FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automatic Selection of Software Code Regions for Migrating to GPUs

Fábio Gaspar

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. João Cardoso

March 11, 2022

Automatic Selection of Software Code Regions for Migrating to GPUs

Fábio Gaspar

Mestrado em Engenharia Informática e Computação

March 11, 2022

Abstract

The demand for computational power is higher than ever. With the increasing challenges of packing more transistors in a single chip and growing manufacturing costs, new approaches are explored to improve performance and reduce energy consumption. One emerging trend is heterogeneous computing systems that combine multiple processing units, including hardware accelerators. Accelerators excel in specific workloads, delivering top performance and energy efficiency. Given the heterogeneity of such systems, programming them reveals several challenges to developers: *(a)* know different tools and programming models, *(b)* deep architectural understanding of available accelerators, *(c)* manually modify existing applications code, identifying hotspots and determine which accelerator is most suited for each (intuition, profiling, etc.), and *(d)* re-write code using one or mode different programming models. Consequently, taking advantage of heterogeneous systems yields a lot of burden on developers and researchers, slowing down the adoption.

Our work presents a framework that automatically identifies hotspots in sequential C code and adapts it for running in a selected target — CPU parallel or GPU. Our approach uses static analysis to characterise the hotspot and analytical models to estimate execution times, guiding the optimal target selection. Our framework uses source-to-source techniques to insert OpenMP directives for running the hotspot in the selected target. If the analytical models estimate no benefit in parallelising or offloading the hotspot, the code is not modified to run sequentially in the CPU.

The framework has two major parts:*(a)* performance modelling, and *(b)* automatic parallelisation and offloading. Our evaluation uses existing benchmark suites with different versions for the same kernel: sequential and manually annotated with OpenMP. Firstly, we evaluate the analytical models by analysing the OpenMP versions and comparing the estimated optimal target to real-world measurements. We show that characterising the hotspot at the Abstract Syntax Tree (AST) level yields good results for a relative performance context, achieving 9 optimal decisions for 12 examples. Despite 2 slowdowns, the impact is negligible, adding fractions of a second to baseline execution time. Secondly, we evaluate our methodology for automatic parallelisation and offloading using OpenMP. We compare the achieved speedups of our automated approach to the manually annotated versions in the benchmark suites. We achieved a 7.9× geometric mean speedup, reducing the total execution time from 150.7 to 23.9 minutes.

Keywords: Automatic Parallelisation, Automatic Offloading, Performance Modelling, Static Analysis, Heterogeneous Systems, CPU, GPU

Acknowledgements

First and foremost, I want to express my gratitude to my supervisor Professor João Cardoso, for the opportunity and continuous support throughout this dissertation.

I would also like to thank Professor João Bispo and the SPeCS laboratory at FEUP for the help and welcoming environment, and my friends, especially those at the IEEE University of Porto Student Branch, for the friendship, good memories and knowledge-sharing environment.

Finally, I want to thank my family. This accomplishment would not be possible without them.

Fábio Gaspar

Contents

List of Figures

List of Tables

Abbreviations

ALU	Arithmetic Logic Unit
API	Application Programming Interface
AST	Abstract Syntax Tree
CC	Compute Capability (NVIDIA)
CFG	Control Flow Graph
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CWP	Compute Warp Parallelism
DRAM	Dynamic Random-Access Memory
EOL	End-Of-Life
FP	Floating-Point
FPU	Floating-Point Unit
FPGA	Field-Programmable Gate Array
GCC	GNU Compiler Collection
GDDR	Graphics Double Data Rate
GPGPU	General Purpose Graphics Processing Units
GPU	Graphics Processing Units
HPC	High Performance Computing
HT	Hyper-Threading
IACA	Intel Architecture Code Analyzer
ILP	Instruction Level Parallelism
IPC	Instructions Per Clock
IR	Intermediate Representation
ISA	Instruction Set Architecture
LD	Load memory operation
LLVM	Low Level Virtual Machine
MWP	Memory Warp Parallelism
OpenCL	Open Computing Language
OOO	Out-of-Order
OpenACC	Open Accelerators
OpenMP	Open Multi-Processing
RAM	Random Access Memory
RCU	Retire Control Unit
ROB	Reorder Buffer
SA	Simulated Annealing
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
SM	Streaming Multiprocessor
SMT	Simultaneous Multi-Threading
ST	Store memory operation
TPU	Tensor Processing Unit
VRAM	Video Random Access Memory

Chapter 1

Introduction

1.1 Context and Motivation

Over the last decades, the chip manufacturing industry has driven performance and energy efficiency improvements by continuously advancing lithography processes that reduce transistor size and increase density. Gordon Moore observed performance doubled approximately every two years, while maintaining the manufacturing cost, power consumption and die area [62]. However, manufacturing costs have been rising rapidly, and the advances in lithography are constrained as the transistors reach atomic scale [68].

Heterogeneous systems are emerging as one possible solution to respond to the growing demand for performance. These systems combine general-purpose CPUs with specialised devices that achieve outstanding performance and energy efficiency in specific workloads [68, 33]. At present, CPU-GPU systems are commonplace in the latest TOP500 and Green500 supercomputers list [3], and many computing workloads can benefit from GPUs' highly parallel architectures to accelerate the application, including simulations and machine learning.

Modifying existing applications to make the best use of available computational resources in heterogeneous systems presents several challenges resulting in a slow adoption. With the ongoing trend for increasing availability of heterogeneous systems — combining CPUs, GPUs, and specialised accelerators — exploring solutions that ease their adoption is crucial to accelerate scientific research and for many workloads. Over the last years, the industry and academia contributed with assistant tools to guide developers, new programming models to increase portability and productivity, and approaches to optimise applications automatically. Nonetheless, there is still room for improvement, and many issues remain unsolved [61].

1.2 Challenges in heterogeneous computing

Writing applications for CPU-GPU systems is challenging for developers and researchers for varied reasons [61]:

Architecture details. To extract maximum performance, developers must understand the execution flow at the architectural level to leverage available parallelism and optimise memory usage — which is the main factor for slow execution times and power consumption [33]. For instance, on CPUs, the cache is crucial to reduce memory accesses time. Therefore, algorithms should be cache-oblivious, improving spatial and temporal data locality. GPUs execute applications differently, constantly pre-empting groups of threads to overlap memory accesses with computation. The programmer must first break the algorithm into many independent work units to take advantage of the thousands of streaming processors making calculations in parallel.

Workload partitioning. Due to the characteristics of accelerators and workload properties, some tasks execute faster on the CPU and others on the GPU. Selecting the optimal scheduling and work partitioning needs to consider several factors. For instance, the host CPU and the GPU do not share memory. Therefore, offloading computation to the GPU has an overhead for transferring data back and forth. More complex partitioning strategies try to keep both CPU and GPU busy by distributing the workload on both units or finding independent tasks in the application. However, doing this process manually requires expertise and is time consuming. Moreover, the optimal strategy is not portable among different platforms and hardware. Chikin et al. [30] show that offloading a convolution kernel to the GPU in a *POWER8 CPU + NVIDIA Tesla K80* system results in a slowdown of 2.1×. However, in a *POWER9 + NVIDIA Tesla V100* results in a 4.4× speedup. This example illustrates that workload scheduling and partitioning depends strictly on the hardware. After one new product iteration for the CPU and GPU, the optimal strategy has changed. Therefore, previously optimised applications have to be re-written or adjusted whenever the platform changes.

Programming models. Programming models can be categorised as low and high-level. NVIDIA introduced CUDA [11] for general-purpose computing on NVIDIA GPUs and supports C, C++ and Fortran. AMD presented the HIP [15] programming model which is similar to CUDA model, but it can target AMD and NVIDIA GPUs. Support for NVIDIA is achieved by transpiling the HIP code to CUDA. OpenCL [9] emerged as an agnostic programming model for diverse hardware, including GPUs and FPGAs. It is an API for C/C++, and the library implementation is left to vendors. Unlike CUDA and HCC, OpenCL enables code portability — one source code can target various accelerators. The three models are considered low-level because they expose many architectural details and have steep learning curves. High-level programming models abstract most of the micro-architecture details. Moreover, they generally require fewer changes in the existing code base and are portable. OpenMP [10] and OpenACC [8] are directive-driven programming models and examples of high-level programming models. The problem is programming models have trade-offs between performance, ease of use and portability [38].

In conclusion, developing or modifying applications for heterogeneous systems poses many challenges. It requires vast experience, knowing many programming models and vendor-specific

tools. Moreover, optimising an application for top performance can be a long and time-consuming process. Developers may not have the necessary expertise and time to alter their algorithms to take advantage of heterogeneous systems. Making heterogeneous systems more accessible and easy to use is essential in many areas of application. Contributions that assist or automate applications parallelisation and offloading are of enormous interest. Despite the existing contributions, several problems remain unsolved or have various limitations [61].

1.3 Objectives

Our work proposes a framework that automatically parallelises loop nests for CPU and GPU using source-to-source techniques without any user intervention. We propose static analysis at AST level to find parallelisable code regions. Analytical models are used for guiding target selection for each parallel region. Given the relative performance modelling problem, we hypothesise using AST analysis to collect the metrics the analytical models need (e.g. the number of floating-point operations). The parallelisation on CPU and offloading to GPU is achieved using the OpenMP programming model and inserting the necessary pragmas.

We expect our approach to select at compile time between CPU parallel and GPU to reduce the hotspots execution times compared to the baseline, i.e. sequential execution in the CPU. Besides, we aim for an agnostic compiler solution that generates modified source-code with OpenMP pragmas, allowing the user to modify the code further and use the most suitable compiler available for the target platform.

1.4 Methodology

Our framework addresses two main problems. One is determining which loop nests can be parallelised, for instance, ensuring the loop is free of loop carried dependencies. The second part is performance modelling, using analytical models to estimate execution time for three targets: *(a)* CPU sequential, *(b)* CPU parallel, and *(c)* GPU. The target with the lowest estimated execution time is considered the optimal one.

Our approach uses static analysis and source-to-source techniques. We use Clava [25], a source-to-source framework to abstract compiler infrastructures, such as LLVM. Using a domain-specific language, LARA, one can write scripts that query and modify source code at the AST level. Clava outputs C/C++ files from the AST, reflecting any modifications performed with LARA scripts [28, 27]. Existing literature work that performs analysis at the IR level is restrained with a specific compiler infrastructure, and their output is, usually, a binary file. Such approaches work as black-boxes with few customisation opportunities. Our approach using Clava means the output of our framework is source-code. It enables the user to modify further the code for tuning the application or porting it to another system. Moreover, the user is free to compile the code with any compiler, which is particularly advantageous when using some compute platforms, such as IBM

or Cray systems, that offer customised compilers for their hardware platforms and leverage better performance [55].

Our approach builds on top of AutoPar [21], a library available in Clava, to determine which loops are safe for parallelisation. The code is parallelised using the directive-based OpenMP programming model. Depending on the target, different pragmas and clauses are inserted– CPU parallel or GPU. Compilers compliant with OpenMP and that support offloading directives, such as GCC and Clang, can compile the resulting source code. The binary launches on the CPU and the selected loops can run in parallel in the CPU or be offloaded to GPU, as decided statically by our framework.

We use state of the art analytical models to guide target selection at compile time. In our context, execution time estimation does not have to be accurate. Instead, we need reasonable relative estimates. We explore the feasibility to characterise parallel loop nests at the AST level rather than complex analysis in lower-level representations or even machine instructions. For comparison, we also use LLVM MCA [2], a tool that analyses assembly instructions and produces estimates for the number of clock cycles spent in the CPU.

The main contributions are:

- A framework that automatically selects between CPU and GPU targets for critical loops and involves automatic parallelisation via OpenMP and source-to-source techniques.
- An approach based on static analysis techniques and analytical models to select the best target for each parallel region *(a)* CPU sequential, *(b)* CPU parallel, and *(c)* GPU.
- Demonstrate the feasibility of characterising hotspots at AST-level in a relative performance modelling context.
- Integration of LLVM Machine Code Analyser [2] in a source-to-source approach to estimate computational cost of instructions in CPUs. Extensions to the Chikin et al. [30] method to improve estimations accuracy.
- Adaptation of the Kim et al.'s [47] GPU analytical model for the OpenMP offloading context and to address some limitations concerning modern GPU architectures.
- Experimental results using 14 benchmarks from the UniBench [56] repository show the capability of the approach to map and improve performance for 11 of the 14 kernels.

1.5 Structure of the Dissertation

The remainder of this dissertation is organised as follows: Chapter 2 presents LLVM MCA and an overview of the GPU architecture with an emphasis on NVIDIA GPUs. Chapter 3 describes the state of the art in heterogeneous computing techniques for CPU-GPU systems. Chapter 4 describes the implemented analytical performance model for the CPU, based on existing work. Besides, it describes the two approaches considered for characterising the application: AST analysis and

using LLVM MCA. Chapter 5 presents the used GPU analytical model, from state of the art, and Chapter 6 identifies and addresses some of its limitations. Chapter 7 discusses the framework overall, including implementation details for the analysis at AST level, parallelisation strategy and how the analytical models are integrated for guiding target selection. Chapter 8 presents the experimental results. Finally, Chapter 9 concludes the dissertation and proposes future work.

Chapter 2

Background

2.1 LLVM Machine Code Analyser

The `llvm-mca` [2] is a performance analysis tool that helps predict the performance of a region of code and diagnose bottleneck issues. It analyses a sequence of assembly code and produces reports that include: Instructions Per Cycle (IPC), CPU cycles, pressure on the various hardware resources, and timeline views of the instructions state transitions at every simulated cycle (dispatched, executed, retired, ...). According to official documentation [1], the reported information is inspired in Intel's Architecture Code Analyzer (IACA), which has reached the end of life (EOL) back in 2019.

`llvm-mca` uses the scheduling models available in the LLVM infrastructure, which are specific for each target device/architecture. The models specify: instruction latencies and define available hardware resources, the number of resulting micro-ops after decoding an assembly instruction, ports needed by instruction's micro-ops, and the dispatch width (rate for issuing micro-ops from the frontend section to execution engine section), among other architectural parameters.

`llvm-mca` is a static analysis tool, i.e., it does not execute instructions in the CPU. Instead, it uses the information available in the scheduling models to simulate the different steps in a real CPU, further discussed below. The region of assembly code is simulated in a loop for a number of user-configurable passes. It is advantageous when inspecting a loop's body and exploring the various degrees of pipelining of modern superscalar CPUs.

2.1.1 What MCA models exactly?

This section analyses `llvm-mca` in the context of superscalar architectures to illustrate what exactly is modelled and with what accuracy. This analysis is based on official documentation and by inspecting some scheduling models in the LLVM's source-code repository[2].

[1] https://www.llvm.org/docs/CommandGuide/llvm-mca.html
[2] https://github.com/llvm/llvm-project/tree/main/llvm/lib/Target/X86

The CPU is organized in various sections. To execute an instruction it is necessary to go through multiple stages. The main sections are commonly described as [46, 70, 34]: *(a)* Front-End, *(b)* Execution Engine or Back-End, and *(c)* Memory Subsystem.

2.1.2 Front-end

In the front-end, instructions are fetched from an instruction cache (I-Cache), decoded and split in a sequence of micro-operations (μops) which are natively executed in the execution units. The micro-ops flow into the back-end section at a given rate that depends on various factors but is bounded by the interconnects — typically, just a handful of ops per clock cycle.

`llvm-mca` does not model the front-end of a CPU, and therefore any bottleneck there is not reflected in the analysis. Instead, `llvm-mca` assumes all instructions have been decoded and fetched into the first step of the back-end. The actual first unit in the back-end varies from architecture. In general, the first unit is a *ReOrder Buffer (ROB)* or a *Retire Control Unit (RCU)* for out-of-order CPUs. The mentioned buffers queue micro-ops to be retired in programs order after execution completes. This approach allows micro-ops to be scheduled and issued in any order as micro-ops become ready for execution.

2.1.3 Back-end

Assuming the micro-ops are already queued in the front-end, ready to be dispatched to the back-end, `llvm-mca` models the following stages from the back-end phase:

Dispatch. Instructions, already decoded in micro-ops, are retrieved in groups and in program order into ROB or RCU. The group's size is limited by the issue width, an architecture parameter. `llvm-mca` fetches instructions if the following requirements are met: *(a)* there are available entries in the ROB/RCU buffers, *(b)* there are sufficient physical registers for allocation and renaming, and *(c)* the scheduler(s) buffer entries are not full. The dispatch pipeline phase in `llvm-mca` not only pushes instructions into ROB/RCU but also dispatches the instructions to the respective schedulers. Recent Intel x86 micro-architectures have a unified scheduler unit [17, 34], while AMD's Zen micro-architectures [19, 60, 70] have separate schedulers, for integer and memory instructions, and for floating-point and vectorised instructions. Each scheduler has its queue of instructions and physical register files. `llvm-mca` pipeline model supports and manages multiple schedulers and register files.

Issue. Instructions in the scheduler queues may not be ready for execution if the source operands are not ready. At every simulated cycle, `llvm-mca` checks which scheduled instructions are ready for execution. Instructions in the ready state are issued into proper execution units. In contrast to ROB/RCU, which has to remember the original program's instructions order, the scheduler can issue in any order, assuming the CPU is Out of Order (OoO). Older instructions are prioritized over recent instructions, and a round-robin

scheduling policy is used to even execution units pressure. Note that in Intel/AMD x86 architectures, some micro-ops can be issued to multiple ports. `llvm-mca` tries to even execution units's usage. The ports to which a given instruction can be allocated is part of the scheduling models.

