyhedron Q indicates that P precedes Q and should be scanned earlier. Finally, if no ordering exists between two polyhedra they they can safely execute in parallel.

We will now apply a simple optimization pass on a small SCoP in order to optimize the schedule of its statements. Consider the following source code:

```
1  for ( i=1 ; i<=max(N,M); i++ ) {
2     for ( j=1; j<=N; j++ ) {
3        if ( j==i )
4           S1(i,j);
5        else if ( i<=j )
6           S2(i,j);
7        if ( j==N )
8           S3(i,j);
9     }
10 }
```

LISTING 2.2: Example code

The polyedra generated during the extended Quillere algorithm execution are presented in figure 2.4.1. The Y-axis stands for the dimension introduced by iterator j, while the X-axis corresponds to the iterator i.

The code which scans the final polyhedra in lexicographic order is presented in *listing 2.3*.

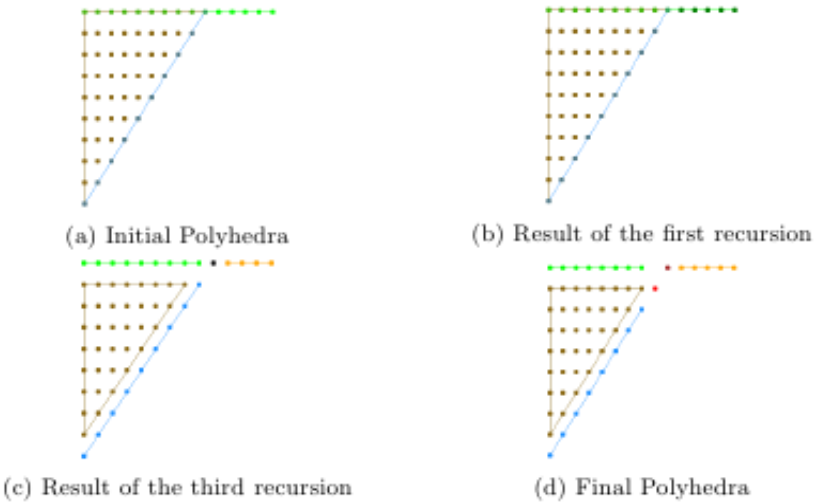

FIGURE 2.2: Steps of the extended Quillere algorithm

```
1  for ( i=1; i<=N; i++ ) {
2      S1(i,i);
3      S2(i,i);
4      for ( j=i+1; j<N; j++ ) {
5          S2(i,j);
6      }
7      S2(i,N);
8      S3(i,N);
9  }
10 S1(N-1, N-1);
11 S2(N-1, N-1);
12 S2(N-1, N);
13 S3(N-1, N);
14 S1(N, N);
15 S2(N, N);
16 S3(N, N;
17 for ( i=N+1; i<=M; i++ ) {
18     S3(i, N);
19 }
```

LISTING 2.3: Optimized code

Instead of producing if statements, polyhedral analysis has opted to make use of loop transformations and statement duplication. The resulting source code has fewer branches but its size has increased; this performance/code-size trade-off ratio can be fine-tuned by the compiler.

2.5 Ehrhart polynomials

The ability to produce useful information regarding memory accesses such as [14]:

- Dynamic array size.
- Times an array is accessed.
- Number of array elements accessed[1].
- Number of loop iterations executed, etc.

is quite important because it allows for compiler optimizations which otherwise would be dangerous to perform since application correctness would not be guaranteed.

Ehrhart [15] showed that the number of integral points in a parametric polyhedron can be represented as a quasi-polynomial (pseudo-polynomial) expression. The only requirement is that the polyhedron Pp can be represented as a convex combination of its parametric vertices:

$$P_p = \left\{ x \quad \mathbb{Q}^n \; x = \lambda V(p), 0 \leq \lambda_j, \sum_j \lambda_j = 1 \right\}$$

Where $V(p)$'s columns are the vertices of P_p and each vertex $v_j(p)$ is an affine combination of the parameters with rational coefficients. In essense a quasi-polynomial is a polynomial whose coefficients are n-periodic numbers which depend (periodically) on the polyhedron's variables (iterators).

In mathematic terms [14, 15] a n-periodic number is a function $\mathbb{Z}^n \quad \mathbb{Z}$, such that there exist periods $q = (q_1, ..., q_n) \quad \mathbb{N}^n$ such that $U(p) = U(p')$ whenever $p_i \equiv p'_i \, mod \, q_i$, for $1 \leq i \leq n$. The least common multiple of all q_i is called the period of $U(p)$.

2.5.1 Ehrhart example

To illustrate the functionality of the Ehrhart polynomials we will now apply Ehrhart's methodology in order to count the number of loop iterations in the double nested loop

[1] In our algorithm we use Ehrhart polynomials for this exact purpose. After our polyhedral analysis and workload-partitioning pass, we use the Ehrhart polynomials to get an (over)estimation of the number of elements addressed by each memory accessing statement. We will elaborate on this aspect of our approach in *section 4.1*.

presented in code listing 2.1. As we mentioned earlier the polyhedron produced is

$$P_{\binom{N}{P}} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix}, \begin{pmatrix} N \\ P \end{pmatrix} \quad \mathbb{Z}^2 \begin{pmatrix} i \\ j \end{pmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \\ 0 & -1 \end{bmatrix} + \begin{pmatrix} N \\ P \end{pmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \geq \begin{bmatrix} 2 \\ 2 \\ 0 \\ 0 \\ -1 \end{bmatrix} \right\}$$

The respective erhrhart polynomial is:

$$\#P_{\binom{N}{P}} = \begin{cases} (N-1) * N, & \text{if } M \geq N+1 \text{ and } N \geq 2 \\ (N-1) * (M-1) & \text{if } N - M + 1 \geq 0 \text{ and } N \geq 2 \text{ and } M \geq 2 \end{cases}$$

Since the polyhedron is parameterized, the Ehrhart polynomial uses the polyhedron's parameters to produce a count for its integral integer points. This follows our intuition because the parameters determine the loop bounds. The outter loop will always execute $N-2+1 = N-1$ times whereas the inner loop will execute $N+1-2+1 = N$ times if $M < N+1$, otherwise if $N+1 < M$. the loop will iterate $M-2+1 = M-1$ times. In any case the Ehrhart polyhedron is equal to multiplying the number of times the loops will iterate.

Chapter 3

Background - Silicon OpenCL

Silicon OpenCL (SOpenCL) [3] is an architectural synthesis CAD tool targeting heterogeneous parallel computing platforms. The objective is to allow a software programmer develop an OpenCL application once and deploy it on any platform, without the hassle of additional modifications. Before going into any more details regarding SOpenCL we will first briefly introduce the OpenCL programming model.

3.1 Open Computing Language - OpenCL

In the past years due to the high demand for realtime and high-definition 3D graphics GPUs have evolved into highly parallel, multithreaded, manycore processors with tremendous computational horsepower and very high memory bandwidth [16] (*figure 3.1*)[1].

The GPU is specialized for compute-intesive, highly parallel computation since it was designed to facilitate rendering graphics. Thus more chip area is used for data processing rather than data caching and flow control as illustrated by *figure 3.2*. The general principle is to hide memory access latency with calculations instead of big data caches.

3.1.1 Platform model

An OpenCL application consists of two parts: kernels which execute on OpenCL devices and a host program which executes on the host. The latter is responsible for initializing and managing the kernel execution on the computing devices.

[1]The images presented in this section are from NVIDIA's "OpenCL Programming Guide for the CUDA Architecture" which can be found at http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf

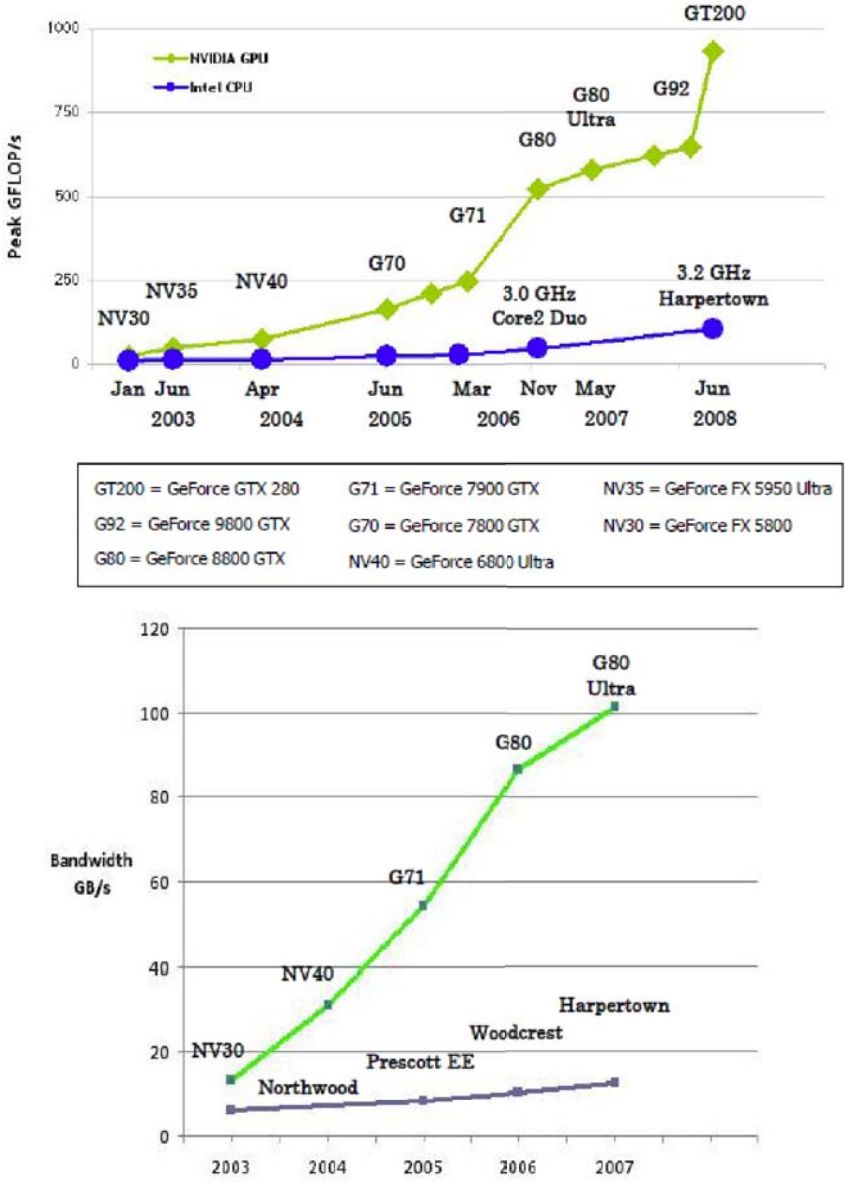

FIGURE 3.1: Floating point operations per second and memory bandwidth for the CPU and GPU.