Write-Back. Operations that complete execution remain in ROB/RCU, as they must be committed in the program's order. When one operation's source operand depends on an executed instruction result, the value can be broadcasted from ROB/RCU enabling speculative execution.

Retire Stage. Also known as the commit stage, is when the result is written to the physical register, and the respective entry in ROB/RCU can be released.

In the back-end section, `llvm-mca` models the main parts of current pipeline superscalar processors. However, it does not model branch predictions.

Another limitation is the lack of modelling for HyperThreading (HT) [54] or Simultaneous Multithreading (SMT). HT and SMT are Intel and AMD processors' feature that enables sharing CPU Core resources between two different threads. It is particularly advantageous when two threads require different CPU resources, increasing resource usage and overall system throughput. Note that with HT/SMT some Core resources are partitioned statically, while others are competitively shared (e.g., schedulers and execution units) [17, 60]. Consequently, the accuracy of the `llvm-mca` estimations is affected. Although the documentation [2] does not explicitly state the lack of support for HT/SMT, evidence can be found in source-code comments [3].

One additional limitation, not stated in `llvm-mca` documentation [2], is lack of modelling for optimisations that happen on the fly in the CPU. For example, the *Move Elimination* or *Zero-Latency Move Instructions*, where some register to register move operations are solved in the front end, not consuming any resources in the back-end (e.g., queues, scheduling units) [17, Section 3.5.1].

```
.intel_syntax
main:
   mov eax, ecx
   add eax, ebx
```

Listing 2.1: An assembly excerpt to demonstrante *Move Elimination* optimisations

To demonstrate the lack of pipeline-level optimisations in `llvm-mca`, consider the assembly example in Listing 2.1. The first instruction moves `ecx` to `eax`, and then makes an addition: `eax = eax + ebx`. In this example, it would be possible to skip the initial `mov` operation and map the register `eax` to `ecx` at physical level. Analysing the snippet with `llvm-mca` shows that the move elimination optimisation is not considered, as all instructions consume resources in the back-end and there is a data dependency between the two — the `add` only starts executing after `mov` completion. Therefore, it is very likely that other optimisations are not modelled either.

[3] https://github.com/llvm/llvm-project/blob/main/llvm/lib/Target/X86/X86ScheduleZnver2.td#L119

2.1.4 Memory subsystem

The last section in the execution pipeline of a CPU is the memory subsystem. Load and store operations are issued to dedicated execution units and buffered in load and store buffer units. The memory buffers have an interface for the first level data cache (L1D) for reading and storing data.

Modern CPUs offer mechanisms for forwarding stores to loads. Consider a store operation at address A. It will be buffered in the store unit while the actual data is flushed to the cache, committing the memory operation. Also, suppose a load operation for the same address A that follows the store in the original program order. Even if the data is not committed to the cache, the data from the store can be forwarded to the load, serving it immediately. The technique enables the CPU to run speculatively and avoid the latency of storing and then reading from the L1D cache.

Regarding `llvm-mca` there are some limitations in the memory subsystem. Although it does not bound the store and load buffers, allowing infinite memory operations in-flight, it has Command Line Interface (CLI) parameters to specify custom number of entries (`-lqueue` and `-squeue`).

The memory model does not support different memory operation types (e.g., write-back, write-combining) but conservatively ensures consistency. For example, load operations can be re-ordered as long as there are no interleaving stores. Stores are performed in order and cannot pass previous loads.

Other limitations are the lack of store-to-load forwarding and no modelling of the cache hierarchy. Therefore, `llvm-mca` does not simulate cache misses or hits in L1D. Instead, it uses a load-to-use latency for load operations that should approximate a cache hit in L1D.

2.2 NVIDIA GPU Overview

2.2.1 CUDA programming model

From developers point of view, the GPU is a scalable array of processing units that execute hundreds or thousands of threads in parallel. As the CUDA programming model has different abstraction layers to enable transparent scalability, the same application can execute on different architectures with varying computing resources.

In CUDA, the programmer splits the problem into independent sub-problems that can execute concurrently and in any order. These independent sub-problems are represented as *thread blocks*. The blocks are scheduled and organised in one or more groups of threads that cooperatively and simultaneously execute the sequence of instructions [12, Chapter 2].

Despite the top-down description, in practice, the code is written from the perspective of a single thread following a Single Instruction, Multiple Threads (SIMT) model. In CUDA C/C++, the thread operations are defined in a function known as the *kernel*. Listing 2.2 illustrates the SIMT model adding two vectors: $C = A + B$. The most basic approach for breaking the problem into independent tasks is assigning each data element to a single thread, i.e., each thread adds one element of `A` with `B` and stores the result in `C`. All threads perform the same action, but on different

elements of the vectors. The presented approach assumes the number of launched threads matches the vector sizes, and each thread has an unique identifier, `tid`. In Listing 2.2, the identifiers range in $[0, N[$, where N is the problem size and number of threads launched.

```
__global__ void vecAdd(float *A, float *B, float *C) {
   int tid = blockDim.x * blockIdx.x + threadIdx.x;
   C[tid] = A[tid] + B[tid];
}
```

Listing 2.2: Naive vector addition in CUDA programming model

The common approach to ensure threads operate on distinct data elements is to use a global unique identifier. As explained before, a kernel is launched in a GPU organised in different thread blocks, and each thread block has one or more groups of threads. Each thread block has an ID, assigned sequentially, and each thread has a relative ID to the thread block. The pair $(BlockId, ThreadId)$ defines an unique identifier for each thread. At code level, an unique integer identifier can be calculated as $BlockId * BlockDim + ThreadId$, where $BlockDim$ is the thread block size. Listing 2.2 shows how to calculated the identifier in CUDA, storing the result in the `tid` variable.

In order to launch a kernel in the GPU, the programmer specifies a launch configuration — also known as grid geometry —, setting the number of thread blocks and respective size. The blocks and threads within the block can be organised in a 1D, 2D or 3D grid. Accordingly, the register identifiers (e.g., `blockIdx.x`) have the x, y and z axis, making CUDA more flexible when working with multi-dimensional algorithms.

Listing 2.3 shows one possible kernel launch configuration for the vector addition in Listing 2.2, assuming the problem size is $16 \times 256 = 4096$.

```
int main() {
   ...
   vecAdd<<<16, 256>>>(A, B, C);
   ...
}
```

Listing 2.3: Launching vector addition kernel with 16 thread blocks and 256 threads per block

2.2.2 Streaming Multiprocessor

The previous section presented the GPU at a high abstraction level from a programmer's point of view. This section provides more details on how blocks and threads are scheduled and executed at the hardware level in NVIDIA architectures.

The SM is one of the main building blocks at the hardware level in NVIDIA GPUs. The SM aggregates all parts for executing applications, including execution units, warp schedulers, dispatch units, registers, and other on-chip memories (e.g., caches). It also includes units for graphical applications, but that is out of context for this work. A GPU device combines one or more SMs, off-chip memories shared among the SMs, and the host interface to receive commands and data from the CPU.

When the GPU receives instructions from the host CPU to execute a kernel, the thread blocks are distributed among the SMs. In turn, each SM arranges the thread blocks in groups of threads of fixed size, called *warps* — 32 threads in all architectures released so far. Accordingly, each SM manages a pool of warps and is responsible for creating the warps, scheduling and executing them.

Each SM has one or more *warp schedulers*, a unit responsible for selecting the best warp from the pool for execution. Some warps might not be ready to execute due to data dependencies, synchronization barriers and other reasons. Therefore, the scheduler must select warps in a ready state. The scheduling algorithm is not documented, but it is presumably based on readiness and fairness, i.e., all warps are given even time slices.

When one warp is elected for execution, the dispatch unit commits one instruction at a time. If all threads are on the same execution path, they execute the instruction in parallel and reach maximum efficiency. When the threads diverge, the SM needs to execute instructions for all the active paths. For instance, consider an `if else` statement, where half of the warp executes the statements in the `if` block and the other warp executes the `else` block. When the SM issues instructions for the `if` block, threads on the `else` block are inactive and do not execute any instruction, reducing the overall throughput and efficiency.

One significant difference between GPUs and CPUs is that GPUs devote most of the die area to computing units, while CPUs have large I/O and control blocks, shrinking the space available for the cores. Even inside each core, considerable space is dedicated to complex units such as branch predictors, decoders, instruction re-ordering, among others. On the other hand, GPUs try to remove as much control logic on the hardware as possible. As a result, instructions are executed in order and the scheduling and when to pre-empt warps is delegated to compilers [48].

Another interesting design choice in GPU architectures is that warp execution context resides on-chip during the lifetime of a warp, enabling free warp switching in the warp schedulers. On the other hand, CPUs need multiple steps to save and restore the context of each thread. That design approach means that when a warp stalls (e.g., memory operation), the SM can swap to another ready-to-execute warp with small or no overhead keeping the computing units as busy as possible. To achieve high occupancy levels it is vital to select a suitable grid geometry.

Reducing control logic and warp scheduling latencies implies that hardware resources, such as registers, are assigned when the warp is created and the allocation is fixed. Until the warp completes the execution, the allocated resources cannot be used elsewhere. As resources are finite, there is a limit on the number of simultaneous warps on the SM, thereby limiting the size of thread blocks the programmer can use. Note that thread blocks are assigned to SMs as a whole and it is the SM responsibility to create the warps — the programmer has no control over it. The number of distinct thread blocks assignable to an SM is also constrained due to the finite resources available. The GPU has a pool of thread blocks that are assigned to SMs as they complete executing a set of thread blocks.

2.2.3 Execution units in Streaming Multiprocessors

This section outlines the most relevant execution units available in the SM [52]:

CUDA Core, Core, Streaming Processor. The terminology varies from author to author, but as the NVIDIA documentation primarily uses the *Core* and *CUDA Core* terminology, that nomenclature is used throughout this work. Terminology aside, a CUDA Core has an ALU unit and a Floating Point (FP) unit. It supports 32-bit and 64-bit integer operations. On the other hand, the FP only supports 32-bit operations and includes fused multiply-add instructions. It is important to note that both ALU and FP sub-units do not implement all instructions natively. For instance, integer multiplication and divisions are not native operations in many NVIDIA architectures, depending on the target market segment. This topic is further discussed later on.

Load/Store Units. Memory instructions are dispatched to Load (LD) and Store (ST) units. These are responsible for calculating the requested addresses and issuing the data requests to the memory units freeing the SM to continue executing instructions from the same or other warps.

Special Function Units (SFU). Execute transcendental instructions, such as trigonometric functions, square root, logarithmic and others.

Double Precision (DP) Units. Units for handling 64-bit double-precision operations. These units are often not mentioned or depicted in the SM diagrams in official documentation, especially on GPUs for the consumer segment, as the number of units just too low. As an example, the typical ratio of CUDA Core to DP units is 32/1 per SM. For professional and data-centre segments, the scenario is entirely different. The Tesla P100, based on Pascal architecture, has a 2/1 ratio per SM [6].

According to the Fermi architecture report [4], the CUDA Cores are fully pipelined, and the dispatch units issue instructions every cycle.

2.2.4 Memory Hierarchy

GPU memory hierarchies are more complex than in CPUs, in the sense that the number of memory components is higher and some memory spaces are disjoint. The memory layers vary in capacity, latency and are often optimised for specific purposes.

In the CUDA programming model, memory can be viewed in two ways: the logical memory space and the memory chips at the hardware level. A programmer usually thinks of memory in the logical representation, which is exposed in the programming model. Logical memory includes the global, local, shared, texture/surface and constant memory spaces. The hardware is composed of device memory (DRAM), caches and other dedicated on-chip memory units — e.g., shared memory or registers. Often, multiple logical spaces map to one memory device or share datapaths.

The most relevant logical memories available in CUDA are enumerated below:

Global Memory. In general, a global memory access is thought as accessing the device or DRAM memory, the memory device with largest capacity that is shared between all SMs. It is also the slowest memory to access. Device memory is connected to a two-level cache

system, and all read/write memory goes through the cache hierarchy. Generally speaking, reads and writes are always cached in L2, but some architectures have opt-in L1 caching and the size is user-configurable.

Shared Memory. This memory space has a dedicated on-chip per SM. It is allocated per thread-block basis and is often used for data-sharing (communication) and synchronisation between threads that belong to the same thread-block. It is low latency memory (similar to register accesses) with high throughput. Shared memory usage is fully controlled by the programmer. On some older architectures, L1 and shared memory were unified in a single memory device.

Registers. Each partition in the SM a register file shared by active threads in the warp scheduler. Each architecture has a limit on the number of registers that can be allocated to a thread and respective thread block.

2.2.5 Driver and Runtime APIs

The CUDA C/C++ programming language is an extension of the C/C++ language. The syntax extensions are minimalist and mainly used for kernel launching, marking functions as kernels (i.e. code that executes on the GPU device) and specify memory location (e.g., constant, shared memory, and others).

Additional features are available through APIs. There are two types of interfaces: *CUDA Driver API* [13] and *CUDA Runtime API* [14]. The former is a low-level API that provides more fine-grained control over contexts/processes and module management. The Runtime API is built on top of the Driver API and is the interface that is most commonly in CUDA. It is used to allocate memory buffers on the GPU, manage data transferring between host CPU and GPU, among others [12, Chapter 3].

2.2.6 Compute Capability

In the CUDA programming model, *Compute Capability* (CC) is a concept that represents a set of features supported in a given GPU. The CC also defines some architectural properties, e.g., the number of registers available and on-chip memory sizes.

The compute capability is expressed with major and minor revision numbers, $M.m$. Generally, the major number is incremented with the introduction of new architectures. For instance, all devices with CC 6 are from the Pascal architecture. The minor revision is related to refinements on the architecture or different configurations for targeting distinct market segments (e.g., desktop and mobile personal computing, data centre, HPC) [12].

2.3 Summary

This Chapter presented the LLVM MCA [2] tool, explaining the capabilities of LLVM MCA, as well as its limitations regarding superscalar pipeline modelling. This Chapter also presented an overview on NVIDIA GPU architectures, the CUDA programming model and terminology used throughout this dissertation.

Chapter 3

Related work

This chapter presents a literature review on automatic code parallelisation and offloading for CPU-GPU systems and its associated challenges. Section 3.1 describes automatic and naive parallelisation of sequential C code, generating code that targets GPUs using source-to-source techniques. Section 3.2 introduces skeleton programming, which are libraries with high-order functions for common parallel patterns, abstracting the details of GPU programming. Other work addresses automatic workload partitioning on available processing units for best performance and is described in Section 3.3. Section 3.4 presents approaches for creating a fully automatic flow that detects potential parallel code regions, generates code for GPU, and partition the workloads for best performance. Essentially, it is a combination of methodologies from Sections 3.1 and 3.3 in a single framework. The following Section 3.5 addresses the data management problem that exists when offloading computation to the GPU. Section 3.6 presents an overview on revised literature. Some related work is described in greater detail as it is closely related to our work.

3.1 Automatic parallelisation for GPU

This section presents approaches that automatically analyse sequential C code, find parallel regions, and generates GPU code.

In this dissertation, we refer to such approaches as *naive* because they do not evaluate if there are any benefits in offloading to the GPU and can result in slowdowns. Most approaches we found use source-to-source techniques for code generation, with the following advantages.

- The developer can inspect the resulting code and fine-tune it.
- It is possible to use any compiler compliant with the used programming model, as different compilers support diverse architectures or achieve distinct optimisation levels.
- The developer can revert some of the changes due to performance penalties or algorithm correctness.

The proposed tools for automatic parallelisation targeting GPUs use one of the following programming models: *(a)* CUDA [11], *(b)* OpenCL [9], and *(c)* OpenACC [8].

3.1.1 C-to-CUDA

C-to-CUDA [24] generates parallel CUDA from sequential C code. It parallelizes affine *doall* and *doacross* loops using polyhedral models. In *doall* loops, loop iterations are independent and can execute simultaneously. In the second class of loops, `doacross`, inter-iteration dependencies exist. C-to-CUDA enables the fusion of successive loop nests by working on the largest region of static control code. Further, it explicitly manages the memory hierarchy, generating efficient access patterns on global memory to enable coalescing and the use of on-chip memories, like constant memory, for repeatedly used data.

3.1.2 PPCG

PPCG [71] is similar to *C-to-CUDA* [24]. It is a source-to-source compiler that uses polyhedral techniques to generate CUDA. Despite the similarities, the authors highlight a few key distinctions. For example, concerning data allocation in the GPU memory hierarchy, C-to-CUDA moves all arrays to shared memory without considering capacity constraints and inter-thread data usage that could benefit from global memory coalescing. PPCG is cautious in this regard. It tries to move data to registers when there's is reuse; if access is non-coalesced data is moved to shared memory; otherwise, it is left on global memory. They also present a code generation scheme for imperfect loop nests where statements are not in the innermost loop (perfect loop nests).

PPCG and *C-to-CUDA* reach performance levels close to hand-written and optimised code. However, these tools generate kernels whenever possible without considering profitability in offloading. Furthermore, by using polyhedral models the approach is limited to affine loops.

3.1.3 DawnCC

DawnCC [58] is a source-to-source compiler that automatically annotates C code with OpenMP/OpenACC directives to exploit data parallelism and offload computations to GPU. According to the authors, DawnCC was the first tool to insert OpenACC/OpenMP directives without user intervention and ensures correctness. Moreover, the authors address two particular challenges: pointer disambiguation and optimisation of data transfer operations.

The compiler supports *doall* loops, i.e., loops where every iteration can execute simultaneously. It uses well-known compiler techniques to find the loops. The tool inserts parallelism/offloading directives and explicit data transfer clauses to specify what needs to be copied from CPU to GPU and vice-versa. DawnCC addresses the pointer aliasing problem as it can compromise the parallelisation correctness. Loop nests are pointer aliasing free if all memory locations are referenced by at most one pointer. It uses symbolic range techniques to compute the lower and upper bound indices for every pointer symbol. Given the base address and the bounds, it computes the region of memory referenced by a given pointer, which is sufficient to evaluate if two pointers

overlap. DawnCC inserts the pragmas with a conditional directive that at runtime asserts if the loop is free of pointer aliasing. If it is not, the sequential code executes. Besides, OpenACC data directives need explicit memory boundaries. The symbolic range analysis solves that issue as well.

The evaluation results show the relative performance between code automatically annotated by DawnCC and the original sequential version. For embarrassingly parallel programs, the DawnCC version results in considerable speedups. However, shorter benchmarks exhibit slowdowns due to the lack of cost analysis to weight offloading decisions.

3.2 Skeleton programming

Skeleton programming consists of a library with templates for general data and task-parallel compute patterns. The library defines interfaces for the patterns in order to abstract the details of parallel programming. Another advantage in skeletons is portability. The purpose is that the library has an optimized and tunned implementation that encapsulates memory hierarchies, synchronization, programming models, and other features very close to the architecture.

The patterns are generic and related to data element accesses and computation. Therefore, the developer has to implement a user-function that defines the actual operation on a set of elements and is a parameter for the skeletons — high-order functions.

3.2.1 SkePU

SkePU [35] is a C++ library that implements parallelised skeletons for common data access patterns. Skeletons are implemented and optimised in various programming models, targeting CPUs and GPUs. A skeleton is a high-order function and requires a user-defined function to specify the operation. To make use of skeletons, the user needs to modify his program where needed.

The skeletons available in SkePU support patterns such as *Map*, *Reduction*, *MapReduce*, *MapOverlap*, and *MapArray*. For instance, the mapping skeletons iterate over a data sequence and transforms the original items into new ones according to some function, $f(x)$, defined by the user. The skeleton's operators are defined by the user via available macros in SkePU. Skeletons are initialised with the declared operator and then invoked on data input — an array or a C++ iterator.

SkePU skeletons are objects with member functions for the different implementations — sequential CPU, parallel CPU with OpenMP, CUDA, and OpenCL. The user can invoke a specific implementation or leave the decision to SkePU, which selects the implementation and target based on availability — however, the exact approach or heuristic is not described in [35]. SkePU supports multi-GPU with OpenCL and CUDA implementations, and by default, uses all GPU units available and evenly distributes the workload. However, the user can limit the number of GPUs that SkePU uses.

Furthermore, SkePU proposes a container based on STL's[1] `vector` that tracks data located on GPU to manage data transferring. Data is sent back to the CPU only when the host CPU

[1] https://www.cppreference.com/Cpp_STL_ReferenceManual.pdf

accesses it, using a lazy approach. This approach prevents redundant data movements, particularly in subsequent skeletons that run on GPU as input data for skeletons resides on GPU already. Despite its benefits, the container has a single dimension. As, it is the only supported data structure in SkePU, it limits the flexibility in using the library.

SkePU 2 [37] is an improvement on the original SkePU. It simplifies the programming interface adopting C++11 features and type-safety. The skeletons are more flexible than the predecessor version and support multi-dimension containers — MapArray becomes redundant. Moreover, it adds support for the *Scan* pattern (also known as the prefix or cumulative sum). The reduction skeleton for matrices can be applied in one or two dimensions with configurable direction (row-first or column-first). Row-major order is also supported.

3.2.2 Muesli

Muesli [36] is a C++ library that offers several data-parallel and task-parallel skeletons that support multi and many-core architectures, and clusters. It uses OpenMP for multi-core CPUs, CUDA for GPUs, and *Message Passing Interface (MPI)*[2] to handle communication in clusters.

Data-parallel skeletons include a *map*, *zip*, *fold*, and *scan* patterns. The skeleton behavior is defined with user functions or functors implemented using C++11 templates. Besides, it supports task-parallelism and offers skeletons for *farm*, *pipe*, *divide and conquer*, and *branch and bound* topologies.

Muesli has distributed data structures to be used with the skeletons. There are variants for an array, matrices, and sparse matrices. The data structures abstract the data partitioning and communication between host and GPU in the same node and between separate nodes with MPI. Moreover, the data structures enable the developer to have a global view of the data structure and not worry about local partitions. However, *Muesli* also gives the possibility to use local indices (relative to a partition on some processing unit). The data structures track data modifications in host CPU and GPUs to check data coherence and prevent redundant communications.

C++ templates enable support for standard and user-defined data types, which makes the library very flexible. One problem, however, is the need for serialisation in inter-node communication. By default, *Muesli* offers implicit serialisation for primitive data types or pointerless objects. Such data types are stored in memory as contiguous blocks. However, for more complex structures that have pointers, it is necessary to pack all the data in a contiguous memory block before transferring it between nodes. Such object classes must extend a *Muesli* class and implement the object serialisation method, which is invoked on transmission and reception, for packing and unpacking the object.

Muesli is a portable library and can be compiled to target different systems: a combination of shared-memory multi-core CPUs with GPUs or multi-GPUs and distributed memory systems with clusters of the previous combinations. However, it is not clear if the user can control workload

[2] https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

partitioning considering the problem size to prevent slowdowns or if *Muesli* has built-in techniques to manage it. The article seems to suggest *Muesli* uses everything available in the system.

3.2.3 Bones

Bones [64] is a source-to-source approach based on algorithmic skeletons techniques to ease C code parallelising for CPU and GPU. Skeleton libraries require the user to modify the code manually, reconstructing the algorithm on top of the available skeletons. Besides, some libraries require specific data structures and containers. *Bones* proposes algorithm classifications to address some of the difficulties in skeleton programming. The classifications are introduced manually by the user. However, these are typed with compiler pragmas, reducing the user effort when comparing to other skeleton libraries such as *SkePU* [35, 37], or *Muesli* [36]. Moreover, *Bones* aims to generate target code that is readable and editable for further manual tuning.

The algorithm classification grammar lets the user define input and output data-structures and the data access patterns: elements are accessed individually, data is accessed in a tile pattern, and so on. For instance, `512x512|element → 512x512|element` indicates an algorithm that reads elements individually from a 512×512 data structure, and writes to a 512×512 structure. According to this grammar rule, the operations are independent and can be parallelised. In the source code, the classification is introduced with enclosing pragmas: `#pragma kernel 512x512|element → 512x512|element ... #pragma endkernel`.

Bones parses the input sequential C code with algorithm classification pragmas and generates the AST. The AST is used to extract information and instantiate a skeleton according to the user's algorithm classification. From the AST, it generates the final code. Skeletons support different targets: CPU, AMD GPU, and Nvidia GPU. For the first and second targets, skeletons produce OpenCL, and for the latter, CUDA.

3.3 Automatic workload partitioning

One of the concerns in heterogeneous computing is distributing the tasks in a program among the available processing units. Various aspects need attention to extract maximum performance. For instance, tasks can be mapped based on their characteristics. An algorithm that is very irregular and has complex branching is more suited to CPU. On the other hand, highly parallel problems benefit from GPU architectures with many thousand of cores available.

Offloading tasks to an accelerator is not for free. For instance, there is overhead due to data transferring as the host and accelerators may not share memory. Literature uses static, dynamic, or hybrid approaches to evaluate offloading benefits and account for overheads. Static analyses are inherently incomplete as not all information is available at compile time (e.g., the iteration number in a loop). Therefore, an alternative is dynamic analyses, i.e., at runtime. However, this can introduce considerable overhead. Hybrid approaches mix static and dynamic analyses. At compile time they extract relevant information and at runtime conditional expressions are evaluated to make a final decision.

Selecting the optimal target for an individual task is quite complex by itself. However, this does not ensure an optimal workload partitioning as some processing units are idle. Consequently, more complex analyses explore load balancing, pipelining and task parallelism. In load balancing, a data-parallel task is distributed among the processing units to keep all units busy. If all units complete their sub-tasks at similar times, the partitioning is optimal. Pipelining techniques try to hide communication and other overheads, keeping some units busy with computation, while the preparation steps happen in parallel ahead of offloading computation. Finally, task parallelism detects independent tasks that can execute simultaneously. Since they are independent, they can be distributed among available hardware units. The main issue is then selecting the optimal allocation scheme to reduce power consumption or execution times.

Literature on workload partitioning is vast. This section addresses existing work on individual code region analyses. Techniques such as load balancing, pipelining or task parallelism are out of scope for our work objectives.

3.3.1 Hybrid CPU-GPU execution in SkePU

Kessler et al. [77] extended SkePU 2 [37] with a hybrid workload partitioning backend. For simplification, SkePU 2 is referred as SkePU in this Section. The proposed approach inspects SkePU skeletons individually and splits the problem among CPU cores and multi-accelerators — known as *workload balancing*. The partition ratios can be set by the user or automatically estimated with performance benchmarking.

When the Skeleton is invoked, the workload is partitioned in two parts accordingly to a ratio parameter: one for the CPU and another for accelerators. The ratio parameter is configured by the user or auto-tuned on a skeleton basis. The CPU partition is split into $N-1$ parts, where N is the number of cores in the CPU. The remaining core is responsible for managing the accelerators. The accelerator partition is further divided when multiple devices are available, a feature available in CUDA and OpenCL backends. However, in this approach, the workload is evenly distributed. It does not ensure optimal partitioning if the available accelerators are not equal devices.

With auto-tuning enabled for estimating the partition ratio, the framework uses performance benchmarking to find a suitable value. As estimating optimal workload partitioning for all cases requires complex and demanding algorithms, they propose an auto-tuner that yields good estimates on common scenarios and assume execution time has a linear relation with problem size. The tuner is performed once per skeleton for a given application. Therefore, different applications require individual tunning. The auto-tuner measures execution time on CPU and accelerators using the different backends available. The collected execution times are approximated with a linear regression $t = a \times N + b$, where N is the problem size, and a and b are constants. The approximation assumes the user function has constant complexity. The partition ratio R splits the workload into two partitions: $N \times R$ for CPU and $N \times (1-R)$ for accelerators. The partition ratio is optimal when the execution time is equal, avoiding idling in processing units. Therefore, solving both linear regressions for R outputs the optimal partition ratio. The value of R may indicate that

accelerator-only or CPU-only are the optimal schemas. In such cases, the SkePU uses just one of the backends accordingly.

3.3.2 Chikin et al.

Chikin et al. [30] propose a novel approach for using performance analytical models to guide the offloading decisions. The authors also present an initial study on target-offloading, introduced in OpenMP 4.0 specification.

Their approach uses one analytical model for CPU and another for GPU. Besides, it proposes a new symbolic analysis to evaluate the memory access patterns and classify them as coalesced or uncoalesced. The inputs are applications with hotspots outlined with OpenMP directives. The prototype framework augments the compiler to collect information in the static analysis phase to construct skeleton models. The OpenMP regions are duplicated, creating a compiled version for CPU (parallel) and another for GPU. At compile time, the skeleton models are incomplete as not all information is available. The OpenMP runtime is augmented to feed the missing data to the skeleton models and to estimate the speedup in offloading to the GPU, deciding which kernel version grants optimal performance. The overhead is minimal because calculating analytical model estimations consists in computing few linear equations.

The experimental results show that overall execution times are improved, which means the offloading decisions are optimal. It further demonstrates that predicted GPU speedups are relatively close to the real values. The authors suggest that the GPU model requires further tunning, for example, a detailed memory hierarchy modelling. Finally, some assumptions and heuristics also lead to more errors, such as assuming that conditional branches execute 50% of the time. Analytic performance models reduce runtime overheads and do not require previous runs of the target application for profiling, making it appealing for production systems. Precise models can estimate execution times or relative speedups with high accuracy, an important factor for optimal offloading decisions.

3.4 Automatic parallelisation and workload partitioning

This section presents work that automatically finds parallel code regions in sequential applications, evaluates performance gains in offloading and makes the necessary code transformations to execute the code region on an accelerator. It addresses the problems presented in Sections 3.1 and 3.3, all in the same framework.

3.4.1 HTrOP

Plessl et al. [67] proposes *HTrOP*, a tool that automatically analyses sequential code, detects computational intensive regions (hotspots) and generates parallel OpenCL kernels — that can target multiple co-processors such as GPGPUs and FPGAs. The decision for offloading happens at runtime, when all information is available. It either selects the best available accelerator or

executes the sequential version on CPU. The tool takes LLVM bitcode as input. Therefore, it supports any language in the LLVM infrastructure: e.g. `clang` for C/C++, `flang` for Fortran.

Hotspot analysis is performed at compile-time and supports a subset of loops that fit the polyhedral model with the following restrictions: the loop trip count expressions must be an affine function and not have cross-iteration dependencies. Hotspots are assigned a score computed as a weighted sum of floating-point and integer instructions. Hotspots that do not exceed a user-defined threshold score are not offloaded. Remainder hotspots are considered offloading candidates and are wrapped into functions that contain a parallel OpenCL code generated from LLVM bitcode. The offloading decision for these hotspots is delayed for runtime. In order to achieve it, function calls are augmented with a control unit that makes the decision when all the information needed is available: *(a)* input/output data sizes, *(b)* available accelerators, and *(c)* the possibility to reuse one accelerator on successive hotspots to reduce additional overheads. The control unit might decide to execute the sequential version if there is no offloading benefit. Otherwise, it selects the best accelerator available and compiles the OpenCL kernel, generated at compile-time, for the selected device.

The proposed approach supports different accelerators using the OpenCL programming model. Moreover, evaluation results show performance levels close to manually written OpenACC applications. However, hotspot detection is limited to affine loops that fit polyhedral models. Also, it does not regard each accelerator architecture which might benefit from code transformations and optimizations. Finally, the runtime kernel compilation induces significant overheads that need amortization, therefore reduces the set of problems that could profit from offloading.

3.4.2 Etino

Etino [65] is an automatic and fully static tool for mapping computations in sequential C code to the GPU. It results in a hybrid program that uses both CPU and GPU to speed up the application compared to the baseline sequential CPU code. Cost models guide the workload partitioning.

Etino parses the original program and constructs an *Extended Call Graph* (ECG), which in practice is a *Call Graph* with additional information. Except `main`, functions are duplicated in the graph, creating a node for CPU execution and another for GPU, i.e., f_{cpu} and f_{gpu}. The ECG captures caller-callee relations. Assuming a function f calls g in the original program, in ECG, both f_{cpu} and f_{gpu} link to g. That way, the callee knows the caller's execution context, and it is possible to account for communication and accelerator initialisation overheads. Cost models augment the ECG to estimate the computing cost for every function and the cost for a function call others (edges). An algorithm processes the ECG to find a subset graph that minimizes the overall cost. This approach's weakness is that separated invokes for a function g in the same caller f are not distinguished in the graph. As a result, all calls for g in function f are assigned to the same processor. After the scheduling algorithm, *Etino* calls DawnCC [58] to insert the OpenACC/OpenMP directives in parallel regions assigned to GPU. Although a function is set to run on the GPU in ECG, only parallelisable loops are annotated with OpenACC/OpenMP and thus

are offloaded. Functions set to run on CPU remain unchanged. The result is then compiled with `ppcg` or `clang` to produce the final executable.

Etino uses analytical performance models to capture hardware characteristics. The authors argue that some hardware characteristics might interact with each other in complex ways, hence they suggest using heuristic approaches to find the parameters. As a result, the authors propose using *Simulated Annealing (SA)* and uses a quality metric that compares the original version's performance on the CPU to the current Etino's scheduling version. The cost model starts with random values, and at each iteration, the model parameters change slightly. As the quality metric increases, meaning a speedup against the baseline, the model parameters are guiding good scheduling schemes. For tunning the cost model, Etino uses a set of benchmarks as input and the operation is done once per system. One interesting aspect in Etino is that code parameters are also tunned with SA. For instance, the loop trip count. It assumes loops execute a fixed amount of times and all branches in conditional statements run.

The evaluation results show that the careful workload partitioning in heterogeneous systems results in more considerable speedups or avoids slowdowns than other related work that offloads all parallel loops to the GPU. However, the evaluation does not compare Etino to handwritten code. That comparison would be interesting to understand the impact of disregarding data input sizes and compare manual workload partitioning against Etino to understand the efficiency in autotuned cost models with SA.

3.5 Data management optimisation

In the typical heterogeneous systems, accelerators and host CPU do not share memory. Consequently, when an application offloads computation tasks to accelerators it has to transfer data from the host's memory to the accelerator's memory. At the end of the computation in the accelerator, the results may need to be transferred back to the host. Accelerators and host CPU are usually connected via buses with limited bandwidth. Therefore, data and instruction movements may induce significant overheads that limit the profitability in offloading. Careful data management can significantly improve performance.