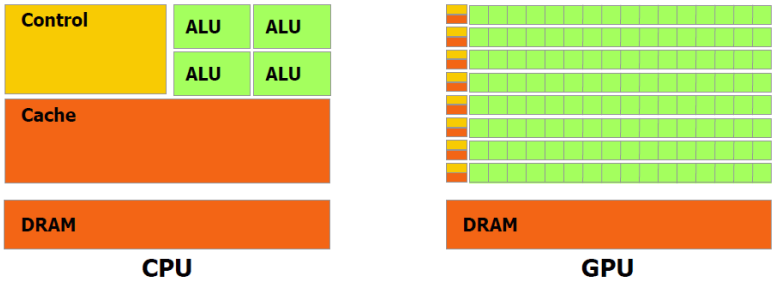

FIGURE 3.2: The GPU devotes more transistors to data processing.

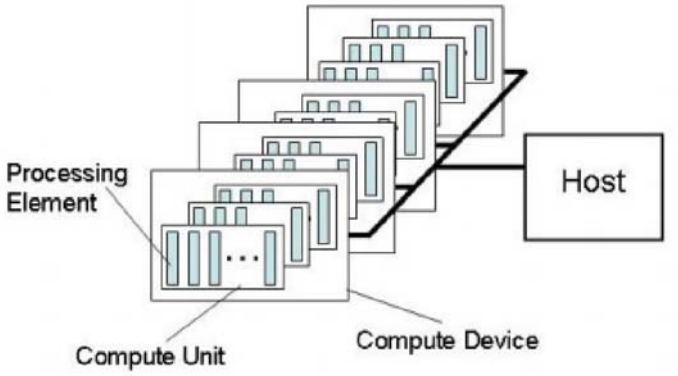

FIGURE 3.3: OpenCL Platform Model: One host with one or more compute devices each with one or more compute units each with one or more processing elements.

Execution devices in OpenCL vary and can be CPUs, GPUs or even custom hardware accelerators in the form of FPGAs or SoCs. A compute device is partitioned into a number of *compute units* (CUs) which are further divided into one or more *processing elements* (PEs). Kernel execution on a device occurs within the processing elements. The platform model is illustrated in *figure 3.3*.

3.1.2 Execution model

An OpenCL application runs on a host which submits commands to execute computations on the processing elements within a device. These execute a single stream of instructions as SIMD units. More specifically, processing elements may either execute in lock-step in a single stream of instructions (e.g. GPUs) or as SPMD units with each PE maintaining its own program counter (e.g. CPUs).

Before the execution of an OpenCL kernel an index space must be created on which the thread topology is mapped. This index space is called NDRange; each point in it is a thread. In OpenCL terms such points are called *work-items*. These are identified by their position in NDRange.

Work-items are organized into work-groups. These blocks of threads are identified by a unique work-group ID with the same dimensionality as the index space used for the work-items. As such, a thread may be uniquely identified by its global ID or a by a combination of its local ID and work-group ID.

OpenCL's NDRange is a 1D, 2D, or 3D index space. It is defined by an integer array of length N $1, 2, 3$ specifiying the extent of the index space in each dimension starting at an offset index F which by default is 0. Each work-item's global and local ID are n-dimensional. The global ID components take values in the range

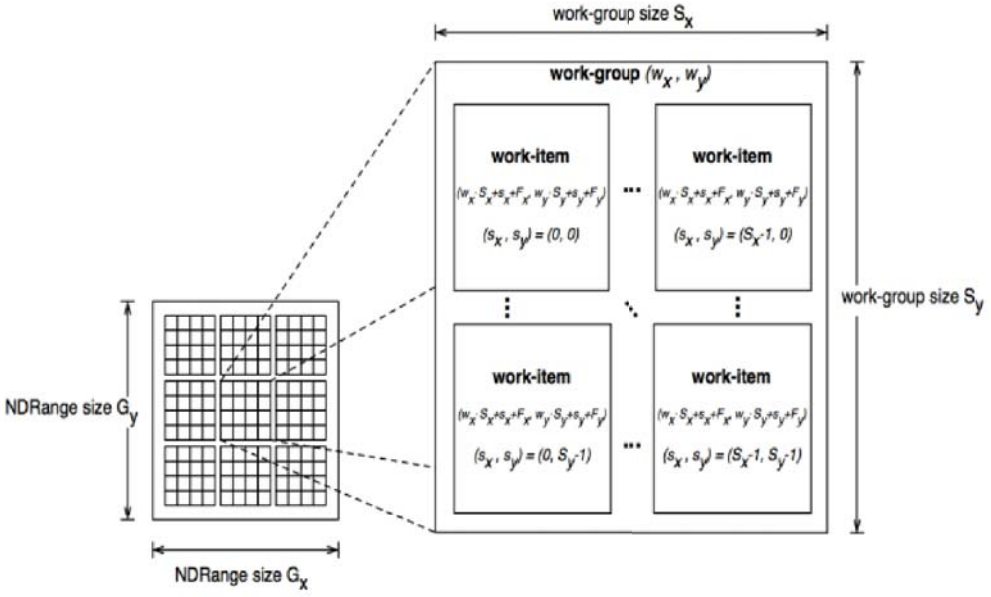

FIGURE 3.4: NDRange: Work-items with global IDs and their mapping onto the pair of work-group and local IDs

$[F, F + global\_size\_DIM - 1]$; where DIM is x, y, or z and $global\_size\_DIM$ is the number of elements in that dimension.

In the same manner work-groups are assigned IDs, an array of length N defining the number of work-groups in each dimension. Work-items are assigned to a work-group and given a local ID tuple with componentsin the range $[0, work\_group\_size\_DIM - 1]$. The organisation of threads in an NDRange can be seen in *figure 3.4*.

In addition to the NDRange, an OpenCL *context* has to be constructed prior to the execution of OpenCL kernels. The context contains information about the following resources:

Devices The compute devices to be used for the execution of the kernels.

Kernels The OpenCL functions to be executed on the compute devices.

Program Objects Program source and executable that implements the kernels.

Memory Objects Set of buffers visible to host and OpenCL compute devices that contain data which will be passed as parameters to the kernels.

Following the instatiation of the OpenCL context, a *command-queue* is created in order to manage the execution of the kernels on the compute devices. The *command-queue* is used to hold commands originating from the host untill they are scheduled on a context's compute device. Such commands may be one of the following:

Kernel execution commands Initiate the execution of a kernel on a device.

Memory commands Initiate a memory transfer to, from, or between memory objects, or map and unmap memory objects from the host address space.

Synchronization commands Constrain the order of execution of commands.

The above commands execute asynchronously between the host and the device. However commands execute relative to each other in one of two modes:

In-order execution Commands are launched and complete in the order they appear in the command-queue, in other words the commands are serialized.

Out-of-order execution Commands are issued in order, but a command does not wait for prior ones to complete before it executes. The programmer is responsible to explicitly enforce order constraints by means of synchronization commands.

3.1.3 Memory model

Memory in OpenCL may be one of the following four different types:

Global memory This memory is instatiated by the host and permits read/write access to all work-items, in all work-groups.

Constant memory Constant memory is instatiated and filled with data by the host; it permits read access to all work-items, in all work-groups but not write access.

Local memory This memory is local to a work-group and contains data which are shared to all the work-items in that work-group.

Private memory Private memory contains a work-item's private variables which are not shared with any other work-item.

More information regarding the different characteristics of the types of OpenCL memory can be found in *table 3.1*. Furthermore, a conceptual OpenCL device architecture is illustrated in *figure 3.5*.

In OpenCL host and compute device memory models are disjoint for the most part. They do however need to interact in order to make transferring data between the devices and the host program possible. This can be accomplished by either explicitly copying data to/from the devices or by mapping/unmapping regions of a memory object.

	Global	**Constant**	**Local**	**Private**
Host	Dynamic allocation. Read/Write access.	Dynamic allocation. Read/Write access.	Dynamic allocation. No access.	No allocation. No access.
Kernel	No allocation. Read/Write access.	Static allocation. Read access.	Static allocation. Read/Write access.	Static allocation. Read/Write access.

TABLE 3.1: Memory region, allocation and memory access capabilities.