OpenMP, starting with version 4.0, supports offloading to accelerators. It grants implicit data movements, delegating that task to the compiler. However, compilers like Clang and GCC move all data dependencies of a kernel to GPU memory without judging data availability on the target GPU. When the kernel completes, data is sent back to the host regardless it is needed or was modified. Such approach can generate redundant communications. Consider offloading two consecutive kernels, where the result of the first one is an intermediary computation and input for the second one. The implementation in Clang/GCC sends the intermediary results back to the host, despite not being needed. Moreover, the exact intermediary data is sent to the GPU, although the data was already in memory as a result of the first kernel. This section presents work that addresses memory communication optimisations in OpenMP/OpenACC applications.

3.5.1 DawnCC

DawnCC [58], first introduced in Section 3.1, also contributes to optimising data transfers. Two consecutive kernels might use the same data, or the result of one kernel being the input for a second kernel, thus already in GPU memory. Individual kernel analysis leaves out opportunities to reduce the number of memory transfers. *DawnCC* uses data environments, supported on OpenMP and OpenACC, to solve the problem. Essentially, a data environment defines a region where some some data transfers happen when entering and leaving the region. Data that is moved to the GPU is persistent throughout the region. The data environment aggregates one or more parallel regions, therefore no communication in-between is necessary.

DawnCC uses control flow graphs (CFG), data dependency graphs, and symbolic range analysis. With the CFG and data dependency analysis, it explores the possibility for data re-use and memory that can be allocated in the accelerator for intermediary results. Besides, it uses the symbolic range analysis to reduce the amount of data transferred, sending just the necessary data instead of entire arrays.

The evaluation highlights the importance of memory transferring optimizations, noting gains up to 50% in some benchmarks. Moreover, the authors compare the explicit data management against *CUDA Unified Memory* — a runtime NVIDIA technology that manages buffer allocation and coherence between CPU and GPU memories without user intervention. The results reveal that for larger data problems, the manual memory management results is significantly better.

3.5.2 Chapman et al. tool

Chapman et al. [59] propose a tool that performs static data reuse analysis between the kernels at compile-time and makes the necessary transformations in the source code to manage data communication efficiently.

Their approach processes applications with OpenMP target directives, indicating the code region should be offloaded to the GPU. It uses LLVM infrastructure to perform static data reuse analysis between kernels on the AST level. For every target region, it performs live data analysis on the accessed variables. The tool considers two hypotheses where data re-use is very likely: *(a)* kernel call happens inside a loop, and *(b)* kernels called in the same function. After collecting the data and performing the analysis, it makes the necessary transformations inserting OpenMP data mapping clauses.

3.5.3 OpenMP Automatic Offloading (OAO)

Wang et al. [72] propose the *OpenMP Automatic Offloading* (OAO) tool. It translates shared-memory OpenMP programs, originally written for multi-core CPUs, to execute on heterogeneous systems by adding OpenMP offloading directives. Moreover, it introduces a runtime library to reduce host-device communication.

The OAO library is a set of APIs inserted automatically in the source-code to track the data state. For example, it traces if data is located on the host memory only, device only or is present

on both. In the latter case, the data state may be coherent. If it is not, then either the CPU or device have a more recent version of the data and it may be necessary to transfer it. Essentially, the data state is tracked in a state machine that transitions between states throughout the program execution. Given the data states, the OAO manages the data transferring to minimize communication as much as possible. OAO uses source-to-source techniques to insert the API calls that update the state machines state, as well as data transmission.

One advantage for the OAO runtime approach is tracking the data consistency on the application level, supporting intra-function and inter-function memory management. However, according to the authors it cannot track multilevel pointers.

3.6 Overview

Given the presented objectives in Section 1.3, the following questions are of interest to define our methodology:

- What are the possible strategies to find code regions or hotspots that may benefit from CPU parallelisation or GPU offloading?
- How to estimate the optimal target, between CPU, CPU Parallel and GPU, for computing a hotspot?
- How to transform sequential C/C++ code to run in parallel in CPU or in GPUs?

The presented literature throughout this chapter proposes different techniques to address these questions. Table 3.1 summarises the most relevant work found on literature. It compares the input for the implemented framework/tool, the programming models used, the approach for detecting hotspots, and finally, how offloading decisions are driven. Furthermore, it also indicates whether the tool implementation is available in some way, e.g. as source-code or binaries. Table 3.2 shows the benchmarks used by literature for evaluation. In case no specific benchmark suite is used, the names of the kernels are enumerated.

Table 3.1: Overview of existing literature for automatic parallelisation and offloading in CPU-GPU heterogeneous systems

Name	Input	Generates	Hotspot detection	Offloading decision	Public?
C-to-CUDA [24]	C	CUDA	Polyhedral models. Supports *doall* and *doacross* loops	Naive	✗
PPCG [71]	C	CUDA	Polyhedral models. Supports *doall* and *doacross* loops	Naive	✓[3]
DawnCC [58]	C	OpenMP / OpenACC	Classic compiler techniques. Supports pointer aliasing free *doall* loops	Naive	✓[4]
Chikin et al. [30]	OpenMP with `target` regions	Extended OpenMP runtime for dynamic offloading decisions	N.A.	Analytical performance models (CPU + GPU)	✗
HTrOP [67]	LLVM bitcode	Executable (generates OpenCL, compiles on runtime)	Polyhedral models. Affine *doall* data-parallel loops	Static and Runtime. Total number of integer/floating-point instructions above user-defined thresholds and accelerator reuse heuristic	✓[5]
Etino [65]	C	Executable binary (generates OpenACC)	Classic compiler techniques. Uses DawnCC for annotation, thus supports pointer aliasing free *doall* loops	Cost models, automatically tunned with Simulated Annealing	✓[6]

[3] https://repo.or.cz/ppcg.git
[4] https://github.com/gleisonsdm/DawnCC-Compiler
[5] https://github.com/pc2/htrop
[6] https://hub.docker.com/r/gpoesia/etino/tags

Name	Benchmarks
C-to-CUDA [24]	Coulombic Potential, N-Body Simulation, MRI Kernels, Stencil Computation Kernels, Gauss Seidel
PPCG [71]	Matrix Transpose, PolyBench 3.1 suite [75]
DawnCC [58]	PolyBench suite [75]
Amaral et al. [30]	UniBench suite (based on Polybench) [75, 56]
HTrOP [67]	2D matrix multiplication, Black-Scholes, Finite impulse response signal processing, Chain of convolutions to enhance image quality 2D simulation of heat transfer, N-body particle simulation, Motion detection, Cryptographic hash function with 256-bit digest, 3D stereo matching
Etino [65]	PolyBench [75], MgBench, Computer Language Benchmarks Game (BenchGame), DataMining Benchmark Suite

Table 3.2: Benchmarks used in literature for evaluation

3.6.1 Parallelisation strategies

Most of the presented work does hotspot detection considering loops as the primary source for intensive computing tasks. It is expected, as most scientific work performs several operations on distinct data elements. Therefore, in practice, the algorithms are implemented as loops that allow repeating the same set of instructions over different data elements. Although some approaches apply classic compiler techniques to analyse loops and determine if they are parallelisable, there is a trend for using Polyhedral models, despite its limitations for non-affine and irregular loops. In Table 3.1, only Chikin et al. [30] does not detect hotspots automatically, as the input is source code with manually annotated OpenMP regions.

Our approach is similar to existing literature in the sense that we only consider loops as potential candidates for acceleration. However, we use static analysis at AST level and classic compiler techniques to determine if loops can be parallelised.

Despite our work and some of existing literature targeting multiple devices, architecture specific tuning (e.g., loop level transformations) is not address in the literature presented in this dissertation.

3.6.2 Artefact and supported targets

Our methodology uses source-to-source techniques to generate OpenMP code with offloading support. We argue that generating modified source-code offers more flexibility for users. For instance, they can further modify the code, tuning it for specific target architectures. Another advantage for source-to-source techniques is that users can choose any compiler compliant with OpenMP, which may leverage better performance when using vendor compilers on specific platforms (e.g. IBM, Cray) [55]. As the OpenMP offloading capabilities improve and compiler implementations are more optimised, we argue it is an interesting programming model for our proposal. It reduces the invested time in generating code for targeting different vendor hardware. Besides, OpenMP is well

known. Therefore, users that may want to further modify the code may already be familiar with OpenMP.

Compared to existing literature, some approaches use source-to-source techniques as well. C-to-CUDA [24] and PPCG [71] generate CUDA for accelerating the application, thereby are limited to NVIDIA GPUs. Etino [65] generates OpenACC, which is vendor agnostic, and it is expected to work on AMD and NVIDIA GPUs. DawnCC [58] can generate OpenACC and OpenMP with offloading directives. Although their focus is GPUs, it may be possible to disable offloading through compiler flags and run the OpenMP regions in parallel in the CPU as well.

The remainder approaches generate binary files. There is a trend for using LLVM infrastructure in recent research work (see, e.g., Chikin et al. [30] and HTrOP [67]). We argue it is a limitation because such frameworks work as black-boxes from the users perspective. Besides, they are tied with the LLVM infrastructure, which we consider to be an important limitation. Chikin et al. analyses pre-annotated applications with OpenMP and targets parallelisation in multi-core CPUs and GPUs. HTrOP generates OpenCL, which has a wider support for targeting different accelerators. For instance, there is support for FPGAs. However, the experimental results consider CPUs, GPUs and Intel Xeon Phi.

Our approach considers three targets: CPU sequential (original), CPU parallel and GPU. Different OpenMP directives are inserted depending on CPU parallel or GPU target selection. Although DawnCC can be used to target parallel CPUs, it requires manual configuration at compile time via flags. Some compilers may not even support disabling offloading. Our technique generates the code for the three targets, thus no further user intervention is needed.

3.6.3 Target selection

Concerning target selection, C-to-CUDA [24], PPCG [71] and DawnCC [58] are all naive. In other words, their approaches does not perform target selection analysis. Instead, they try to parallelise all loops as long they can prove loop is absent of loop carried dependencies. Since the parallelisation strategy is naive, it may result in application slowdowns. Nonetheless, these tools are useful to facilitate the parallelisation work.

On the other hand, HTrOP [67], Etino [65] and Chikin et al. [30] use some strategy to guide target selection. HTrOP uses a simple approach that needs user intervention for tuning a threshold value to improve target selection. Etino uses cost models. Their approach is interesting since the cost model parameters are tunned by a simulated annealing algorithm. Chikin et al. uses well established analytical models from the literature to estimate the benefit of offloading OpenMP parallel regions to the GPU.

Since naive parallelisation may result in slowdowns, our methodology uses analytical models to guide target selection. We use the same analytical models as Chikin et al. We contribute to the GPU analytical model to address some of its limitations. Chikin et al. also explores using LLVM MCA [2] for estimating computational cost in the CPU, which is one of the parameters for the CPU analytical model. LLVM MCA simulates the superscalar CPUs pipeline execution, analysing the code at assembly level. Therefore, it is expected to generate accurate estimations

compared to our AST level analysis. We also identified some limitations in their LLVM MCA approach and we propose some refinements.

3.6.4 Static analysis

We use AutoPar [21], a library integrated in Clava [25], that performs static analysis to determine if loops are parallelisable. Our approach uses static analsysis at AST level to collect the metrics necessary for the analytical models and evaluate the relative performance between CPU, CPU Parallel and GPU. Code generation and target selection is done at compile time.

HTrOP [67] and Chikin et al. [30] propose hybrid approaches. Both collect some information at compile time. The HTrOP approach prepares the OpenCL version for selected parallel regions at compile time. But, the offloading decision is always delayed to runtime. Their runtime selects the target accelerator considering the available devices in the system, cost estimation and other heuristics. After selecting the target, the OpenCL kernel is compiled. Although the authors try to minimise the induced overhead due to on the fly compilation, there is an impact in the overall execution time. Nonetheless, one advantage for their approach is that in production systems the available accelerators may change. Hence the motivation for compiling the OpenCL at runtime considering what is effectively available in the system. With respect to Chikin et al., they delay the offloading decision to runtime if some parameters are statically unknown (e.g., loop trip counts). In contrast to HTrOP, they generate one OpenMP version that may either target GPUs or parallel CPUs at compile time. Therefore, the compiler always generates code for both targets. At runtime, the analytical models are evaluated for selecting the target and no further compilation is needed.

3.7 Summary

This chapter presented a literature review on heterogeneous computing, particularly CPU-GPU systems. The presented work aims to ease the adoption of such systems.

Some methodologies propose skeleton programming or special languages for abstracting parallel and GPU programming details. However, the developer has to adapt the existing code base manually. Moreover, skeletons are not general enough, and therefore it might be difficult to express an algorithm on the available skeletons.

Other works propose automatic frameworks that parallelise sequential code, partition the workloads and optimise data communication. Some are source-to-source approaches that try to parallelise or offload all loops without estimating if there is any benefit performance or energy wise. Other strategies use heuristics or analytical models to guide offloading decisions.

Compared to existing literature, our approach *(a)* uses fully static analysis to prevent overheads while the application is running, *(b)* the analysis is done at AST level and is compiler agnostic, *(c)* uses source-to-source techniques, ensuring maximum flexibility for the user, and *(d)* we adapt analytical models from existing literature for selecting the optimal target and address potential limitations.

Chapter 4

CPU Analytical Models

This chapter introduces the analytical models used to estimate kernel's execution times on the CPU in two scenarios: *(a)* sequential, using a single CPU core, and *(b)* parallel with OpenMP on physical cores.

The models implemented in this dissertation for CPU targets are based on Liao and Chapman's OpenMP model [51]. Liao and Chapman estimate computational costs by counting operations at the OpenUH's Intermediate Representation (IR) [29] and mapping IR operations to machine instructions with preset scheduling tables. Chikin et al. [30] uses the same CPU model; however, the authors experiment using LLVM Machine Code Analyser (MCA) [2] to estimate computational cycles, replacing some parts of the cost model.

This dissertation does not contribute to the analytical models — it only makes some adjustments where needed. However, we propose different approaches for estimating the cost of a code region, an input parameter for the models.

- An AST-level analysis approach. Although less accurate than analysis in lower-level representations, our aim is optimal target selection rather than accurate time estimations. Operations counted at the AST are mapped to ISA instructions and then available latency and throughput tables are used to estimate the elapsed cycles.

- Use `llvm-mca` [2], as proposed by Chikin et al. [30], but addressing some limitations in their method. A lower-level analysis tool allows to compare target selections against our AST-based approach.

This chapter is organised as follows. The first sections introduces the OpenMP model proposed by Liao and Chapman [51]. The following section presents the models implemented in this work, based on Liao and Chapman. Then we present the two approaches for estimating computational cost: first, the AST-based analysis and then using `llvm-mca` [30].

4.1 Background: Cost-models for CPU

Equation 4.1 shows the Liao and Chapman [51] cost model for OpenMP parallel regions targeting shared-memory systems. The model is the basis for the analytical model implemented in this dissertation, presented later in Section 4.2. The remainder of this section describes the most relevant parts of the Liao and Chapman model for this dissertation.

$$
\begin{aligned}
Parallel\_region_c &= Fork_c + \sum_{j=1}^{m} [max(Thread_0\_exe\_j_c, ..., Thread_{n-1}\_exe\_j_c)] + Join_c \\
Thread_i\_exe\_j_c &= Worksharing_c + Synchronisation_c \\
Worksharing_c &= Parallel\_for_c \mid Parallel\_section_c \mid Single_c \\
Synchronisation_c &= Master_c \mid Critical_c \mid Barrier_c \mid Atomic_c \mid Flush_c \mid Lock_c \\
Parallel\_for_c &= Schedule\_times \times (Schedule_c + Loop\_chunck_c + Ordered_c + Reduction_c) \\
Loop\_chunck_c &= Machine_c\_per\_iter \times Chunk\_size + Cache_c + Loop\_overhead_c
\end{aligned}
\tag{4.1}
$$

4.1.1 The fork-join model

The OpenMP programming model is a fork-join model. When the initial thread encounters a parallel region, such as the `omp parallel`, it calls routines to initialise a team of threads that start idle. Launching the threads has a computational cost of $Fork_c$ cycles, which varies from system to system and the number of threads launched [29]. There is an implicit synchronisation barrier at the end of the parallel region to ensure all threads terminate their work before the main thread continues executing the program beyond the parallel region [10]. The cost to terminate the parallel region and destroy the threads is $Join_c$.

A parallel region may have one or more work distribution constructs. The summation in $Parallel\_region_c$ accumulates the cost for m workshare regions. Each workshare region distributes work by the threads participating in the parallel region. As the n threads in the team operate in parallel, only the longest-running thread defines the workshare execution time, as expressed by $max(Thread_0\_exe\_j_c, ..., Thread_{n-1}\_exe\_j_c)$.

The overall cost for each parallel construct in OpenMP, in Equation 4.1, is $Parallel\_region_c$ cycles.

4.1.2 Synchronisation

In many parallel algorithms, some operations are not independent or must happen in a specific order. In those cases, algorithms are designed to exploit parallelism as possible, but eventually, threads perform operations serially. For instance, threads that write to the same variable must

access it atomically (i.e., one thread at a time). OpenMP has several constructs to enable synchronisation that introduce overheads as denoted by $Synchronisation_c$ in Equation 4.1. Explicit synchronisation barriers are not considered in this dissertation.

4.1.3 Workshare Region

Inside the parallel regions, OpenMP workshare constructs are used to assign tasks to threads. The most common example is the `omp for` construct used in for-loops. According to a scheduling policy, it splits the iteration space and assigns chunks of iterations to threads.

The parameter $Worksharing_c$ in Equation 4.1 models different OpenMP constructs that may be used in parallel regions. For this dissertation, we focus in the parallel for construct, $Parallel\_for_c$, used in for-loops.

4.1.4 Parallel For-Loops

The cost of a parallel for construct, $Parallel\_for_c$ (Equation 4.1), depends on *(a)* the computational cost to execute one chunk of iterations, $Loop\_chunck_c$, *(b)* overheads due to scheduling, *(c)* reduction operations, and *(d)* serialised sections in the loop body (e.g., `omp ordered`). OpenMP has various scheduling policies, and the number of times chunks are assigned to a thread, $Schedule\_times$, may vary.

The static and dynamic scheduling policies are the most common ones. With dynamic scheduling, the threads are allocated an initial amount of work, and whenever one thread finishes its current chunk, it requests more work. The OpenMP runtime monitors the progress and continuously assigns more work to threads when requested. In contrast, static scheduling divides the iteration space into even chunks of work at compile-time. Each thread determines its chunks with simple arithmetic, using the thread identifier and the number of participating threads. Therefore, runtime scheduling policies may allocate chunks to threads multiple times, i.e., $Schedule\_times \geq 1$. For the static policy, $Schedule\_times = 1$. Furthermore, costs associated with scheduling, $Schedule_c$, are negligible in the static policy.

The cost for computing a chunk of iterations, $Loop\_chunck_c$, depends on the computational cost of one iteration, $Machine_c\_per\_iter$, multiplied by the chunk size. Liao and Chapman's model (Equation 4.1) also adds cache access cycles, $Cache_c$, and overheads on loop control flow, $Loop\_overhead_c$. The loop overhead is the cost for incrementing the control variable and testing the conditional expression for entering the loop.

4.1.5 Computational cost per loop iteration

To determine $Machine_c\_per\_iter$, Liao and Chapman use a cost-model from the OpenUH compiler [51, 29], inherited from the Open64 compiler [73]. Equation 4.2 shows the cost model to

determine $Machine_c\_per\_iter$.

$$
\begin{aligned}
Machine_c\_per\_iter &= Resource_c + Dependency\_latency_c + Register\_spilling_c \\
Resource_c &= maximum(FP_c, ALU_c, Load\_Store_c, Issue_c) \\
FP_c &= \frac{total\_FP_c}{\#FP\_units} \\
ALU_c &= \frac{total\_ALU_c}{\#ALU\_units} \\
Load\_Store_c &= \frac{total\_Load\_Store_c}{\#Load\_Store\_units} \\
Issue_c &= \frac{Num\_inst}{Issue\_rate}
\end{aligned}
\tag{4.2}
$$

The cost model has three main parts, described below.

$Resource_c$. It is the number of cycles required to compute instructions in the execution units. It assumes no instruction dependencies and does not account for cycles spent in memory accesses that could stall execution. In OpenUH [51, 29], instructions are counted at the IR level and then mapped to target machine instructions to accumulate the respective latencies. The model assumes CPU architectures where the scheduler unit issues machine instructions to distinct execution units, e.g., Arithmetic Logic Unit (ALU), Floating Point (FP). Multiple execution units for different instruction types enables instruction-level parallelism. Therefore, the model estimates the total number of cycles per operation type — e.g., $total\_FP_c$ for floating-point operations —, and divides it by the number of available units to process those operations. A possible bottleneck in the CPU back-end is the instruction dispatcher, which only issues $Issue\_rate$ instructions per cycle. $Resource_c$ is the maximum number of cycles elapsed among the independent executions units and the dispatcher unit.

$Dependency\_latency_c$. The number of cycles the CPU has to stall execution due to dependencies between instructions. A dependence graph is used to make the estimation. During stall periods, the CPU is not doing valuable work, extending the execution time.

$Register\_spilling_c$. When $Resource_c$ is calculated, there is an assumption that all instruction's operands are in registers. However, during the register allocation phase in the compiler, some variables may not be stored in a register and are spilled to memory. $Register\_spilling_c$ is the number of cycles spent accessing the memory hierarchy considering the register allocation.

The details for register spilling and dependency latency are omitted in Equation 4.2, as they are not considered in this dissertation.

4.2 Implemented analytical models

The previous section discussed the cost models used in the Open64/OpenUH compilers [73, 29] and the OpenMP model proposed by Liao and Chapman [51]. This section presents the models implemented in this work, based on Liao and Chapman model.

The main adjustment needed is for calculating the number of elapsed cycles in a sequence of instructions, $Machine_c\_per\_iter$, as the original models do not suit modern x86 CPU architectures. Furthermore, as our work only addresses a subset of OpenMP constructs, parts of the model are omitted for simplification.

4.2.1 Execution units and instruction issuing

The cost models in Open64 [73] are evaluated in a MIPS R10000 processor [74]; Liao and Chapman evaluate OpenUH's models [51, 29] in an Itanium 2 processor from Intel [57]. The scheduler units in both CPUs issue instructions to isolated execution units that operate in parallel. For example, consider the Itanium 2 core architecture illustrated in Figure 4.1. Decoded instructions are queued per operation type: `B` for branching, `M` for memory operations, `I` for integer arithmetic, and `F` are for Floating-Point instructions. Moreover, there are dedicated execution units for branching, integer and floating-point, as shown in the diagram bottom.

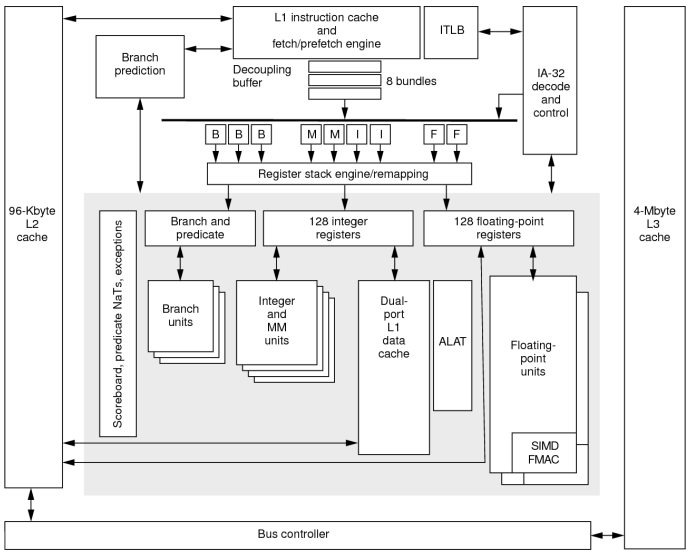


Figure 4.1: The Itanium 2 core architecture [57]

Modern Intel/AMD x86 core architectures arrange the execution units differently. The scheduler issues instructions to ports that interconnect to private and manifold execution units. The designation "private execution unit" means an execution unit in port p is not shared with any other port p'. Each has a different set of execution units. Figure 4.2 illustrates the Intel Skylake core architecture.

Furthermore, execution units are more specialised and go beyond a floating-point, integer and memory operations distinction. In Skylake micro-architecture, only port 0 can execute integer and

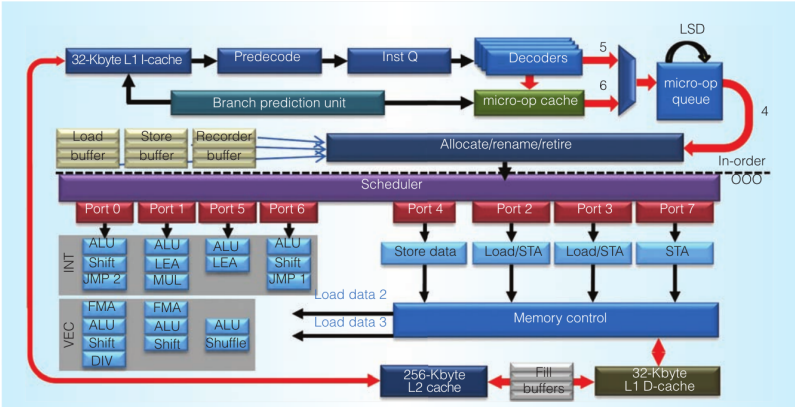


Figure 4.2: Intel 6th gen., Skylake, core architecture [34]

floating-point division; ports 0, 1, 5 and 6 support integer arithmetic (ALU); a shift operation is only supported in ports 0, 1 and 6, even though it is commonly executed in ALU units [17, 20].

The issue is even more complex as the same instruction from the x86 ISA may generate distinct micro-ops depending on the operand type, requiring different number of ports and in different orders (assuming dependencies between micro-ops). Consider the integer multiplication, `IMUL`, and the Skylake micro-architecture. The `IMUL` variant for memory operands of 16 bits results in 5 micro-ops and uses the ports $2 * p0156 + 1 * p06 + 1 * p1 + 1 * p23$. The variant for 64 bits operands is decoded in 3 micro-ops that use the ports $1 * p1 + 1 * p23 + 1 * p5$. The presented notation is the same as used by *uops.info* [20], where $1 * p1$ means one micro-op that can only execute in port 1, $p1$, and $2 * p06$ means two micro-ops, that can be issued to either port 0 or 6.

Given the presented issues, the model proposed by Liao and Chapman [51, 29] needs adjustments to model modern x86 superscalar architectures, which is the target of this dissertation. Note that execution pipelines similar to MIPS R10000 or Itanium are still used. A recent example is the Berkeley Out-of-Order Machine (BOOM) [76] which implements the RISC-V ISA and is heavily inspired by the MIPS R10000 processor. Therefore, different CPUs need specific analytical models.

4.2.2 Modelling instructions costs in x86 architectures

This dissertation hypothesises analysing code regions at the AST level, e.g., counting operations. We argue is not feasible to accurately model port usage and dependencies between micro-ops, similarly to what `llvm-mca` does (see Section 2.1), with information available at the AST level. To correctly model the execution steps in the CPU back-end, it is necessary to analyse lower-level representations such as low-IR (assuming a near one to one mapping to ISA instructions) or assembly. Nonetheless, it may be possible to relax instruction cost estimations in a relative performance modelling context. We propose using AST analysis to count operations, map them from AST semantics to an ISA instruction, and use existing instruction cost tables.

Two metrics are usually considered for elapsed cycle estimations of instructions:

Latency. It is the number of cycles required to execute an instruction, from the moment the scheduler unit issues it until it completes execution. Due to the Out-of-Order (OOO) nature of the processors, more cycles may be required to retire/commit the instruction architecturally, as instructions are retired in the program's order.

Throughput. The number of instructions that can be issued per clock cycle or the number of instruction's results per clock cycle. An alternative concept is the reciprocal throughput, defined by Agner Fog [40] as the average cycles per instruction for a series of independent instructions of the same type. Consider four pipelined ALU units and integer addition instructions, `ADD`. According to the first definition, the throughput for `ADD` is four instructions per cycle because each ALU unit can receive a new instruction simultaneously every clock cycle. According to Fogs's definition, the reciprocal throughput for `ADD` is 0.25 average cycles per instruction as four units execute the additions in parallel [20].

Neither of these metrics is by itself accurate for cycle estimations. Accumulating latencies per instruction does not model the pipelining of the execution units and does not consider the number of ports/units for that instruction. On the other hand, throughput can be overly optimistic because it is often impossible to saturate the pipelines, either because of the nature of the program that lacks a chain of instructions of the same kind or because of dependencies among the instructions.

Estimations in this work focus on loop nests where the same block of instructions is executed multiple times. Considering all the mechanisms inside a CPU to minimise the latency of executing instructions — e.g., branch predictors, Out-of-Order execution, and pipelining —, the observed cycles spent in computation should be closer to throughput rather than accumulating latencies.

Equation 4.3 presents our approach for estimating elapsed cycles executing instructions, using instructions reciprocal throughput. Similarly to the Liao and Chapman model in Equation 4.1, $Machine\_per\_iter_c$ represents the computational cost for one iteration of the loop body.

$$\begin{aligned} Machine\_per\_iter_c &= max(Instruction_c, Issue_c) \\ Instruction_c &= \sum_{I \in Instructions} rTroughput(I) \\ totalMicroOps &= \sum_{I \in Instructions} microOps(I) \\ Issue_c &= \frac{totalMicroOps}{Issue\_rate} \end{aligned} \tag{4.3}$$

The $Instructions$ is a list of ISA instructions in the loop body. The reciprocal throughput, $rTroughput(I)$, for all ISA instructions, I, is accumulated, giving a lower bound in cycles necessary to execute all instructions — $Instruction_c$. Note that the reciprocal throughput for an instruction I is the average number of clock cycles to execute it and already models the available instruction-level parallelism.

Scheduling units can only issue $Issue\_rate$ micro-ops per clock cycle. If the scheduler unit cannot feed the execution units fast enough, it bottlenecks the back-end. Therefore, the elapsed cycles on the back-end are determined by the issuing rate and the elapsed cycles in execution units.

4.2.3 CPU Sequential and Parallel Models

The analytical models implemented for sequential and parallel execution in CPU are presented in Equation 4.4 and Equation 4.5, respectively.

$$\begin{aligned} Sequential_c &= Loop_c + Loop\_overhead_c \\ Loop_c &= Machine\_per\_iter_c \times Num\_loop\_iter \\ Loop\_overhead_c &= Loop\_overhead\_per\_iter_c \times Num\_loop\_iter \end{aligned} \tag{4.4}$$

$$\begin{aligned} Parallel_c &= OmpParallel_c + \sum_{i=1}^{\#wr}(WorkshareRegion\_i_c) \\ WorkshareRegion\_i_c &= Loop_c + Loop\_overhead_c \\ Loop_c &= Machine\_per\_iter_c \times Chunk\_size \\ Loop\_overhead_c &= Loop\_overhead\_per\_iter_c \times Chunk\_size \\ Chunk\_size &= \frac{Num\_loop\_iter}{Num\_threads} \end{aligned} \tag{4.5}$$

Compared to the original models by Liao and Chapman [51], we changed the equations to determine $Machine\_per\_iter_c$, to better suit modern x86 architectures. Concerning the parallel model, Equation 4.5, this dissertation focus on the parallel worksharing loops using the `omp parallel for` construct. OpenMP constructs related to other workshare constructs and synchronisation barriers are not included in our implementation. Scheduling overheads are omitted as we only consider static scheduling, with negligible overhead.

4.2.4 Analytical Model parameters

The analytical models have device and application dependent parameters. The device-specific parameters are summarised in Table 4.1.

Parameter	Unit	Description
$rTroughput(I)$	cycles	A map of ISA instructions to reciprocal throughput, i.e. average number of cycles to execute the instruction
$microOps(I)$	u-ops	A map of ISA instructions to the number of resulting micro-ops after decoding in the front-end
$Issue\_rate$	u-ops/cycle	Maximum number of micro-ops that can be issued per clock cycle, from the front-end to the back-end
$OmpParallel\_c(T)$	cycles	The overhead for creating a team of T threads in a omp parallel construct
$Loop\_overhead\_c$	cycles	Average cost per iteration in a loop

Table 4.1: Device dependant parameters for the CPU analytical models

Overheads associated with OpenMP constructs can be obtained using the EPCC OpenMP micro-benchmark [26]. The overhead associated with the `parallel for` static scheduling construct, $OmpParallel_c$, varies with the number of threads launched. In this work, the number of

threads is either the number of physical or logical cores, hence the OpenMP micro-benchmark is used for those number of threads.

Liao and Chapman [51] uses a fixed cost for the common loop header, $Loop\_overhead\_per\_iter_c$: `for(int i = 0; i < N; i++)`. The same approach is implemented in our work. The resulting assembly consists of three instructions. One that increments/decrements the induction variable, another to make the comparison with `N`, and finally the jump instruction. One can use `llvm-mca` to estimate the cost of executing that sequence of instructions or simply add the latencies.

Information regarding latencies, throughputs and number of micro-ops can be obtained from existing tables [40, 20, 17].

4.3 Using AST-level analysis for cycle estimations

The previous section established the analytical model that is used in this work for CPU sequential and parallel estimations. This section presents our approach using AST-level analysis to estimate the computational cost of a loop iteration. We count operations at AST level and then map to equivalent ISA instructions — *Instructions* in Equation 4.3.

4.3.1 Counting operations at AST

For AST analysis, we use Clava [25], a source-to-source compiler that abstracts complex compiler infrastructures. It allows to query and transform the AST via a domain-specific language called LARA [28, 27]. The AST is traversed on the code region of interest using LARA scripts for counting the operations and memory accesses. Each operation is characterised by a type (e.g. addition, multiplication), a bitwidth (8, 16, 32, 64 bits) and a flag to distinguish between integer and floating-point data types. Our analysis estimates loop trip counts to precisely count operations in nested loops. Details concerning our AST level analysis and related issues are discussed in Section 7.4.

4.3.2 Mapping AST operations to ISA

In Section 4.2.2, we proposed using existing latency and throughput tables [40, 20, 17] to estimate the necessary cycles to execute a sequence of instructions. As the tables provide costs for ISA instructions, our approach has to map AST operations to equivalent instructions in the ISA.

Mapping AST-level operations to ISA instructions is complicated, as it depends on how the compiler generates code, applied optimisations, and some ISA extensions being enabled or disabled by the user. Furthermore, there are many instructions variants for the same arithmetic operation. For instance, some instructions have versions per operand's bitwidth, operand type (memory, register, immediate or literal) and whether the operation is signed or unsigned. It is challenging to guess precisely what instruction the compiler selects from AST-level analysis.