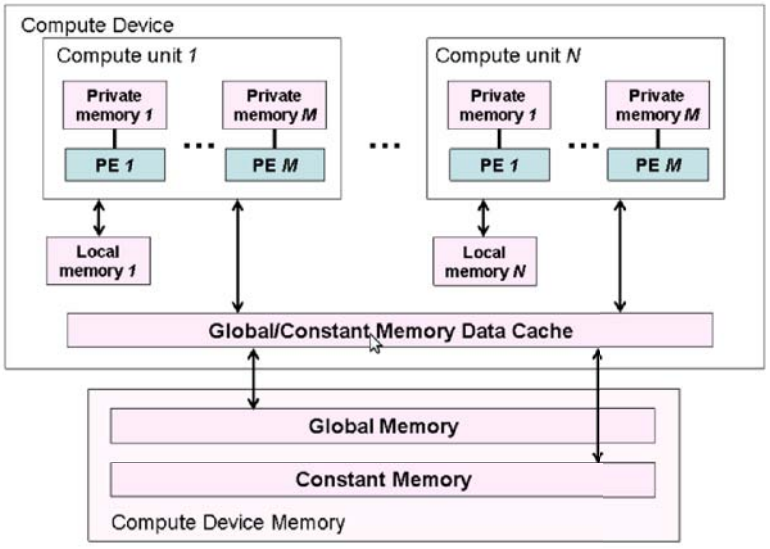

FIGURE 3.5: Example OpenCL device architecture with processing elements, compute units and devices. The host is not shown.

Explicit data copying to/from the compute devices is managed by the host. These memory transfers may either be blocking or non-blocking; a blocking call will complete when the associated memory resources on the host can be safely reused. However, a non-blocking command will return as soon as the command is eqnueued without waiting till its completion.

The host program may also choose to map a memory object onto its address space. Similarly to explicit data copying the map/unmap commands may either be blocking or non-blocking. Once the host has finished performing read/write accesses onto the mapped memory region it unmaps the memory object.

3.1.4 Synchronization mechanisms

Synchronization in OpenCl is performed in two possible scenaria:

- Synchronization between work-items in a work-group.
- Commands within one or more command-queues in a single OpenCL context.

In an OpenCL kernel the work-items of a work-group may make use of a work-group barrier in order to achieve synchronization amonsgt them. However using a barrier in OpenCL must be done carefully. All work-items in the group or none at all must execute the barrier, otherwise the kernel will come to an unnatural halt and will eventually have to be terminated. Finally there is no built in synchronization mechanism between work-groups.

Synchronizing commands in command-queues can be achieved by either using command-queue barriers or waiting on an event. A barrier can be used to synchronize commands in a single command-queue. It ensures that all previously commands have finished executing and any resulting updates to memory objects are visible to subsequently enqueued commands before they execute.

3.1.5 Memory objects

Memory objects can either be *buffer objects* or *image objects*. A *buffer object* is a one-dimensional array of scalar, vector, or even user-defined structures. An *image object* is used to store a two- or three- dimesnional texture, frame-buffer, or image. Its elements are selected from a list of predefined image format. In both cases the minimum number of elements in a memory object is one. Memory objects are described by a **cl_mem** object and are used as input/output arguments to the OpenCL kernels.

There are two fundamental differences between a *buffer* and an *image* object:

1. Elements in a buffer are stored in sequential fashion and can be accessed using a pointer by a kernel executing on a device. Elements of an image are stored in a format that is opaque to the user and cannot be directly accessed using a pointer. Built-in functions are provided by the OpenCL C programming language to allow a kernel to read from or write to an image.
2. For a buffer object, the data is stored in the same format as it is accessed by the kernel, but in the case of an image object the data format used to store the image elements may not be the same as the data format used inside the kernel. Image

elements are always a 4- component vector (each component can be a float or signed/unsigned integer) in a kernel. The built-in function to read from an image converts image element from the format it is stored into a 4-component vector. Similarly, the built-in function to write to an image converts the image element from a 4-component vector to the appropriate image format specified such as 4 8-bit elements, for example.

3.1.6 Trivial example - vector_add

A vector add kernel is one of the simplest OpenCL kernels and a great way to get introduced to the OpenCL programming model. Each thread reads two integers then sums them and finally stores the result in an output array. The respective OpenCL kernel can be seen in *listing 3.1*.

```
1  __kernel void vector_add(  __global const double *a,
2      __global const double *b, __global double *c,
3      const unsigned int n)
4  {
5      /* Each thread uses one integer from the a,b,c arrays based on his
6      unique global id, the kernel is instantiated using an 1D NDRange */
7      int id = get_global_id(0);
8
9      /* Avoid out of bounds access, this *might* introduce divergence in the
10     last threads but it is not a major concern since it does not involve a
11     large number of threads or a large number of operations */
12     if (id < n) {
13         c[id] = a[id] + b[id];
14     }
15 }
```

Listing 3.1: Simple vector add kernel implemented in OpenCL.

3.2 SOpenCL - Silicon OpenCL

The tool consists of a two level compilation process: High Level Compilation (HLC) and Low Level Compilation (LLC). The **high level compiler** processes an OpenCL application and partitions its kernels as appropriate across the available computing platforms (*figure: 3.6*): a) CPU, b) GPU, and c) FPGA. The **low level compiler** processes OpenCL kernels selected to run on FPGA platforms. The task of the LLC is to compile an OpenCL kernels and generate an equivalent hardware design that fis the target FPGA decice and fulfills performance requirements. Additionally, SOpenCL provides

runtime environments for each of the target platforms to facilitate their integration and the execution of OpenCL kernels.

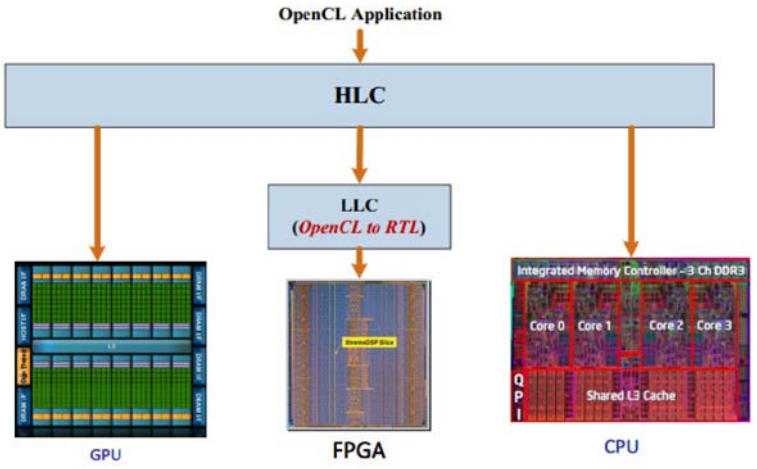

FIGURE 3.6: Silicon-OpenCL Tool Flow.

Figure 3.7 shows the low level compiler flow. The LLC converts unmodified OpenCL kernels into a system on chip (SoC) with hardware and software components. The tool flow generates a hardware accelerator for each OpenCL kernel in two phases: OpenCL-to-C transformation, and C-to-RTL. The tool flow also generates the runtime environment and drivers, in addition to the testbench generated for simulation and verification purposes. The OpenCL-to-C frontend developed by Daloukas [17] generates a C function from an OpenCL kernel by coarsening the computation granularity. The C-to-RTL backend developed in this thesis generates a hardware accelerator RTL description for each OpenCL kernel.

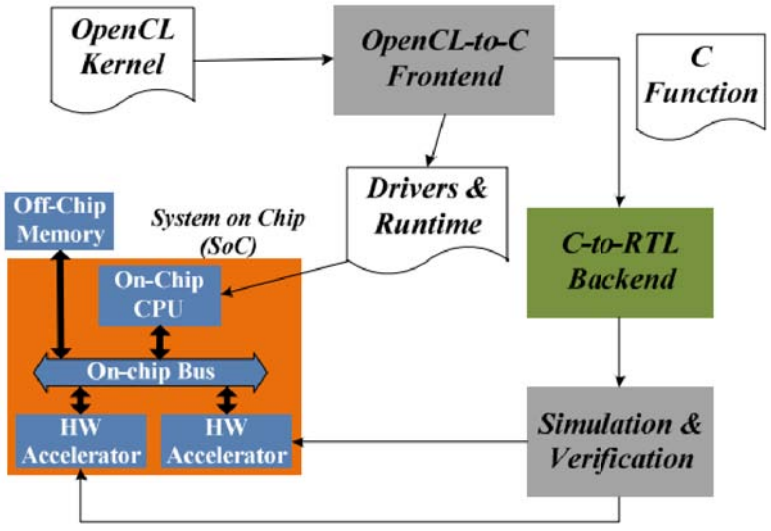

FIGURE 3.7: Silicon-OpenCL Low Level Compiler.

Figure 3.8 shows the C to RTL back end tool flow which along with the front end is based on the LLVM compiler infrastructure [18]. LLVM compiler translates the input C function into an assembly-like intermediate representation, called LLVM-IR. The LLVM compiler provides conventional optimizations and transformations such as dead code elimination, redundant code elimination, constants propagation, algebraic transformations, loop transformations, loop unroll, and loop invariant code motion. Given the LLVM-IR, the backend performs two sets of tasks, low level transformations and optimizations, and hardware allocation and generation.

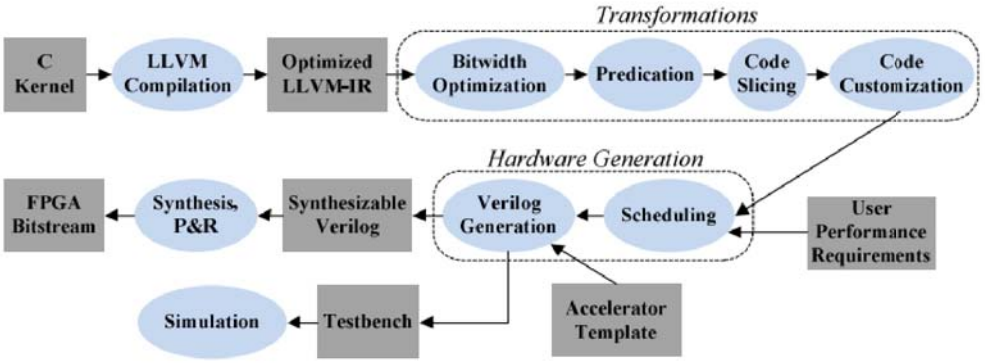

FIGURE 3.8: Silicon-OpenCL C-to-RTL backend.