Table 4.2 shows latencies and reciprocal throughput for some of the integer signed (`IDIV`) and unsigned (`DIV`) division instructions available in x86 ISA. The difference between `DIV` and `IDIV`

for the same operand type is not very large, but for different operands bitwidth the 64-bit version takes 3.5× more cycles on average than the 32-bit variant.

Instruction	Latency (cycles)	Reciprocal Throughput (avg. cycles per instruction)
`IDIV (M64)`	[38, 102]	24.14
`IDIV (M32)`	[23, 34]	6
`DIV (M64)`	[6, 95]	21
`DIV (M32)`	[23, 34]	6

Table 4.2: Latency and throughput for a subset of integer division instructions in Intel Skylake architecture [20]

Our approach only distinguishes instructions per operands bitwidth and assumes operands are in registers. For the division example, we do not differentiate between unsigned or signed divisions and the average between the two is used. Our approach only considers arithmetic, relational and logical operations. SIMD and other extensions to the ISA are not considered.

Our approach has limitations that may compromise the accuracy of the estimations. As a similar approach is used for the GPU estimations and given the relative performance analysis context, the introduced error may be acceptable to guide optimal target selection.

4.4 Using LLVM Machine Code Analyser (MCA)

The second approach for estimating the number of cycles per iteration, $Machine_c\_per\_iter$, is using the LLVM Machine Code Analyser [2] (see Section 2.1 for details). To the best of our knowledge, Chikin et al. [30] are the first to use `llvm-mca` for performance prediction in research work. The `llvm-mca` tool is used in this work to compare against our approach based on AST-level analysis. Besides, we address limitations in Chikin et al. approach [30].

4.4.1 Limitations in Chikin et al. approach

The following limitations were identified in Chikin et al. [30] approach:

1. The authors extract the parallel loop's body and use `llvm-mca` to estimate the cycle cost of one iteration. However, in the presence of loops in the region under analysis, `llvm-mca` only simulates one iteration as it does not simulate control-flow (see Section 2.1 for limitations with `llvm-mca`). Therefore, a more thorough analysis is needed to reduce errors in the estimations.

2. It is assumed that loops within the parallel loop body have 128 iterations. However, the relative performance between distinct targets is not constant.

Control flow limitation

The first limitation has to do with the lack of control-flow simulation in `llvm-mca`. As explained in the Section 2.1, the tool takes as input a sequence of assembly instructions that are simulated in order. Instructions that modify the next instruction address, e.g. `call` or `jmp`, consume resources in the execution engine, but any other side effect is not considered.

To illustrate it, consider the assembly snippet in Listing 4.1, showing a loop with 100 iterations that increment the value in register `eax`.

```
.intel_syntax
mov ecx, 100

loop:
   add eax, 1
   dec ecx
   jnz loop
```

Listing 4.1: Simple loop in x86 Assembly

If `llvm-mca` simulated control-flow, the execution engine would execute $3 \times 100 + 1 = 301$ instructions. However, the `llvm-mca` reports 4 instructions, as shown in Figure 4.3. For clarification, the command-line parameter `-iterations` is the number of passes to use in the simulation. It is handy for a repeating sequence of instructions to explore the pipelining effect. The value of 1 is used to make a single pass on the sequence of instructions and prove that the jumps and register state are not simulated.

```
$ llvm-mca --iterations 1 demo.asm

Iterations:        1
Instructions:      4
Total Cycles:      6
Total uOps:        4
```

Figure 4.3: LLVM MCA report for the assembly excerpt in Listing 4.1

To ensure an accurate cycle estimation, a more careful approach is needed when using `llvm-mca` for code regions that may contain loops.

Constant loop trip count limitation

The second limitation in the Chikin et al. [30] approach is assuming that all loops within the parallel code region execute 128 iterations. Assuming a constant trip count is a significant limitation that the authors acknowledge. Although it could make sense in a relative performance analysis context, the performance ratio between two targets is not constant as the problem size varies.

Consider the matrix multiplication algorithm in Listing 4.2, with N^3 temporal complexity. The OpenMP pragma, `omp parallel for`, indicates that only the outermost loop, `i`, is parallelised.

Suppose a constant trip count is used for the loops `j` and `k`, like Chikin et al. [30] approach. In that case, the problem's complexity is $N \times 128 \times 128$, reducing considerably the number of estimated operations each thread would perform. The error in the estimation can affect the offloading decision, for instance, it could spoil the estimative for GPU due to lack of work to justify the offloading.

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
   for (int j = 0; j < N; j++) {
      C[i * N + j] = 0.0;
      for (int k = 0; k < N; ++k) {
         C[i * N + j] += A[i * N + k] * B[k * N + j];
      }
   }
}
```

Listing 4.2: Matrix multiplication with column-major access in matrix B

Besides the direct impact in computational cost estimation, other target differentiator properties depend on the number of operations. For instance, consider the memory subsystem in a CPU versus a GPU. The matrix multiplication in Listing 4.2 is not cache-oblivious as every memory access to `B[k * NJ + j]` will result in a cache miss.

In CPUs, memory accesses like the example are substantial performance penalties. One of the instructions will always have an operand dependency that takes too long to serve due to access to the main memory. Besides, the loop has a small instruction intensity for overlapping memory accesses. Thus, even with branch prediction mechanisms, the execution engine will eventually stall the thread's execution.

GPUs operate differently as they constantly pre-empt warps. Thus, assuming there are sufficient warps to keep execution units busy, while one warp accesses memory, other warps can execute. By the time the first warp gets a chance to execute again, perhaps the data has been fetched from memory, and the warp is in a ready state. The execution mechanism in a GPU allows hiding the cost of accessing memory to some extent.

CPUs are generally more sensitive to cache misses than a GPU. To model the performance impact of cache misses, it is necessary to estimate the number of memory accesses and how many may result in misses. In the example, the number of cache misses is a function of the loop's trip count. Assume a hypothetical analytical model that accurately estimates cache misses. If the number of memory accesses is orders of magnitude below the reality, then the model's estimation could be wrongly optimistic and favour the CPU, guiding wrong offloading decisions.

The conclusion is that even in a relative performance context, it is fundamental to feed the models with input parameters as accurately as possible. The problem is finding a good balance between accuracy and how detailed the models must be. The more complex and architecture-dependent the models are, the less portable they are, as models need to be generated for each target architecture.

4.4.2 Potential pitfalls

In addition to the limitations found in Chikin et al. [30] approach, some pitfalls were identified when using `llvm-mca` and may affect the estimations accuracy.

Analysing specific blocks of code

Passing an assembly file as input to `llvm-mca` makes it analyse the entire sequence of assembly instructions. In our context, it is necessary to reduce the scope of the analysis to portions of code, e.g. a loop body.

To reduce the scope of analysis to blocks of code, `llvm-mca` supports comments in the assembly file to define the boundaries of the region of interest: `# LLVM-MCA-BEGIN` and `# LLVM-MCA-END`. It is possible to insert the annotations at the source-code level using the compiler directives for inline assembly: `__asm volatile("# LLVM-MCA-BEGIN kernel")`.

In our approach, the analysis is automated using Clava [25]. The inline assembly is inserted temporarily for analysis, then the code is compiled to generate assembly and finally `llvm-mca` is invoked to generate a report. Implementation details are revisited in Section 7.4.3.

The problem is that compilers assume code in `__asm` directives to have side-effects, impacting the code generation for statements around it as it may inhibit specific optimisations. In other words, compiling the source code with and without the annotations may result in a different assembly code. Although the inline assembly is just a comment, the assembly template is not parsed, according to GCC documentation[1]. Consequently, the compiler does not know the inline assembly, `__asm volatile("# LLVM-MCA-BEGIN kernel")`, is just a comment without any side-effect. Clang handles inline assembly like GCC, thus has the same issue [2].

Impact of inline assembly in loop-level optimisations

The hotspots considered in this work are loop nests, and loops have the potential for various optimisations, e.g. vectorisation. Depending on how the inline assembly directives are inserted in the source code, it may inhibit optimisations and affect the compiler outcome.

Our aim, similarly to Chikin et al. [30] approach, is using `llvm-mca` is to estimate the cost for one parallel iteration — $Machine\_per\_iter_c$ in Equation 4.4 and 4.5. Therefore, the `llvm-mca` directives are inserted in the parallel loops for analysis and two approaches are possible: *(a)* insert inside the loop's body, and *(b)* wrap the entire loop, including header and body.

Listing 4.3 illustrates the first approach where the annotations are inserted inside the loop body. The issue is it creates an inter-iteration dependency, i.e., the $i+1$th iteration depends on the ith iteration, and optimisations are inhibited.

[1] "GCC does not parse the assembler instructions themselves and does not know what they mean or even whether they are valid assembler input." (https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#AssemblerTemplate)

[2] https://clang.llvm.org/docs/LanguageExtensions.html#asm-goto-with-output-constraints

```
#define LLVM_MCA_BEGIN(name) __asm volatile("# LLVM-MCA-BEGIN " name)
#define LLVM_MCA_END(name) __asm volatile("# LLVM-MCA-END " name)
void vecAdd(float *A, float *B, float ALPHA) {
   for (int i = 0; i < N; i++) {
      LLVM_MCA_BEGIN("kernel");
      A[i] = A[i] + B[i] * ALPHA;
      LLVM_MCA_END("kernel");
   }
}
```

Listing 4.3: Inserting directives for `llvm-mca` inside the loop body

The second approach inserts the annotations around the entire loop, wrapping both the header and body, as presented in Listing 4.4. In this approach, the compiler can perform local loop-level optimisations.

```
#define LLVM_MCA_BEGIN(name) __asm volatile("# LLVM-MCA-BEGIN " name)
#define LLVM_MCA_END(name) __asm volatile("# LLVM-MCA-END " name)
void vecAdd(float *A, float *B, float ALPHA) {
   LLVM_MCA_BEGIN("kernel");
   for (int i = 0; i < N; i++) {
      A[i] = A[i] + B[i] * ALPHA;
   }
   LLVM_MCA_END("kernel");
}
```

Listing 4.4: Inserting directives for `llvm-mca` around the loop

Although the second approach overcomes the compiler optimisation problem, it has other cons. For instance, it might restrict inter-loop optimisations, such as loop fusion. Adding the directives in the innermost loops of a loop nest may create dependencies in the outer loops. Moreover, the compiler may create various loop versions that end up being surrounded by the directives in the assembly output, compromising the `llvm-mca` estimation accuracy. There are different reasons for loop versioning. For example, if the compiler cannot prove pointers are aliasing free, i.e. the memory regions pointed by two distinct pointers do not overlap. Consequently, the compiler generates two loop versions. One version assumes pointers are aliasing free and loop optimisations are safe to apply. A stricter version assumes pointer aliasing resulting in data dependencies that prevent optimisations such as vectorisation. The compiler injects code to compare the memory boundaries referenced by pointers at runtime and then selects the loop version accordingly. In case the various loop versions are within the annotated region, the `llvm-mca` estimation represents executing one iteration of all loop versions, impacting the accuracy.

Modelling loop-level optimisations

To address the constant loop trip count limitation in Chikin et al. [30] approach, where the authors assume loops in the parallel region to execute 128 iterations, we propose determining the loop trip count accurately whenever possible. The cost for one loop iteration, as estimated by `llvm-mca`,

is multiplied by the trip count to calculate the total loop cost. However, it is crucial to account for loop optimisations.

Suppose a loop with N iterations in the source code. Assuming no optimisations are applied, the assembly instructions corresponding to the loop's body are executed N times. When the compiler applies optimisations, such as unrolling, the resulting assembly may be equivalent to more than one iteration, reducing the number of jumps. In those cases, the estimated cost by `llvm-mca` is not for one iteration but rather f iterations, with f being an optimisation factor.

To demonstrate the issue, consider the vector addition code presented in Listing 4.4. When compiling the code with Clang and `-O3`, the optimisation report indicates the following loop-level optimisations, by appliance order: *(a)* vectorised with a width of 4 and interleaving with a factor of 2, and *(b)* unrolled by a factor of 2.

Listing 4.5 shows the resulting assembly. The instructions are commented with letters to identify the different computing steps. For instance, `(A)` and `(B)` are vectorised instructions moving packs of 4 elements from the vectors in memory to 128-bit registers. The register `xmm0` holds the constant `ALPHA` value, and is multiplied in parallel by the four elements of vector `B` in step `(C)`. The step `(D)` is the vector addition. The remainder steps store the result in memory and are omitted for simplification. The interleaving pass is applied after the vectorisation, and essentially mixes two vectorised iterations. Instructions due interleaving are annotated with apostrophes. Note how memory accesses in `(A')` and `(B')` have an offset of 16, because each vectorised iteration computes 128-bits of data, i.e. 16 bytes. The final optimisation pass applies unrolling over the already vectorised and interleaved version of the loop. Therefore, the first range of steps, `[A-D']` is basically duplicated.

The `rax` register in Listing 4.5 is the loop control variable. With all optimisations applied, the register is increment by 16, meaning the sequence of assembly instructions in `loop` are equivalent to computing 16 iterations. The optimisation factor, f, can be determined multiplying the optimisation factors, i.e. $f = vecWidth \times interleaveFactor \times unrollFactor$. Note the `llvm-mca` directives surround the entire loop block, thus the estimation, *llvmCost*, is for 16 iterations. Given that our approach is based on AST-level analysis, the correct approach for estimating the total loop costs with `llvm-mca` is $llvmCost \times \frac{N}{f}$, where N is the loop trip count determined with AST analysis.

4.4.3 Proposed approach

The previous sections presented various challenges to correctly use `llvm-mca` along with source-to-source techniques. It is necessary to insert the directives in a way that does not interfere with the compilation flow and account for loop-level optimisations when computing the total loop cost.

The solution we found to work the best is to split the code region under analysis in smaller units, which we refer as *basic blocks*. Our approach recursively traverses the AST, with Clava [25], and whenever it finds a loop it creates three basic blocks: one before the loop statement, one after the loop, and one that surrounds the loop itself. `llvm-mca` is used to get the cost for each

```
1  # LLVM-MCA-BEGIN kernel
2  ...
3  .loop:
4     movups xmm1, xmmword ptr [rdi + 4*rax]    # (A)
5     movups xmm2, xmmword ptr [rdi + 4*rax + 16] # (A')
6     movups xmm3, xmmword ptr [rsi + 4*rax]    # (B)
7     movups xmm4, xmmword ptr [rsi + 4*rax + 16] # (B')
8     mulps xmm3, xmm0        # (C)
9     addps xmm3, xmm1        # (D)
10    mulps xmm4, xmm0        # (C')
11    addps xmm4, xmm2        # (D')
12    ...
13    movups xmm1, xmmword ptr [rdi + 4*rax + 32] # (A)
14    movups xmm2, xmmword ptr [rdi + 4*rax + 48] # (A')
15    movups xmm3, xmmword ptr [rsi + 4*rax + 32] # (B)
16    movups xmm4, xmmword ptr [rsi + 4*rax + 48] # (B')
17    mulps xmm3, xmm0        # (C)
18    addps xmm3, xmm1        # (D)
19    mulps xmm4, xmm0        # (C')
20    addps xmm4, xmm2        # (D')
21    ...
22    add rax, 16
23    cmp rax, 4096
24    jne .loop
25 ...
26 # LLVM-MCA-END kernel
```

Listing 4.5: Partial resulting assembly for the vector addition in Listing 4.4. Compiled with Clang and -O3. The problem size, `N`, is set to 4096.

basic block and the costs are accumulated recursively. Basic blocks that correspond to a loop, the cost is multiplied by the loop's trip count considering eventual loop optimisation factors.

Algorithm 1 demonstrates our approach. The function `basicBlockCost` gets a starting AST scope node, $scope_n$. The first child in $scope_n$ starts a new basic block, *BB* (line 2). If no loops are found, then the last child of $scope_n$ is marked as the ending node for *BB* (line 13). `llvm-mca` is used to estimate the cost for executing the sequence of instructions in *BB* and the function `basicBlockCost` returns the accumulated cost.

Each loop found in $scope_n$ ends the current basic block, *BB*. `llvm-mca` estimates its cost and the result is accumulated in *blockCost*. Then the loop is processed recursively (line 8) and the cost for executing one iteration of the loop's body is set in $loop_nCost$. Afterwards, $loop_nCost$ is multiplied by the loop trip count, adjusted to consider loop-level optimisations, to calculate the total loop cost, stored in *blockCost*. The AST node that succeeds the loop begins a new basic block, *BB*, that will end if further loops are found or when the last node of the current scope, $scope_n$, is reached.

To better illustrate our approach with a C/C++ example, consider the example in Listing 4.6, showing the defined basic blocks. The final cost for the parallel region, assuming no loop optimisations, is $BB\_1_c + (BB\_2_c \times M) + BB\_3_c + ((BB\_4 + BB\_5_c \times P) \times N)$, where each BB_i_c is the `llvm-mca` estimation to execute the basic block instruction sequence.

Concerning loops, innermost loops are annotated inserting the `llvm-mca` directives around

Algorithm 1 Using LLVM Machine Code Analyser recursively

```
1:  function BASICBLOCKCOST(scope_n)
2:      regionStart_n ← scope_n.first()
3:      regionEnd_n ← null
4:      blockCost ← 0

5:      for loop_n in scope_n.query("loop") do
6:          regionEnd_n ← loop_n.before()    ▷ Current basic block ends, between current starting
        node and the node before the nested loop
7:          blockCost ← blockCost + MCA(regionStart_n, regionEnd_n)

8:          loop_nCost ← basicBlockCost(loop_n.body())  ▷ Recursively process the nested loop's
        body, estimate cost of one iteration
9:          blockCost ← blockCost + loop_nCost × loop_n.tripCount()    ▷ Multiply estimation for
        one iteration by the loop's trip count

10:         regionStart_n ← loop_n.after()        ▷ Assume a new basic block after this nested loop
11:         regionEnd_n ← null
12:     end for

13:     regionEnd_n ← scope_n.last()
14:     blockCost ← blockCost + MCA(regionStart_n, regionEnd_n)

15:     return blockCost
16: end function
```

the loop to not compromise loop-level optimisations. For outer loops, our experiments show it works better to insert the directives in the loop body, especially in long loops. The assembly output does not necessarily respect the source code order. Thus, it often happens that the ending directive appears before the starting directive and `llvm-mca` fails.