3.2.1 OpenCL to C transformation

As explained in the previous section, OpenCL exposes parallelism at a fine level of granularity by allowing the programmer to embody the task executed by a single logical thread in an OpenCL kernel.Depending on performance requirements, and resource availability, any number of hardware accelerators can be generated spanning from a simple interpolator, executing a single thread per invocation, to an accelerator that produces the complete interpolated frame every time it is invoked. Between these two extremes, a hardware generation tool can generate any number of accelerators, each, potentially, being assigned a different amount of workload per invocation.

In order to enable efficient mapping of OpenCL kernel functions to the underlying platform while at the same time taking into account any hardware constraint, SOpenCL tool applies a series of source-to-source transformations in the high level compiler frontend (*Figure 3.7*) that collectively aim at coarsening the granularity of a kernel function from the work-item to the work-group level.

OpenCL-to-C frontend applies three source-to-source transformations: threads serialization, elimination of synchronization functions, and variable privatization. The end result is a C Function which consists of triple nested loops in order to emulate the execution of OpenCL kernels in an NDRange 3D index space.

The body of a triple nested loop represents the workload of a single work-item, which leads to the conclusion that multiple iterations of a triple nested loop can correspond to multiple work-items, and hence, can be executed in parallel and out of order. Furthermore, explicit local memory representations are transformed into local data arrays in the C function, and can be implemented as on-chip distributed memory blocks.

3.2.2 SOpenCL: LLVM Compiler infrastructure

Owaida [3] developed an LLVm compiler infrastructure to provide a machine independent framework for program optimization, analysis, and refactoring. To provide support for multiple programming languages and different target architectures, LLVM adapts a three-step compilation flow (*Figure 3.9*). The LLVM compiler model provides a RISC-style, yet rich, intermediate representation (LLVMIR) between the frontend, optimizer, and backend.

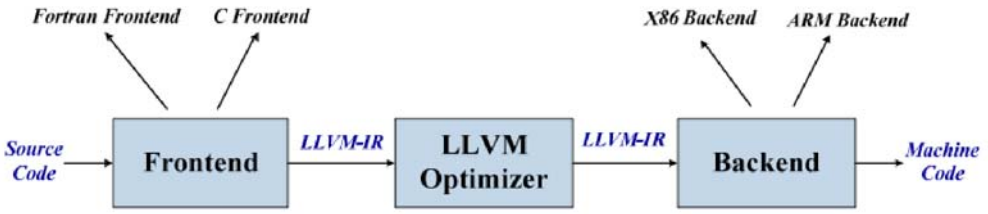

FIGURE 3.9: Silicon-OpenCL LLVM compiler infrastructure.

The clarity and completeness of the LLVM-IR, provides a simple way for conveying information between multiple analysis and transformation passes as well between the frontend and backend. Using LLVM-IR, the compiler framework is a collection of libraries of transformations and optimizations can be used to build a compiler for any language and target architecture. In particular, LLVM-IR is both well specified and the only interface to the optimizer. This property means that all one needs to know to write a frontend for LLVM is what LLVM-IR is, how it works, and the invariants it expects.

3.2.3 Example output

To demosntrate the functionality of the SOpenCL front-end we used as input a Multiple Debye-Huckel method kernel programmed in OpenCL (*listing 3.2*) the C Function which was produced can be seen in *listing 3.3*.

```
__kernel void mdh(__global float *ax, __global float *ay,
    __global float *az, __global float *charge,
    __global float *size, __global float *gx,
    __global float *gy, __global float *gz,
    __global float *val, float pre1,
    float xkappa, int natoms)
{
   int igrid = get_global_id(0);
   float v = 0.0f;
   float dx, dy, dz, dist;

   for ( int jatom = 0; jatom < natoms; jatom++ ) {
      dx = gx[igrid] - ax[jatom];
      dy = gy[igrid] - ay[jatom];
      dz = gz[igrid] - az[jatom];
      dist = sqrt(dx*dx + dy*dy + dz*dz);
      v += pre1 * (charge[jatom] / dist)
         * exp(-xkappa * (dist-size[jatom]))
         / (1.0f+xkappa*size[jatom]);
   }
   val[igrid] = v;

}
```

LISTING 3.2: MDH source code written in OpenCL.

```
1  /* Keep the compiler happy */
2  #define __kernel
3  #define __global
4  #define __local
5
6  __kernel void mdh(__global float *ax, __global float *ay,
7      __global float *az, __global float *charge,
8      __global float *size, __global float *gx,
9      __global float *gy, __global float *gz,
10     __global float *val, float pre1,
11     float xkappa, int natoms,
12     /* The arguments below were introduced by the SOPenCL front end */
13     int __global_id_x, int __global_id_y, int __global_id_z,
14     int __global_size_x, int __global_size_y, int __global_size_z,
15     int __work_group_id_x, int __work_group_id_y, int __work_group_id_z,
16     int __local_size_x, int __local_size_y, int __local_size_z)
17 {
18   unsigned int __kernel_i, __kernel_j, __kernel_k;
19   float __cl_cse8;
20   int igrid;
21   float v, dx, dy, dz, dist;
22
23   for (__kernel_k = 0; __kernel_k < __local_size_z; __kernel_k++) {
24     for (__kernel_j = 0; __kernel_j < __local_size_y; __kernel_j++) {
25       for (__kernel_i = 0; __kernel_i < __local_size_x; __kernel_i++) {
26         igrid = (__global_id_x + __kernel_i);
27         v = 0.0f;
28         for (int jatom = 0; jatom < natoms; jatom++) {
29           dx = gx[igrid] - ax[jatom];
30           dy = gy[igrid] - ay[jatom];
31           dz = gz[igrid] - az[jatom];
32           dist = sqrt(dx * dx + dy * dy + dz * dz);
33           __cl_cse8 = size[jatom];
34           v += pre1 * ( (charge[jatom] / dist) *
35             (exp(-xkappa * (dist - __cl_cse8)) /
36                 (1.0f + xkappa * __cl_cse8)) );
37         }
38         val[igrid] = v;
39       }
40     }
41   }
42
43 }
```

Listing 3.3: SOpenCL's front-end output when the MDH OpenCL kernel is used as input.

Chapter 4

Optimizing memory management on heterogeneous systems.

The target of this thesis is to optimize memory management on heterogeneous systems. Our approach involves performing memory access pattern analysis on kernels in order to produce an accurate[1] estimation of the memory usage. This information is produced in the form of array ranges describing which elements are accessed as well as whether they are read or written. Using these ranges we can statically partition the kernel and generate kernels which use fractions of the original data.

Memory in GPUs is considered small compared to system memory, however the difference is even more profound in the case of FPGAs. Thus an OpenCL kernel may not be able to execute at all on a given compute device due to device memory not being big enough to store all its working set. We propose our algorithm as a solution to this problem; by analysing the OpenCL kernel we can potentially decrease the amount of memory allocated on a single OpenCL compute device, since we detect which ranges of arrays are actually required for each sub-kernel[2] and allocate memory for them instead of the total range of data. Since we can now use more than one compute device concurrently for the execution of a single kernel we can also improve the performance of an application, much like the way a traditional multithreaded application is, usually, sped up when multiple CPUs are used instead of just one.

[1] Our memory access pattern analysis is based on polyhedral techniques. Therefore we assume that the polyhedra generated are convex which may result in inaccurate estimations. However the inaccuracy comes in the form of overestimating the number of array elements used. We believe that this is acceptable since transferring more data than the absolutely necessary may affect the performance gain of the optimization but not the overall correctness of the OpenCL kernel(s).

[2] Throughout this thesis we will refer to the kernels executing part of the total computation as sub-kernels.

Our algorithm in its core uses the Polyhedral Extraction Tool (PET) [19], ISL [20] and PolyLib [7]. PET is a library created to produce a polyhedral model from C source code. It is based on LLVM's C frontend CLANG [21] and ISL. ISL is a library for manipulating quasi-affine sets and relations, it allows for easy construction as well as powerfull and compact representation of the polyhedral model. Finaly PolyLib, a library for manipulating parameterized polyhedra, provides a rich set of functions including finding the vertices and producing Ehrhart polynomials (see *section 2.5*) of parameterized polyhedra.

Currently we are using the SOpenCL framework to produce C source code using an OpenCL kernel as input. We explained in *chapter 3* that the SOpenCL framework coarsens the granularity of an OpenCL kernel and produces C code which executes the computations that take place inside a single work-group. Using an appropriate loop hierarhy[3] the work-group can be executed on a CPU, or after a HLS pass a kernel instance can be deployed on an FPGA. In both cases the C source code will conveniently enclose the original work-item in a triple nested loop which suits the needs of polyhedral analysis. With minor modifications to the current implementation of our algorithm OpenCL kernels can be handled without making use of the coarsening simply by mapping all calls to *get_local_id(dimension), get_global_id(dimension), get_local_size(dimension), get_global_size(dimension)* and so on to (nonexistent) variable much like the SOpenCL framework operates.

4.1 Memory access pattern estimation

Starting from an unmodified OpenCL kernel we use the SOpenCL front end which outputs an equivalent valid C function. This C source code consists of a number of triple nested loops which accurately represent the execution of *work-group*'s *work-items* in a serialized manner. Since the loop iterators are induction variables[4] each triple nested loop is a great basis for a *SCoP* (*section 2.3*) which is our motivation to apply polyhedral analysis. Furthermore in order to fully take advantage of the OpenCL programming model software developers are encouraged to keep divergence to a minimum, therefore we expect the input OpenCL kernel to contain simple control flow, or none at all. This characteristic will be trasfered to the C source code by the SOpenCL front end and will allow PET to generate polyhedra that accurately represent the accesses.

[3] For example, a kernel that uses a 3 dimensional NDRange will make use of a triple nested loop; one loop structure for each dimension.

[4] A variable that gets increased or decreased by a fixed amount on every iteration of a loop, or is a linear function of another induction variable. In our case the iterators are increased by 1.

The output of the memory access pattern analysis algorithm is a set of ranges for all array accesses[5] in the original OpenCL kernel as well N OpenCL kernels which are modified versions of the original kernel and each one completes a fraction of the original computation workload. These N kernels can be executed concurrently by different compute devices. Finally, our tool also provides information about accesses on which polyhedral analysis cannot be performed, because they cannot be placed in a valid SCoP.

4.1.1 Algorithm

Our memory access estimation algorithm starts with an OpenCL kernel used as input to the SOpenCL frontend which will create a C function that performs the computation workload of a single work-group. We use PET [19] to parse the C file and create the polyhedra which represent the accesses to memory arrays. During the analysis we keep track of arrays which are involved in accesses that cannot be represented by polyhedra. If, at any time, we find another access that involves the aforementioned arrays, we do not create any additional polyhedra. Since we cannot represent all accesses to an array we do not want to use incomplete information when we perform our memory analysis algorithm because we would produce highly inaccurate results which could certainly effect the correctness of the final kernels. We have to make the assumption that all elements of such arrays are used in all the sub-kernels we produce. For example consider the following SCoP:

```
1 for (i=0; i<N; ++i) {
2     for (j=0; j<M; ++j ) {
3         c += a[i];        // S1
4         c += a[i*j];      // S2
5     }
6 }
```

Listing 4.1: Example illegal array access.

Even though S1 is a perfectly legal access we have to discard it because S2 cannot be described using the polyhedral model, since $i * j$ is not an affine expression. Had we done otherwise but discard all information about accesses to array a we would not be able to know that elements of a whose index is beyod N are accessed. This is due to the fact that S2 accesses elements up to index $(N-1)*(M-1)$. Therefore during the execution of the kernel we would falsely assume that transferring to the compute device only the elements with index $[0, N)$ is perfectly fine.

[5] A single statement may involve multiple array accesses, for example a[i] = a[i*2] + a[j+N] .

The core of the memory access pattern analysis procedure takes place after PET has finished producing the polyhedra which represent accesses to memory. We begin by forcing the polyhedra to include, in the form of dimensions, all of the parameters generated by the SOpenCL frontend. After asserting that all dimensions are present, we inject the following constraints to describe relations between these (possibly new) dimensions that are not apparent from the parsed C source code:

$$
\begin{aligned}
index &\geq 0 & local\_size(0) &\geq 1 & local\_size(1) &\geq 1 \\
local\_size(2) &\geq 1 & global\_size(0) &\geq 1 & global\_size(1) &\geq 1 \\
global\_size(2) &\geq 1 & global\_id(0) &\geq 0 & global\_id(1) &\geq 0 \\
global\_id(2) &\geq 0 & & & & \\
global\_id(0) &\leq global\_size(0) - local\_size(0) & & & & \\
global\_id(1) &\leq global\_size(1) - local\_size(1) & & & & \\
global\_id(2) &\leq global\_size(2) - local\_size(2) & & & &
\end{aligned}
$$

or equivalently

$$
\left\{ \begin{pmatrix} index \\ local\_size(0) \\ local\_size(1) \\ local\_size(2) \\ global\_size(0) \\ global\_size(1) \\ global\_size(2) \\ global\_id(0) \\ global\_id(1) \\ global\_id(2) \end{pmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & -1 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right\}
$$

The next step of the algorithm is to remove all unnecessary constraints from the polyhedra. This allows for easily detecting whether an access depends on a specific variable as well as reducing the time required to produce the vertices of the polyhedra. A non-essential dimension may be introduced either because it was used in the context of the

access in the enveloping loop structures or because we inserted it during the earlier process of injecting constraints. Removing the unnecessary dimensions involves recursively identifying them and then projecting them out using the Fourier-Motzkin elimination process [22]. The initial set *essential-dims* gets populated by inserting all the dimensions with non-zero coefficients in the equality representing the index of the access. We then iterate through all remaining (in)equalities of the polyhedron. If an (in)equality is a combination of K dimensions of *essential-dims* and L dimensions which are not in the set we insert the L dimensions in it. This process is repeated every time *essential-dims* increases in size. Finally, we project out all dimensions but those listed in the set.

At this point we can tell if a polyhedron can be split. Currently our tool splits the total workload in one dimension, namely X. This is done simply by polling the polyhedron for the dimension global_id(0). If it is present we know that the access depends on the X-axis coordinate of the point describing a thread in the NDRange. If global_id(0) is not present in the polyhedron we get an estimation of the number of the elements which are accessed as well as the range of their indices by computing the Ehrhart polyhedron and the polyhedron's vertices. We start this process by removing all dimensions which are mapped to a variable (*section 2.3*) leaving only the virtual variable *index* which is equal to the linear combination of variables and parameters that form the index of the array access. Essentially, this enables us to get the lower and upper bounds of the elements been read/written by this specific access simply by finding the possibly parametric vertices of the polyhedron. During execution we will use this information to transfer the respective array segments between the host and all computing devices since accesses on them do not depend on the work-items' position in the NDRange.

In the event that an array index involves global_id(0), before we partition the NDRange space into N smaller ones we get an estimation of the total number of elements been accessed using the methodology we described earlier[6]. Even though we currently only partition the kernel in 1D, our methodology can be easily modified to support any partition which can be represented by a polyhedron, for example 2D as well as 3D partitioning schemes; these are illustrated in *table 4.1*. Splitting the polyhedron into smaller parts is achieved by injecting constraints, more specifically inequalities. For example these inequalities would have to be inserted in the polyhedron to generate the 3rd out of 5 sub-polyhedra:

Additional Inequalities =

[6]This information is used mostly by the software developer to get an estimation of the application's total memory footprint.

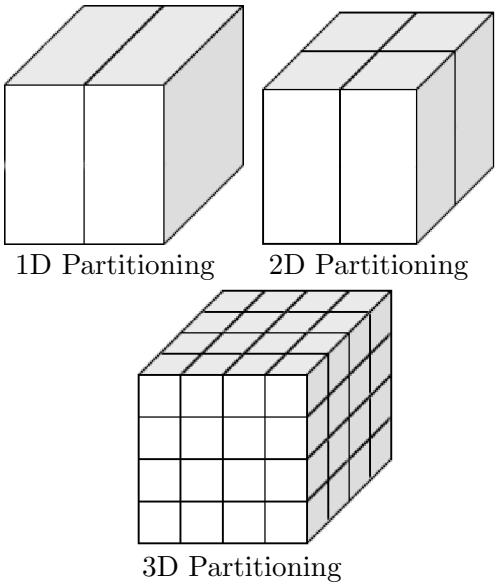

TABLE 4.1: 1D, 2D, and 3D partioning schemes.

$$global\_size(0) \geq \frac{2}{5} * (global\_size(0) - local\_size(0))$$
$$global\_size(0) < \frac{3}{5} * (global\_size(0) - local\_size(0))$$

$$\text{or equivalently } \left\{ \begin{pmatrix} global\_id(0) \\ global\_size(0) \\ local\_size(0) \end{pmatrix} * \begin{bmatrix} 5 & -2 & 2 \\ -5 & 3 & -3 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 5 \end{bmatrix} \right\}$$

After producing a sub-polyhedron representing a fraction of the total access, we use the process described above to produce the Ehrhart polynomials which is an estimation of how many elements are accessed as well as the range of elements' indices in the form of bounds (element_address $[low, high]$). We will use this range to rewrite the access so that we can allocate less memory in the OpenCL compute device. The rewritten access will be of the form $array[original\_index - low\_bound + offset]$ where $offset$ is the sum of the Ehrhart polynomials for all previous accesses to this array [7].

The pseudocode for our algorithm can be seen in *snippet 4.2*, the process is also illustrated in *figure 4.1*.

[7] Currently this last step is performed manually. In the future we plan to automate it using an additional CLANG pass after the polyhedral analysis.

```
1  void memory_access_pattern
2      (
3      OpenCLKernel* kernel,      /* Perform analysis on this kernel */
4      Polyhedron *accesses,      /* From PET*/
5      unsigned int slices,       /* Into how many parts will we
6                                    split the polyhedra */
7      Range* ranges              /* Lower/Upper bounds for every access */,
8      OpenCLKernel *subkernels /* The actual kernels that will be executed
9                                    on the available compute devices */
10     )
11 {
12     int i, j;
13     Polyhedron access, temp;
14     Range r;
15     Ehrhart e;
16
17     for ( i = 0; i < size(accesses); ++i ) {
18         access = assert_dimensions(accesses[i]);
19         access = inject_initial_constraints(access);
20         access = minimize(access);
21         if ( can_split(access) == true ) {
22             /* Calculate the total memory footprint of this access like if
23             it were to be executed on a single device */
24             temp = project_out_vars(access);
25             polyhedron_analyse(temp, &r, &e);
26             ranges = insert_total_range(ranges, access.name, r, e);
27             for ( j = 0; j < slices; ++j ) {
28                 /* Slice the polyhedron into smaller parts
29                 and produce the upper/lower bounds describing
30                 which elements are actually accessed */
31                 temp = split_access(access, j, slices);
32                 temp = project_out_vars(temp);
33                 polyhedron_analyse(temp, &r, &e);
34                 ranges = insert_subrange(ranges, access.name, r, e);
35             }
36         } else {
37             temp = project_out_vars(access);
38             polyhedron_analyse(temp, &r, &e);
39             ranges = insert_total_range(ranges, access.name, r, e);
40         }
41     }
42     /* After the analysis, the subkenels are generated */
43     subkernels = generate_subkernels(kernel, ranges, slices);
44 }
```

Listing 4.2: Example: Matrix multiplication in OpenCL.