```
// the parallel loop body under analysis
{
   // BasicBlock_1: START
   <...>
   // BasicBlock_1: END

   // BasicBlock_2: START
   for (i = 0; i < M; i++) {
      <...>
   }
   // BasicBlock_2: END

   // BasicBlock_3: START
   <...>
   // BasicBlock_3: END

   // BasicBlock_4: START
   for (i = 0; i < N; i++) {
      // BasicBlock_5: START
      for (j = 0; j < P; j++) {
         <...>
      }
      // BasicBlock_5: END
   }
   // BasicBlock_4: END
}
```

Listing 4.6: Basic Blocks demonstrantion in a workshare region with nested loops

To conclude, integrating the `llvm-mca` in a source-to-source approach is prone to fail, especially in more irregular and large code structures. Furthermore, due to loop versioning and optimisations, the estimations may be inaccurate. The lack of control flow simulation makes the tool challenging to use in an automated approached. Although not considered in our approach, in addition to loops, it is also necessary to analyse other structures more thoroughly, such as `if` statements. Finally, we found that inserting the inline assembly directives to delimit code regions for analysis may compromise the compiler outcome and it is an important limitation. Ideally, the annotations should be inserted after the assembly is generated to not avoid any interference. Improving the usage of `llvm-mca` for performance modelling is left for future work.

4.5 Summary

This chapter presented the analytical models used in this dissertation for estimating execution times in CPU targets. The analytical model used is based on Liao and Chapman's OpenMP model [51]. We simplify the model as we only address OpenMP parallel for constructs , and make the

necessary modifications to model x86 architectures. We use two approaches for estimating the computational of a sequence of instructions in a loop body.

The main approach is based on AST-level analysis to count the number of operations, map them to ISA instructions and use available reciprocal throughput values. Our AST-level approach does not consider data dependencies and other hazards, resulting in lower-bound estimations.

The second approach uses `llvm-mca`, first proposed by Chikin et al [30]. We identified the need to manually process loops within the region under analysis, as `llvm-mca` does not simulate control flow. `llvm-mca` is used to estimate the cost of one iteration, which is then multiplied by the trip count determined with AST analysis. We also demonstrated the need to account for loop-level optimisations, as the assembly sequence may correspond to more than one loop iteration. Given our source-to-source technique for inserting inline assembly that bounds sections of interest, the resulting assembly can differ from the original version as it introduces dependencies. Furthermore, in some cases, the ending directive appears before the starting directive causing `llvm-mca` to fail. Other issues are related with loop versioning, that may compromise the estimation accuracy. Integrating `llvm-mca` in a source-to-source analysis presents many challenges. We propose an approach that mitigates some of the issues but is still prone to fail. Using `llvm-mca` for accurate performance modelling is promising, but given our experience, it should be integrated into an optimised-IR analysis or assembly level. Exploring those options is left for future work.

Chapter 5

GPU Analytical Model

This chapter describes the GPU analytical model proposed by Kim et al. [47], which serves as the basis for estimating the execution time of GPU kernels in this work. Understanding Kim et al.'s analytical model is vital to identify limitations, some of which are addressed in Chapter 6. The last section in this chapter explains how to calculate the number of warps that execute concurrently in each Streaming Multiprocessor (SM), a parameter necessary for the analytical model. To best of own knowledge, no existing literature details how to calculate the occupancy, and there is no official documentation from NVIDIA either.

5.1 Overview

Kim et al. [47] proposed an analytical model for GPU architectures that introduces two key concepts: memory and computation warp parallelism.

Memory Warp Parallelism (MWP). Indicates how many memory requests can be served during one warp memory waiting period. In other words, consider the warp $warp_a$ that issues a memory request on time instant t_i and gets the data on at instant t_f. The MWP indicates how many warps, other than $warp_a$, can dispatch memory requests concurrently in the interval $[t_i, t_f]$ — not necessarily conclude them.

Computation Warp Parallelism (CWP). It represents how many warps can execute in a Streaming Multiprocessor during one memory waiting period, $[t_i, t_f]$. CWP is always greater than one because it counts $warp_a$, which completed some computation before starting the memory request.

The following tables summarise the input parameters and analytical model expressions. In order to make the subsequent sections easier to follow, application-related parameters are prefixed with app (short for application) and are listed in Table 5.1. Architecture or device-specific parameters are prefixed with *arch* (short for architecture) and are listed in Table 5.2. These parameters

are inputs for the analytical model. The tables also describe briefly how the input parameters can be collected — the implementation details are discussed in the next chapter in Sections 6.6 and 6.5. Table 5.3 summarises intermediate calculations in the analytical model and what they represent.

Application Parameters		
Name	Description	Source
`appMemInstsCoal`	Number of coalesced memory instructions per thread	Application code
`appMemInstsUncoal`	Number of uncoalesced memory instructions per thread	Application code
`appCompInsts`	Number of issued instructions per thread (includes memory operations)	Application code
`appThreadBlocks`	Number of thread blocks	User-defined or set by the compiler automatically
`appThreads-PerBlock`	Number of threads per thread block	User-defined or set by the compiler automatically
`appRegistersPer-Thread`	Number of registers needed per thread	Depends on how the compiler generates code
`appSharedMemPer-Block`	Number of shared memory bytes used per thread block	In CUDA, Shared Memory usage is explicit and depends on the application code. But, in context of OpenMP, compilers may use it automatically and is not controlled by the user. Thus, it depends solely on the compilers in the context of this work
`N`	Active warps per multiprocessor	CUDA Occupancy [1]
`ActiveBlocksPerSM`	Active thread blocks per CUDA	CUDA Occupancy [1]

Table 5.1: Input parameters of the GPU analytical model with respect to the application

5.2 Memory Warp Parallelism (MWP)

The Memory Warp Parallelism (MWP) indicates how many warps can concurrently access the memory during one warp memory period. It depends on the number of active warps per Streaming Multiprocessor (SM), global memory bandwidth, and memory access latencies. Those constraints are discussed individually in the remainder of this section.

Device parameters		
Name	Description	Source
`archSMCount`	Number of Streaming Multiprocessors (SM) on the GPU device	Device specifications
`archSMFreq`	SM frequency (GHz)	Device specifications
`archGlobalMem-Bandwidth`	The memory bandwidth between global/DRAM memory and SM's memory controllers (GB/s)	Device specifications
`archMem-TransactionSize`	The size of each memory transaction (bytes)	Compute Capability
`archMemTrans-Uncoal`	The number of memory transactions per uncoalesced memory instruction	Compute Capability
`archMemTransCoal`	The number of memory transactions per coalesced memory instruction	Compute Capability
`archMemDepart-DelayCoal`	The delay between consecutive coalesced memory transactions (cycles)	Micro-benchmarks
`archMemDepart-DelayUncoal`	The delay between consecutive uncoalesced memory transactions (cycles)	Micro-benchmarks
`archMemLatGlobal`	The global memory access latency (cycles)	Micro-benchmarks

Table 5.2: Parameters with respect to GPU architecture and device hardware

Model properties	
Name	Description
`MemLatUncoal`	The latency per uncoalesced memory instruction (cycles)
`MemLatCoal`	The latency per coalesced memory instruction (cycles)
`MemLat`	Average latency per memory instruction that depends on the number of coalesced and uncoalesced memory instructions in the application (cycles)
`MemDepartureDelay`	Average delay between consecutive memory transactions (cycles)
`CWP`	Number of warps that execute during one warp memory period, plus one
`MWP`	Number of warps that can access the memory concurrently during one memory period
`Mem_cycles`	Number of cycles in memory per warp
`Comp_cycles`	Number of cycles in computation operations per warp

Table 5.3: Analytical model equations summary

5.2.1 Bandwidth limitation

The first constraint is related to onboard physical limitations. Global memory or DRAM is off-chip memory and has maximum bandwidth, $archGlobalMemBandwidth$. It is shared among all SMs and connects to SM's memory controllers. Kim et al. analytical model [47] considers the global memory bandwidth is shared evenly with the SMs accessing memory at a given time. Thus, the higher the concurrent access, the less bandwidth is available per SM and respective executing warp. Besides, the SM clock frequency, $archSmFreq$, bounds the rate of moving data to on-chip memories (e.g., L2/L1 cache).

Assuming the data moves to the SM's memories at peak bandwidth, $BwPerWarp$ in Equation 5.1 is the maximum bandwidth possible for transferring one memory transaction to local memories. $MemLat$ is the average memory latency, i.e., round-trip time in cycles for sending the memory request to global memory and then transfer the data back to the SM. Memory requests span in multiple memory transactions, each of fixed size, $archMemTransactionSize$.

$$BwPerWarp = \frac{archSMFreq \times archMemTransactionSize}{MemLat} \tag{5.1}$$

Assuming that memory bandwidth is shared evenly among the SMs, the total necessary bandwidth to serve all memory requests at peak bandwidth is $BwPerWarp \times archSMCount$.

Finally, the MWP_{peakBw} in Equation 5.2 is the number of warps that can access the memory concurrently, per SM, when limited by the available memory bandwidth.

$$MWP_{peakBw} = \frac{archGlobalMemBandwidth}{BwPerWarp \times archSMCount} \tag{5.2}$$

5.2.2 Memory access latency

The second factor limiting the number of concurrent memory accesses in an SM during one memory waiting period, is the average memory request latency and the delay between consecutive memory requests.

The delays and round-trip latency depend on the type of memory access — coalesced or uncoalesced. Assuming all threads within a warp are executing on the same branch then all threads execute the same instruction, but with different operands — the thread's state. Therefore, when a warp executes a memory instruction there can be up to 32 requests to different memory addresses. Accesses to the memory hierarchy are done in transactions of size $archMemTransactionSize$ bytes. When threads access contiguous memory positions it may be possible to issue a single memory transaction that serves all the threads — coalesced memory access. However, if that is not possible, then multiple transactions are needed — uncoalesced memory access. A coalesced memory access results in a single memory transaction, while uncoalesced accesses issues 32 transactions (one per thread in the warp) [47].

Suppose different warps running in the same SM and doing memory accesses that can be coalesced. Then, each warp's memory access issues a single memory transaction. But, the transactions are not sent at the same instant in time. Instead, there is a departure delay between consecutive memory transactions from the different warps, $archMemDepartDelayCoal$. For uncoalesced accesses, the transactions resulting from the same memory instruction of an warp are also dispatched with some delay in between, $archMemDepartDelayUncoal$. The delays in both types of memory accesses are illustrated in Figure 5.1. The delays limit how many memory transactions can be issued during one memory waiting period.

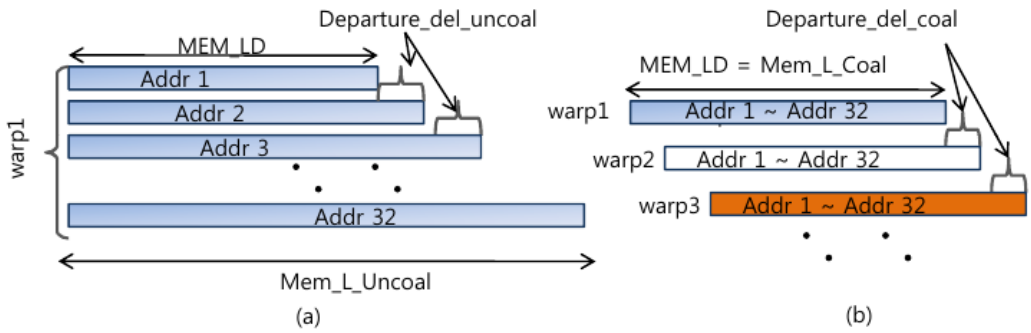


Figure 5.1: Delay between consecutive memory transactions in coalesced and uncoalesced memory accesses [47].

An application might have different combinations of both kinds of memory accesses — coalesced and uncoalesced. Therefore, Kim et al. [47] proposed a weighted average for memory request latencies, $MemLat$ in Equation 5.3, and departure delays, $MemDepartureDelay$ in Equation 5.4.

$$\begin{aligned} MemLat = {} & MemLatUncoal \times WeightUncoal \\ & + MemLatCoal \times WeightCoal \end{aligned} \tag{5.3}$$

$$\begin{aligned} MemDepartureDelay = {} & archMemDepartDelayCoal \times WeightCoal \\ & + (archMemDepartDelayUncoal \times archMemTransUncoal) \\ & \times WeightUncoal \end{aligned} \tag{5.4}$$

The weight factors, $WeightCoal$ and $WeightUncoal$, and latency for each kind of memory request, $MemLatCoal$ and $MemLatUncoal$, are determined by Equation 5.5. Note that $archMemLatGlobal$ is an architecture parameter and is the round-trip latency for a single memory transaction of fixed size (typically 128 bytes). As coalesced memory accesses issue a single memory transaction, then $MemLatCoal = archMemLatGlobal$. For uncoalesced accesses, it is necessary

to add the multiple transactions and the delays in-between.

$$\begin{aligned}
appMemInstsTotal &= appMemInstsCoal + appMemInstsUncoal \\
WeightCoal &= \frac{appMemInstsCoal}{appMemInstsTotal} \\
WeightUncoal &= \frac{appMemInstsUncoal}{appMemInstsTotal} \\
MemLatCoal &= archMemLatGlobal \\
MemLatUncoal &= archMemLatGlobal \\
&\quad + (archMemTransUncoal - 1) \\
&\quad \times archMemDepartDelayUncoal
\end{aligned} \tag{5.5}$$

Finally, Equation 5.6 shows how to determine $MWP_{withoutPeakBw}$, which is the number of memory requests per SM that can be issued during one memory waiting period, assuming the latencies and delays are the limiting factors.

$$MWP_{withoutPeakBw} = \frac{MemLat}{MemDepartureDelay} \tag{5.6}$$

5.2.3 Putting it all together

The final and simplest constraint is that memory warp parallelism cannot exceed N, the number of active warps in one SM — number of warps concurrently executing in an SM. Recall that MWP is the number of warps that can concurrently access the memory during one memory period. Hence, $MWP \leq N$.

Lastly, MWP is the minimum value of all presented constraints, as show in Equation 5.7. MWP_{peakBw} and $MWP_{withoutPeakBw}$ estimate the value for MWP isolating specific limiting factors. Hence, MWP is the minimum of all. For instance, if the minimum value is MWP_{peakBw} then MWP is constrained by the bandwidth available. On the other hand, if the limiting factor is N, then there are no sufficient warps to exploit the available memory level parallelism.

$$MWP = MIN(N, MWP_{peakBw}, MWP_{withoutPeakBw}) \tag{5.7}$$

5.3 Computation Warp Parallelism (CWP)

Computation Warp Parallelism (CWP) is the number of warps per SM that can perform computation during one memory waiting period. CWP is the ratio of cycles spent in memory instructions and in computation instructions, as shown in Equation 5.10. The higher the CWP, the less computation per memory instruction.

The consumed cycles in memory instructions, in Equation 5.8, depends on the coalesced or uncoalesced memory request latencies and the number of issued operations.

$$\begin{aligned} Mem\_cycles = MemLatUncoal \times appMemInstsUncoal \\ + MemLatCoal \times appMemInstsCoal \end{aligned} \tag{5.8}$$

The computation cycles, in Equation 5.9, is a function of the total number of instructions and respective cost in cycles. Kim et al. [47] use a constant cost of 4 cycles for all instructions.

$$Comp\_cycles = Issue\_cycles \times appCompInsts \tag{5.9}$$

$$CWP = min(N, \frac{Mem\_cycles}{Comp\_cycles} + 1) \tag{5.10}$$

Such as MWP, CWP is at least the number of active warps per SM, N. If that is the case, it indicates a lack of warps to increase GPU utilization.

5.4 Application Execution Cycles

The output of the analytical model is an estimation of the number of cycles to execute the application. A GPU has multiple SMs. Assuming all SMs cooperate to execute the same kernel and work is evenly distributed, then each SM requires the same amount of time for computing (approximately). Thereby, the goal is to estimate how much time an SM needs to complete its job, which represents the application execution time.

The previous Sections, 5.2 and 5.3, introduced the MWP and CWP metrics. Depending on the relation between the two metrics, the application may be memory or computation bound. Besides, some memory or computation operations overlap each other and do not contribute to the total execution time.

The following sections address the different relations between MWP and CWP and how to calculate the cycles spent in SMs accordingly.