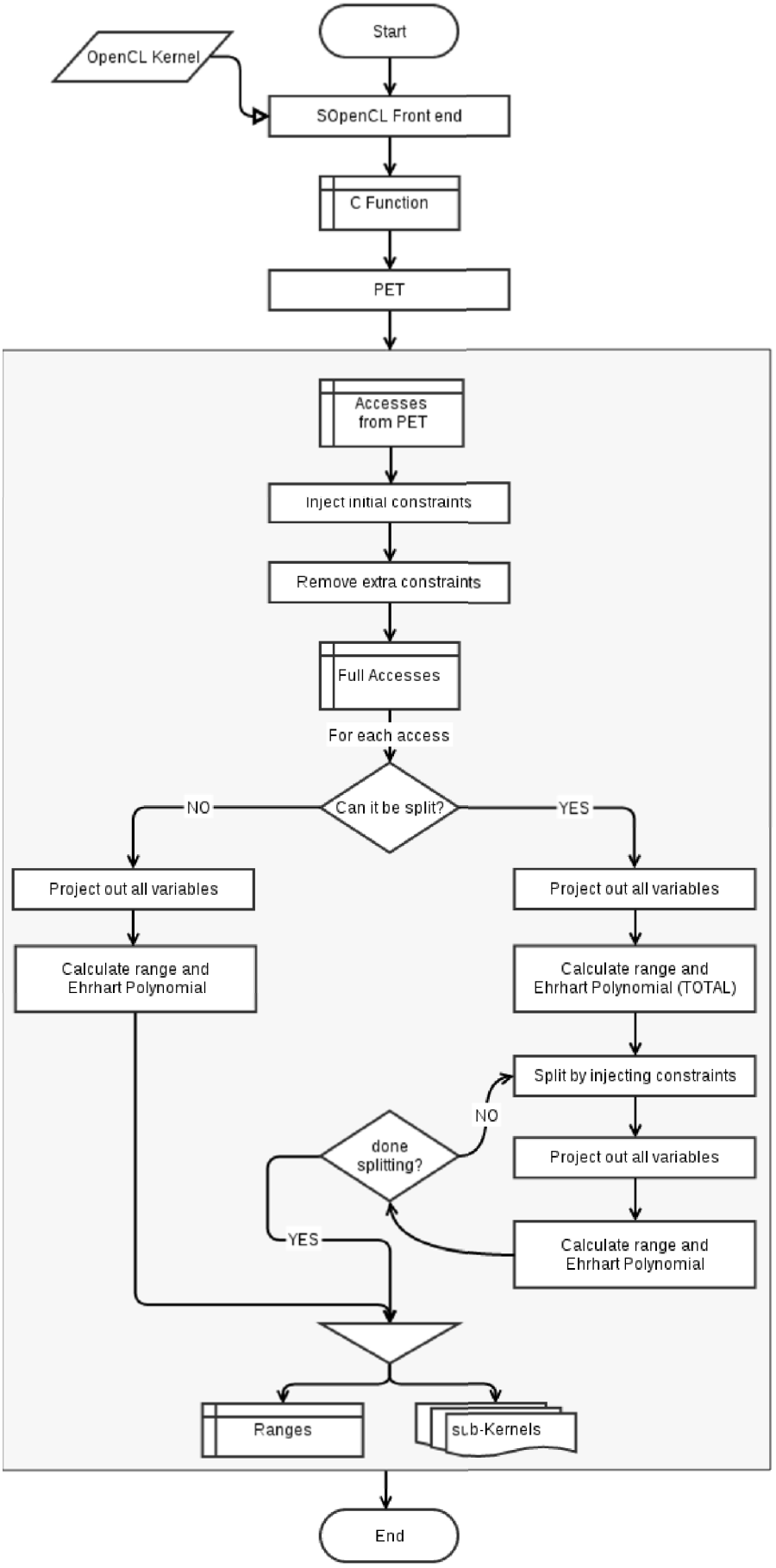

FIGURE 4.1: Flow chart of the algorithm.

4.2 Execution

The array element ranges as well as the Ehrhart polynomials which were generated during our memory access pattern analysis need to be accessed by the host device just before the launch of the generated OpenCL kernels. This is essential in order to allocate memory for the parameters on the OpenCL compute devices and to transfer the respective data. Therefore, we developed a solution to store and recall the ranges/ehrhart-polynomials based on the Flex[23]/Bison[24] toolchain.

Furthermore, after the execution of a kernel, data may need to be rearranged in order to get the same output as if the OpenCL kernel was executing on a single compute device. This may happen if the output of a sub-kernel overlaps with that of another instance. Our tool can easily detect such scenaria simply by testing the union of the polyhedra representing the write-accessed index ranges for a given array. If an overlap exists, i.e the union of the respective polyhedra contains more than 0 integer units, our tool will output information indicating which arrays may need to be rearranged after the execution of the kernels on the multiple compute devices. The user is then responsible to construct the final output of his kernel by properly "gluing" together the outputs of the sub-kernels.

4.3 Vector add example

We will use the trivial example of adding two vectors in order to demonstrate our methodology. The simplest OpenCL kernel which performs the computation can be seen in *listing 4.3*. Its output is an array c, the sum of two vectors which are passed to the kernel via the parameters a and b; all arrays three are of size N.

```
1  __kernel void vector_add(__global int *c,
2      __global const int* a, __global int *b,
3      unsigned int N)
4  {
5      int id = get_global_id(0);
6
7      if ( id < N ) {
8          c[id] = a[id] + b[id];
9      }
10 }
```

LISTING 4.3: Example: Vector addition in OpenCL.

Accesses to arrays *a*, *b*, and *c* are identical, thus they are described by the same polyhedron. We have purposely omitted the detailed process of generating the polyhedra because a more complex example depicting the weakness of our algorithm will be presented in more detail in *section 4.5.1*. Pet generates the following polyhedron for the accesses in the *vector_add* kernel:

$$\left\{ \begin{pmatrix} index \\ local\_size(0) \\ N \\ global\_id(0) \\ global\_size(0) \end{pmatrix} \quad \mathbb{Z}^5, \begin{pmatrix} index \\ local\_size(0) \\ N \\ global\_id(0) \\ global\_size(0) \end{pmatrix} * \begin{bmatrix} -1 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ -1 & 1 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & -1 & 0 & -1 & 1 & 1 \end{bmatrix} \geq \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ -1 \end{bmatrix} \right\}$$

The output of our algorithm correctly identifies that the array accessess of the kernel before the partitioning involve the elements whose indices fall in the range $[0, N-1]$. After slicing the kernel in half on dimesion X, we receive two kernels. The first includes all work-items with $global\_id(0) \quad \left[0, \frac{global\_size(0)}{2}\right)$ while the second encloses work-items with $global\_id(0) \quad \left[\frac{global\_size(0)}{2}, global\_size(0)\right)$. All arrays in the first kernel access elements whose indices belong in the range $\left[0, \frac{global\_size(0)}{2}\right)$, whereas in the second kernel the elements in range $\left[\frac{global\_size(0)}{2}, N\right)$ are accessed. The ehrhart polynomials associated with the accesses are $(\frac{1}{2} * global\_size(0) + [0, -\frac{1}{2}] * global\_size(0))$ and $(N + (-\frac{1}{2} * global\_size(0) + [0, -\frac{1}{2}] * global\_size(0)))$ respectively.

4.4 Hybrid numerical PDE solver

A partial differential equation is a differential equation which contains unknown multivariable functions and their partial derivatives. PDEs are used to formulate problems involving functions of several variables, and are either solved by hand, or used to create a relevant computer model.

We evaluated our algorithm by using it to optimize an application which solves PDEs using a monte-carlo approach in order to reduce the complexity of the problem. It is an attempt to provide better results for multi-physics, multi-domain PDE problems.

A solution of a PDE is generally not unique; additional conditions must generally be specified on the boundary of the region where the solution is defined. This hybrid methodology attempts to exploit this by splitting the initial problem into smaller ones whose boundaries lie within the original one's. However, in order to produce solutions

for the new set of problems conditions have to be specified for each point on the new problems' boundaries. This is where the monte carlo approach of the application comes into place. First the original problem is split into smaller non-overlapping regions. Then for each point on the boundary of each region a bumber of random walks is performed which generates a weighted sum. This sum will be used as the boundary condition for the point of which the paths originated. After this preprocessing step, all generated PDEs are solved and then the total solution is constructed by merging the produced solutions to the set of smaller PDEs.

We will use an (imaginary) application which performs weather forecasting to present a usage scenario. Currently, in order to produce an estimation of the weather conditions on a given region a single, quite large, PDE set is used. The boundary conditions are supplied after measurements take place on the borders of the region (*figure 4.2*). Afterwards, the PDE is solved and the results are used to produce the weather forecast. However, this is a really expensive process time-wise. Additionaly, due to the size of the region there is inherent heterogeneity which may result in inaccurate results.

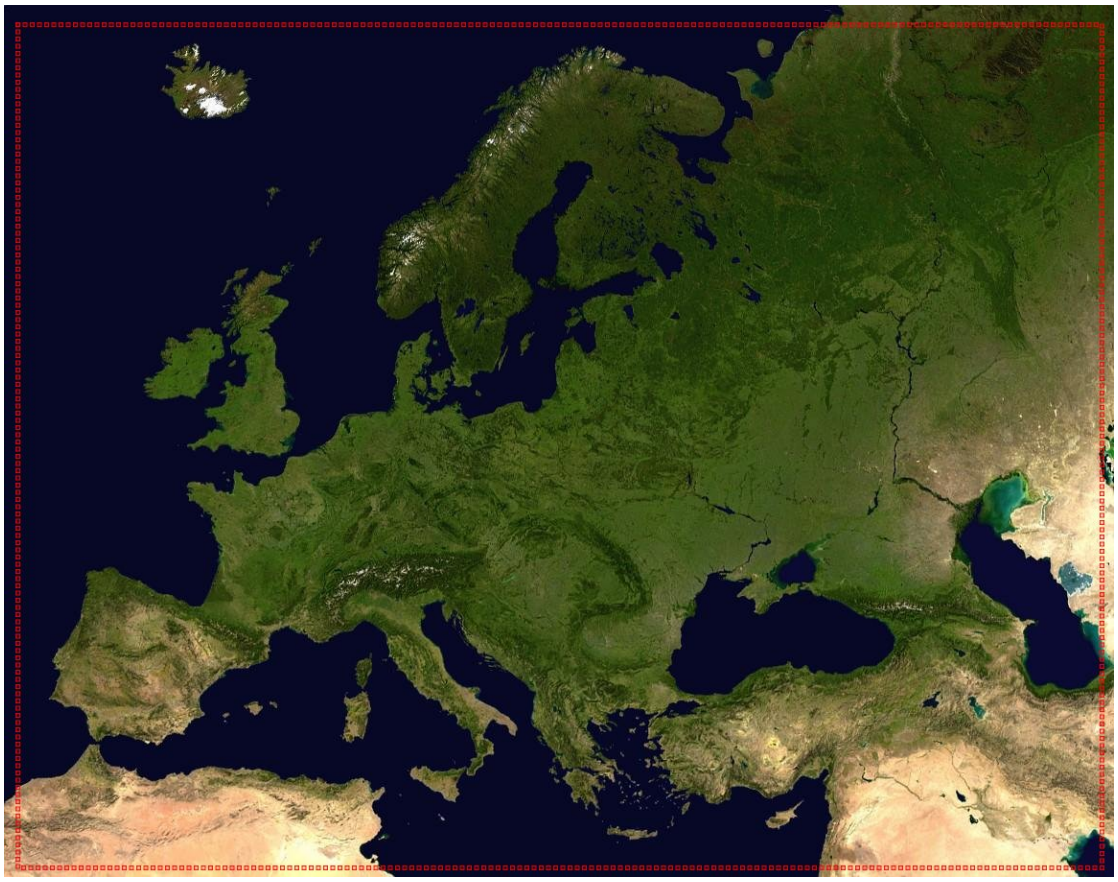

FIGURE 4.2: Boundaries on the large region.

Imagine that the red dots in *figure 4.2* indicate the boundary conditions for a PDE which will be used to generate the weather report for the whole continent of Europe. However

we plan to go for vacations in Greece, therefore we wish to know the weather forecast for a region around Greece just in case we want to visit the surrounding countries as well. This train of thought would result in something similar to what is depicted in *figure 4.3*.

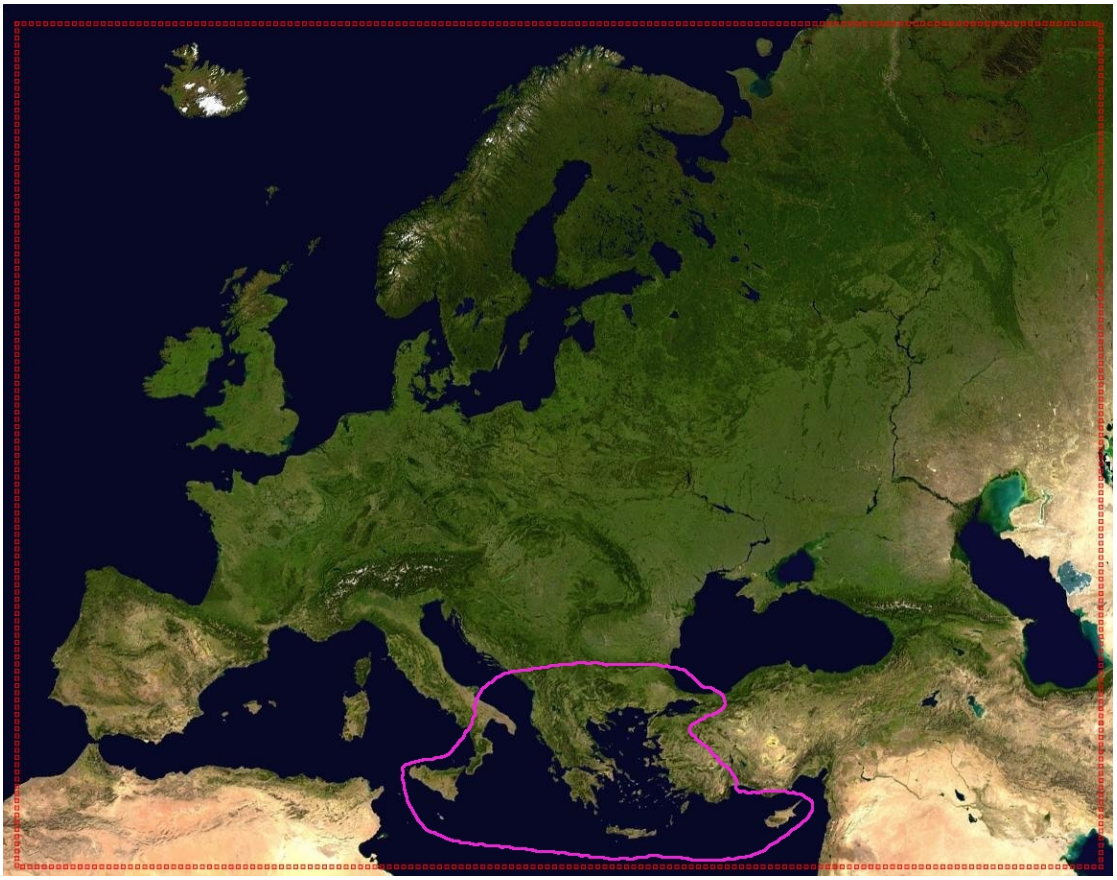

FIGURE 4.3: Boundaries on the smaller region.

Then for each point on the purple boundary a number of random walks are performed. The idea is that the random path begins from a point on the boundary of the region and stops when it reaches the boundary of the original problem. Each path performed creates a number which is an estimation of the boundary condition on the origin of the path. The average of all the generated values for a given point is the final boundary condition for the new PDE (*figure 4.4*)[8].

[8]Note that this is not an entirely realistic scenario, since we are including both land and water as well as islands in the same region; each is described by a whole different PDE. However, we use this figure just for the sake of illustrating the intuition behind the monte carlo approach.

FIGURE 4.4: Random walks, starting from one point on the boundary of the marked region. This would be performed for a number of points on the boundary and the remaining ones would be generated via means of interpolation.

4.4.1 Hybrid numeric PDE solver test case evaluation

In the OpenCL implementation of the algorithm, each point on the boundary is mapped to the execution of an OpenCL work-group. As such, each work-item computes a subset of the total number of random walks. When the number of explored paths reaches the maximum number the intermediate results are averaged in order to produce the final estimation for the paths's point of origin.

Our tests showed that porting the application to OpenCL resulted in an average speedup[9] of 15x allowing us to execute experiments in less than 7 hours on a GPU (*GeForce GTX 480 @1401 MHz*) which would execute for up to 10.8 days on a CPU (*Intel(R) Core(TM) i7 CPU 870 @ 2.93GHz*).

As a preliminary step to applying our methodology we profiled the application's execution using OpenCL on the CPU and GPU. Our experiments indicated that the optimum setup would be to execute, concurrently, 86% of the random walks on the GPU and the

[9]Note that the execution time is greatly affected by the nature of the experiment.

rest on the CPU. This allowed us to achieve up to 1.3x speedup over the GPU-only implementation, in other words a 18.5x speedup over the CPU-only implementation. We used our algorithm to determine which parts of the arrays had to be transferred to the two sub-kernels. To do this, we only had to plug in our tool a set of constraints so that the kernel partitioning scheme would produce 2 sub-kernels, one computes 86% of the total workload and the other the remaining 16%. The speedup comes from executing on both available devices concurrently since we can guarantee that all essential data wil be available to the respective compute device during run-time.

4.5 Validation

Our tool performs memory access pattern analysis which, in turn, enables partitioning the total computation workload into smaller chunks and executing them concurrently on multiple compute devices. This benefits the software developer in multiple ways:

- The application's performance increases.
- If a proper partition scheme is applied, then kernels which could not execute on a single device due to hardware memory size limits they can now be sliced into multiple subkernels which operate on a subset of the original data and subsequently can be executed on multiple devices concurrently.
  - Additionally, the SOpenCL toolchain allows us to execute such sub-kernels on CPUs and FPGAs with minimal hassle.

During the compilation of this thesis we have ported a multithreaded application which uses posix threads to the OpenCL programming model[10]. The application is a hybrid partial differential equation solver. It is based on the idea that solving large problems is a time and memory costly process which can be sped up by partitioning the original problem into smaller ones. By performing a large number of random walks, starting from integral points to the original problem, the application constructs a set of smaller problems whose solutions, when combined, form a close approximation to the solution of the large problem. Using our methodology we were able to get a speedup of $1,3$x by executing the kernel on both GPU (*GeForce GTX 580 @ 1564MHz*) and CPU (*Intel(R) Xeon(R) CPU E5645 @ 2.40GHz*).

Using the example in *section 4.3* as another test case, we exploit the results of the automatic analysis to execute the first subkernel on a GPU (*GeForce GTX 480 @1401*

[10]https://github.com/mvavalis/Hybrid-numerical-PDE-solvers/

MHz), and the second one on a CPU (*Intel(R) Core(TM) i7 CPU 870 @ 2.93GHz*). Using our tool to execute concurrently on CPU and GPU resulted in a $1,74x$ speedup over the execution of the original kernel. Furthermore, since memory is split over both devices we can execute kernels with bigger datasets. In this particular example we doubled the size of the original problem resulting in vectors with $24,576,000$ integers and utilized both CPU and GPU. The execution time was the same as adding vectors of size 12288000 using just the GPU.

However, our experiments have shown that the partitioning scheme greatly affects the memory requirements for the sub-kernel instances. If a partitioning scheme is not appropriate for the kernel at hand, it may make poor use of the devices's memory. In *section 4.5.1* we discuss an example to highlight the weaknesses of polyhedral analysis, as well as the importance of properly partitioning the OpenCL kernel.

4.5.1 A pathologic example

We present below a scenario using a Matrix Multiplication OpenCL kernel. The index of element stored in position $(row, column)$ in a matrix with dimensions $width * height$ is $row*width+column$. One may notice that this index is not an affine combination of *row*, *width*, and *column*, which forbids polyhedral analysis from being applied. However, as long as we replace *width* with a fixed constant value we can use polyhedral analysis. We use this trick to write a matrix multiplication kernel for matrices with dimensions 1000 x *height* stored as an 1D array. Note that *height* can be a parameter since, unlike *width*, it does not lead to any non affine expressions.

```
1  __kernel void matrixMultiply(__global float *c,
2      __global const float* a, __global const float *b)
3  {
4      int x = get_global_id(0);
5      int y = get_global_id(1);
6      float element = 0;
7      int i;
8
9      for (i=0; i<1000; i++) {
10         element += a[i + y*1000] * b[x +i*1000];
11     }
12     c[x + y*1000] = element;
13 }
```

Listing 4.4: Example: Matrix multiplication in OpenCL.

On a closer look one will notice that the dimensions of the matrices are hardwired in the source code, 1000 x *height*. If there was an argument specifying the width and height we would not be able to perform polyhedral analysis, since both statements $element+ = a[i+y*1000]*b[x+i*1000]$; and $c[x+y*1000] = element$; would involve array accesses with non affine indices.

We begin by using our source-to-source compiler to transform this OpenCL kernel into C; the output is listed in *snippet 4.5*. We will then use this source code as input to our memory access pattern analysis algorithm in order to retrieve the memory ranges being used and the OpenCL sub-kernels. For the sake of demonstrating the process we will have the algorithm partition the kernel into two sub-kernels.

```
1  __kernel void matrixMultiply(__global float *c,__global const float* a,
2      __global const float *b,
3      /* The arguments below were introduced by the SOPenCL front end */
4      int __global_id_x, int __global_id_y, int __global_id_z,
5      int __global_size_x, int __global_size_y, int __global_size_z,
6      int __work_group_id_x, int __work_group_id_y, int __work_group_id_z,
7      int __local_size_x, int __local_size_y, int __local_size_z)
8  {
9      unsigned int __kernel_i, __kernel_j, __kernel_k;
10     unsigned int x, y,i;
11     float element;
12
13     for (__kernel_k = 0; __kernel_k < __local_size_z; __kernel_k++) {
14         for (__kernel_j = 0; __kernel_j < __local_size_y; __kernel_j++) {
15             for (__kernel_i=0; __kernel_i<__local_size_x; __kernel_i++) {
16                 x = (__global_id_x + __kernel_i);
17                 y = (__global_id_y + __kernel_j);
18                 element = 0;
19                 for (i = 0; i < 1000; i++) {
20                     element += a[i + y * 1000] * b[x + i * 1000];
21                 }
22                 c[x + y * 1000] = element;
23             }
24         }
25     }
26 }
```

Listing 4.5: Example: Matrix OpenCL kernel after OpenCL-to-C transformation.

First, PET parses the C source code and constructs the initial polyhedral model for the example C source code. The results can be seen in *table 4.2*. After we inject the initial constraints and finish minimizing the polyhedra we know that the memory access

involving array a does not depend on $global\_id(0)$ but accesses to b and c do. Thus, we receive the ranges describing the total access to a, b, and c as well as the element ranges after the partition is performed for accesses to arrays b and c; this information is summarized in *tables 4.3* and *4.4* respectively.

Read/ Write	Array Name	Constraints
R	a	$i + 1000 * global\_id(1) + 1000 * \_\_kernel\_j \geq 0$
		$0 \leq i \leq 999$
		$0 \leq \_\_kernel\_k < local\_size(2)$
		$0 \leq \_\_kernel\_j < local\_size(1)$
		$0 \leq \_\_kernel\_i < local\_size(0)$
R	b	$1000 * global\_id(0) + \_\_kernel\_i \geq 0$
		$0 \leq i \leq 999$
		$0 \leq \_\_kernel\_k < local\_size(2)$
		$0 \leq \_\_kernel\_j < local\_size(1)$
		$0 \leq \_\_kernel\_i < local\_size(0)$
W	c	$\_\_kernel\_i + 1000 * global\_id(1) + global\_id(0) + 1000 * \_\_kernel\_j \geq 0$
		$0 \leq \_\_kernel\_k < local\_size(2)$
		$0 \leq \_\_kernel\_j < local\_size(1)$
		$0 \leq \_\_kernel\_i < local\_size(0)$

TABLE 4.2: Example: Initial polyhedral model.

Read/ Write	Array Name	Ranges	Ehrhart
R	a	$[0, 1000 * global\_size(1) + 999]$	$1000 * global\_size(1) + 1000$
R	b	$[0, global\_size(0) + 999000]$	$global\_size(0) + 999001$
W	c	$[0, global\_size(0) + 1000 * global\_size(1)]$	$global\_size(0) + (1000 * global\_size(1) + 1)$

TABLE 4.3: Example: Element ranges of arrays.

Essentially, the information conveyed by *tables 4.3 and 4.4* is that the unpartitioned accesses to the arrays cover all of their elements[11]. Moreover, after the partition step both subkernels read all elements of matrix b and appear to write overlapped regions of matrix c. The latter occurs because we have partitioned the original workload on one dimension, namely x.

[11] The kernel is executed using the tuple $< 1000, matrix\_height, 1 >$ as its NDRange.

Read/ Write	Array Name	Ranges	Ehrhart
R	$(b_1)_x$	$\left[0, \frac{global\_size(0)}{2} + 998999\right]$	$\frac{1}{2} * global\_size(0) + \left[999000, \frac{1997999}{2}\right]\_global\_size(0)$
R	$(b_2)_x$	$\left[\frac{global\_size(0)}{2}, global\_size(0) + 998999\right]$	$\frac{1}{2} * global\_size(0) + \left[999000, \frac{1997999}{2}\right]\_global\_size(0)$
W	$(c_1)_x$	$\left[0, \frac{global\_size(0)}{2} + 1000 * global\_size(1) - 1\right]$	$\frac{1}{2} * global\_size(0) + [(1000 * global\_size(1))$ $, (1000 * global\_size(1) + -\frac{1}{2})]\_global\_size(0))$
W	$(c_2)_x$	$\left[\frac{global\_size(0)}{2}, global\_size(0) + 1000 * global\_size(1) - 1\right]$	$\frac{1}{2} * global\_size(0) + [(1000 * global\_size(1) + 0)$ $, (1000 * global\_size(1) + -\frac{1}{2})]\_global\_size(0)$
R	$(a_1)_y$	$[0, 500 * global\_size(1) - 1]$	$500 * global\_size(1)$
R	$(a_2)_y$	$[500 global\_size(1), 1000 global\_size(1) - 1]$	$500 * global\_size(1)$
W	$(c_1)_y$	$[0, global\_size(0) + 500 * global\_size(1) - 1000]$	$global\_size(0) + (500 * global\_size(1) + -999)$
W	$(c_2)_y$	$[500 * global\_size(1), global\_size(0) + 1000 * global\_size(1) - 1000]$	$global\_size(0) + (500 * global\_size(1) + -999)$

TABLE 4.4: Example: Element ranges of arrays affected by the partition. Top: partition on dimension X, bottom partition on dimension Y.

Bibliography

[1] OpenMP Architecture Review Board, July 2011. URL http://www.openmp.org/mp-documents/OpenMP3.1.pdf.

[2] KHRONOS Group. OpenCL application program interface version 1.2, November 2012. URL http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[3] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, and Christos D Antonopoulos. Synthesis of platform architectures from opencl programs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 186–193. IEEE, 2011.

[4] Motzkin, Theodore S and Raiffa, Howard and Thompson, GL and Thrall, Robert M. The double description method. 1953.

[5] NV Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear equations. *USSR Computational Mathematics and Mathematical Physics*, 4(4):151–158, 1964.

[6] Vincent Loechner and Doran K Wilde. Parameterized polyhedra and their vertices. *International Journal of Parallel Programming*, 25(6):525–549, 1997.

[7] Vincent Loechner. Polylib, February 2010. URL http://icps.u-strasbg.fr/polylib/.

[8] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, 2011.

[9] Jan Sjödin, Sebastian Pop, Harsha Jagasia, Tobias Grosser, Antoniu Pop, et al. Design of graphite and the polyhedral compilation package. In *GCC Developers' Summit*, 2009.

[10] 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*, pages 7–16. IEEE Computer Society, 2004.

[11] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, pages 283–303. Springer, 2010.

[12] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(5):773–815, 2000.

[13] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28 (5):469–498, 2000.

[14] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Analytical computation of ehrhart polynomials: Enabling more compiler analyses and optimizations. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 248–258. ACM, 2004.

[15] E Ehrhart. Polynôme arithmetique et méthode des polyedres en combinatoire, 35 serie isnm, 1977.

[16] NVIDIA, August 2009. URL http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf.

[17] Konstantis Daloukas, Christos D Antonopoulos, and Nikolaos Bellas. GLOpenCL: OpenCL support on hardware-and software-managed cache multicores. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 15–24. ACM, 2011.

[18] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[19] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT12), Paris, France*, 2012.

[20] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software–ICMS 2010*, pages 299–302. Springer, 2010.

[21] CLANG: a c language family frontend for LLVM. URL http://clang.llvm.org/.

[22] H Paul Williams. Fourier-motzkin elimination extension to integer programming problems. *Journal of combinatorial theory, series A*, 21(1):118–123, 1976.

[23] GNU. flex: The fast lexical analyzer, . URL http://flex.sourceforge.net/.

[24] GNU. Bison - GNU parser generator, . URL http://www.gnu.org/software/bison/.