5.4.1 CWP greater than MWP

The first scenario under consideration is when $CWP > MWP$. In this case, while one warp waits for its memory request to complete, there are enough warps available for computing, thus better using the SM resources. In these circumstances, the application is memory bound. Therefore, the memory operations hide computation cost and contribute the most to the application execution time.

An example is illustrated in Figure 5.2. Green boxes represent computation periods, while orange boxes represent memory periods. The darker colours are the periods that effectively contribute for application execution time. The lighter colour boxes happen concurrently with other operations, and thus their cost is hidden. The vertical axis represents different warps running,

while the horizontal axis displays how the execution flows over time. The boxes are numbered to identify the warps.

In Figure 5.2 example, there are 8 active warps. Up to 2 warps can access the memory concurrently, hence $MWP = 2$. For instance, while the first warp accesses the memory, the second warp can also access it. While the $warp_1$ is on its memory period, two other warps can complete their computation period, thus $CWP = 3$ (counting the $warp_1$ itself).

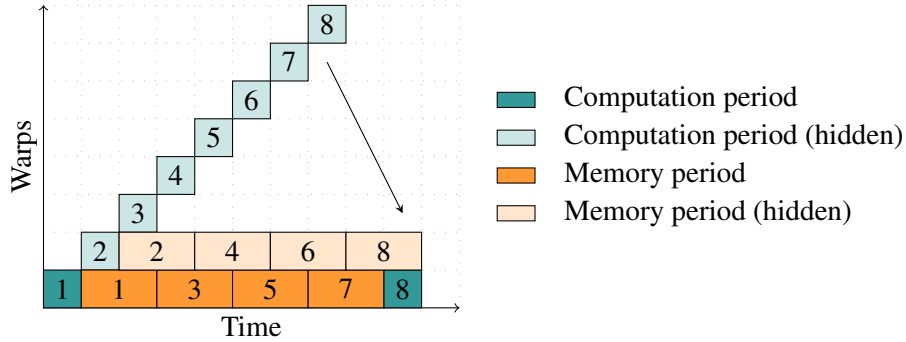


Figure 5.2: An application execution time illustration when warps are memory bounded

Figure 5.2 shows that memory operations contribute the most to the execution time. As some memory accesses are overlapped to a MWP degree, the exact number of memory requests that contribute to the application execution time is $\frac{N}{MWP}$, with N being the number of active warps. Multiplying by the number of cycles each warp spends in memory periods, Mem_cycles, gives the elapsed cycles on memory operations, as shown in Equation 5.11.

$$Exec\_cycles = Mem\_cycles \times \frac{N}{MWP} + \frac{Comp\_cycles}{appMemInstsTotal} \times MWP \quad , \text{ if } CWP \geq MWP \tag{5.11}$$

Some computation periods also contribute to the execution time and depend on the MWP. In Figure 5.2, two computation periods are added. One period is due to $warp_1$'s computation period. The second is because when $warp_7$ finishes its memory period, $warp_8$ is still in progress for an amount of time that equals one computation period. Another way to think about it is re-arranging the periods such that the first memory periods instantiate at the same time and in parallel. For instance, consider $MWP = 3$. Then, only after $warp_3$ completes its computation period, memory requests would begin. At that instant, 3 computation periods were completed which matches the MWP. Thus, the number of computation cycles that contributes to the application execution time is the cycles per computation period, $\frac{Comp\_cycles}{appMemInstsTotal}$, multiplied by MWP.

In Figure 5.2, the length of memory and computation periods is the same. In reality, the length of the periods depends on the distribution of memory instructions on the application code and the cost of computation in between. For simplification, Kim et al. [47] assume all computation periods are roughly the same, taking the total estimated computation cycles and dividing by the number of memory instructions, $\frac{Comp\_cycles}{appMemInstsTotal}$.

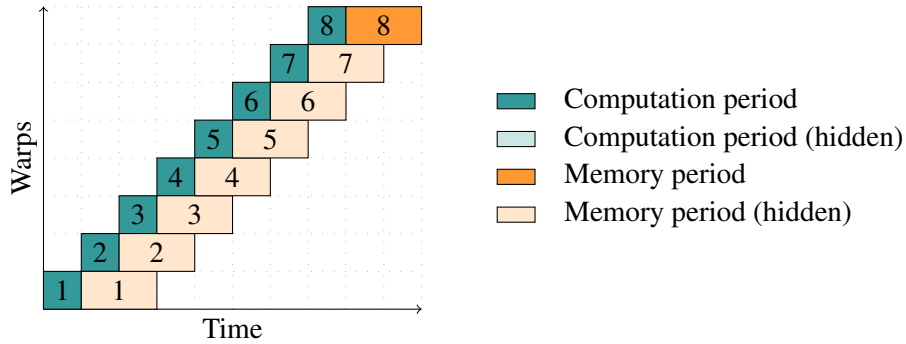


Figure 5.3: An application execution time illustration when warps are computation bounded

5.4.2 MWP greater than CWP

The second scenario is when $MWP \geq CWP$. In contrast to the previous case, computation periods overlap and hide memory periods. In other words, the application is computation bound.

A scenario where $CWP = 3$ and $MWP = 8$ is illustrated in Figure 5.3. Computation periods are the main aspect for total execution time. In fact, all computation periods contribute to the application execution. As shown in Equation 5.12, the computational cycles per warp, $Comp\_cycles$, is multiplied by the number of active warps, N. As in most applications, the execution flow finishes with a memory operation to store the computation result, hence the analytical model adds one memory period.

$$Exec\_cycles = MemLat + Comp\_cycles \times N, \text{ if } MWP > CWP \tag{5.12}$$

5.4.3 Lack of warps

The last possible scenario is when there is lack of warps to explore the available parallelism. This scenario happens when $MWP = CWP = N$. Since memory operations do not overlap computation operations and vice-versa, both contribute to the total execution time, as illustrated in Figure 5.4.

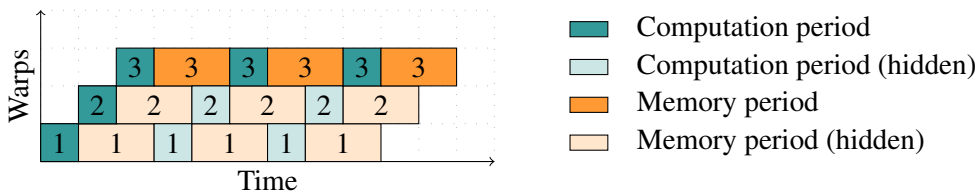


Figure 5.4: An application execution time illustration when it does not explore the available parallelism

However, warps still execute in a time sharing fashion and memory and computation periods overlap each other. As shown in Figure 5.4, all memory and computation periods of one warp contribute to the execution time, e.g. $warp_3$. Thus, $Comp\_cycles$ and Mem_cycles, which are the total cost in computational and memory operations per warp, are counted once (Equation 5.13). Moreover, warps begin executing on different time instants, separated by one computation period,

$\frac{Comp\_cycles}{appMemInstsTotal}$. Therefore, it is necessary to add $N-1$ computation periods to the execution time. In the example, the first computation period of $warp_1$ and $warp_2$ are added.

$$Exec\_cycles = Mem\_cycles + Comp\_cycles + \frac{Comp\_cycles}{appMemInstsTotal} \times (N-1) \quad , \text{ if } MWP = CWP = N \tag{5.13}$$

5.4.4 Thread blocks and warps scheduling

When a kernel launches on the GPU, it is organized in independent thread blocks and a thread block is arranged in various warps (typically, groups of 32 threads). The number of thread blocks and the size of each can be specified by the programmer, or it is determined automatically by the compiler (e.g. in OpenMP offloading).

When the kernel launches, the thread blocks are queued and then scheduled to the available SMs. The SM partitions each thread block in warps to be assigned to warp schedulers. A thread block, and its warps, are scheduled together to the same SM. Although it is not possible to have warps of the same thread block scheduled to different SMs, it is possible to have different thread blocks scheduled to the same SM. When an SM finishes a thread block, it can fetch more blocks from a pool of waiting thread blocks.

The number of thread blocks that are concurrently executing in SM, $ActiveBlocksPerSM$, is limited by the resource usage of each thread, e.g. registers and shared memory, as well as the size of each thread block. The number of active warps per SM, N, is correlated with $ActiveBlocksPerSM$, and is used several times in the analytical model equations. As stated before, the programmer only specifies the number of thread blocks and threads per block. Determining the number of active warps and active thread blocks per SM is addressed in the next Section 5.5.

For example, consider the Pascal GPU architecture and Compute Capability 6.0 [6]. The SMs for that architecture have 65536 32-bit registers. Assume an application where each thread needs 150 registers and the thread block size is 256 threads (8 warps). The number of registers needed per thread block is $256 \times 150 = 38400$ registers, consuming more than half of the register file size. Consequently, it is not possible to have two thread blocks concurrently executing in the SM. However, if the register pressure or the size of each thread block are reduced, it might be possible to have more thread blocks and increase occupancy.

The execution cost, $Exec\_cycles$, represents the cost of executing a single batch of N warps in an SM (either from the same thread block or from multiple thread blocks). As it is likely that the number of thread blocks launched by the application is such that is not possible to scheduled all of them immediately, some thread blocks remain in the queue waiting to be scheduled. Therefore, each SM may execute batches of N warps multiple times.

Equation 5.14 shows how to determine the number of times an SM repeats the execution of N warps, in function of thread blocks launched and how many blocks can be active per SM.

$$RepsPerSM = \frac{appBlocks}{ActiveBlocksPerSM \times archSMCount} \tag{5.14}$$

The final execution cycles is multiplying the cost for computing a batch of N warps by $RepsPerSM$, as shown in Equation 5.15.

$$Exec\_cycles = Exec\_cycles \times RepsPerSM \tag{5.15}$$

5.5 Calculating Occupancy

This section discusses the calculation of the number of active thread blocks per Streaming Multiprocessor (SM) and the number of warps running concurrently. To the best of our knowledge, NVIDIA does not document it and there is not literature about it. However, NVIDIA provides a spreadsheet where users can input their application's resource usage [1]. The spreadsheet calculates the achieved occupancy and how it varies as a function of different resource usage, helping users fine-tune the grid geometry.

As part of this work, it is necessary to automatically calculate the number of active thread blocks for the analytical model. Consequently, it was necessary to inspect the spreadsheet, extract the formulas used internally, and understand their logic (which unveils more details concerning the architecture). The calculation steps are described in the remainder of this section.

The number of active thread blocks per SM is constrained due to the following three reasons:

Warps/Blocks limit. A hardware limit on the number of warps or blocks can simultaneously reside per SM. The limit presumably exists because it is necessary to keep all threads/warps' context and resource allocation is fixed at the hardware level.

Registers. Threads consume registers to store intermediate computation results. Each SM has a fixed register file size, i.e. number of 32-bit registers available. The more registers are needed per thread, the fewer threads can be active per SM. In a bottom-up logic, if there are fewer concurrent threads, the number of active warps and thread blocks is also smaller. Besides, there might be a hardware limit on how many registers can be allocated per thread block.

Shared Memory. It is on-chip memory accessible by all threads in a thread block, and it is allocated per thread block. Similarly to the registers, there is a limit on how much shared memory can be allocated per thread block.

Given the enumerated constraints, the approach to calculate occupancy presented by NVIDIA in occupancy calculator [1] is finding the maximum number of thread blocks assuming individual constraints on: *(a)* warps, *(b)* blocks, *(c)* registers, and *(d)* shared memory. The effective number of active thread blocks per SM is the minimum obtained value.

The following sections discuss each constraint individually. Parameters follow the same nomenclature defined earlier — architectural parameters are prefixed with `arch` and application or user input values with `app`. Moreover, the term *block* in the text and equations refer to a *thread block*. Table 5.4 summarises the necessary GPU parameters to calculate occupancy.

Compute Capability parameters	
Name	Description
`archThreadsPerWarp`	Number of threads that compose a warp
`archMaxWarpsPerSM`	Limit of simultaneously running warps per SM
`archWarpAllocUnitSize`	The granularity for allocating warps per SM, i.e. the number of warps is always multiple of this unit size
`archMaxThreadsPerSM`	Limit of active threads per SM
`archMaxBlocksPerSM`	Limit of active thread blocks per SM
`archMaxBlockSize`	Limit of threads per thread blocks
`archSharedMemPerSM`	Number of bytes of shared memory available per SM
`archSharedMemPerBlock`	Number of bytes that can be allocated per thread block
`archSharedMemAllocUnitSize`	The granularity for allocating shared memory per thread block
`archRegsPerSM`	Number of 32-bit registers available per SM
`archMaxRegsPerBlock`	Maximum number of registers that can be allocated per thread block
`archMaxRegsPerThread`	Maximum number of registers that can be used per thread
`archRegAllocUnitSize`	The granularity for allocating registers per thread

Table 5.4: GPU architecture and device-specific parameters needed to calculate occupancy [1]

5.5.1 Limit on Thread Blocks and Warps

NVIDIA GPUs have a physical limitation for active warps and active thread blocks per Streaming Multiprocessor (SM). As the goal is to calculate how many thread blocks can concurrently execute in the SM, the second constraint is straightforward, as shown in Equation 5.16.

$$BlocksPerSM\_blockBound = archMaxBlocksPerSM \tag{5.16}$$

Concerning the constraint on concurrent warps per SM, it is necessary to calculate how many warps per thread block are launched, $WarpsPerBlock$. Given the number of thread blocks used to launch the kernel (user configured) and the size of each warp (typically 32 threads), Equation 5.17 illustrates how to calculate it.

Each thread block is arranged in $WarpsPerBlock$ warps. Moreover, each SM has a limit of maximum active warps per SM, $archMaxWarpsPerSM$. Dividing both values gives the maximum

number of thread blocks in an SM when bounded by the number of active warps, *BlocksPerSM-_warpBound*, as shown in Equation 5.17.

$$\begin{aligned} WarpsPerBlock &= \left\lceil \frac{appThreadsPerBlock}{archThreadsPerWarp} \right\rceil \\ BlocksPerSM\_warpBound &= \left\lfloor \frac{archMaxWarpsPerSM}{WarpsPerBlock} \right\rfloor \end{aligned} \tag{5.17}$$

5.5.2 Register bounded

On NVIDIA GPUs, each Streaming Multiprocessor (SM) has a fixed amount of 32-bit registers, *archRegsPerSM*. Besides, there is a limit on the number of registers that can be used per thread block, *archMaxRegsPerBlock*. In general, both parameters are equal, meaning a thread block can use the entire register file size. However, in some Compute Capability 5.x and 3.x architectures *archMaxRegsPerBlock* is half of *archRegsPerSM*. Therefore, a thread block can only access half of the register file size. It is not clear why that is the case, but possibly the bank of registers are split at the hardware level.

Overtime, register allocation has changed. In Compute Capability 1.x architectures, registers are allocated per block, but on succeeding architectures they are allocated per warp. The difference changes how to calculate the number of thread blocks per SM when registers are the limiting factor. As Compute Capability 1.x is deprecated since CUDA 7.0, released in 2015 [1], only per-warp allocation is addressed.

Registers are allocated to warps in chunks of size *archRegAllocUnitSize* — since Compute Capability 3.0 the chunk size is 256 registers. The first step is to determine how many registers a warp needs, *RegistersPerWarp*. Given the thread-centred programming model, the number of required registers is defined on a per-thread basis, *appRegsPerThread*. In turn, a warp is a fixed sized group of *archThreadsPerWarp* threads. Equation 5.18 shows the product to compute the total amount of registers needed per warp. As registers are allocated in chunks, $\left\lceil \frac{appRegsPerThread \times archThreadsPerWarp}{archRegAllocUnitSize} \right\rceil$ is the minimum number of chunks needed, each with *archRegAllocUnitSize* registers. Multiplying the number of chunks by *archRegAllocUnitSize* gives the final number of registers allocated per warp.

$$\begin{aligned} RegistersPerWarp = &\left\lceil \frac{appRegsPerThread \times archThreadsPerWarp}{archRegAllocUnitSize} \right\rceil \\ &\times archRegAllocUnitSize \end{aligned} \tag{5.18}$$

The next step is to determine how many warps can be active per SM, assuming that register usage is the limiting factor, $WarpsPerSM_{regBound}$. Once again, it is not as simple as dividing the register file size by *RegistersPerWarp* because warps are also allocated with some granularity, *archWarpAllocUnitSize*. NVIDIA documentation lacks an explanation, but it is possible to observe that granularity (typically 2 or 4) matches the number of warp scheduler units in the SM.

[1] http://developer.download.nvidia.com/compute/cuda/7_0/Prod/doc/CUDA_Toolkit_Release_Notes.pdf

Presumably, registers at the hardware level are statically partitioned and banks of registers are exclusive to each scheduler unit. Assuming an SM with four warp scheduler units and only two active warp schedulers working, warps cannot benefit from unused registers by the idle warp schedulers.

Equation 5.19 shows how the maximum number of warps per SM is determined, taking into account the warp scheduling granularity. Note that the number of registers available is *archMaxRegsPerBlock* instead of the entire register file size, *archRegsPerSM*. As explained in the beginning of this section, in some architectures each thread block can only use a part of all registers in the SM. Since warps are associated to a thread block, inherently the warps can only use *archMaxRegsPerBlock* registers.

$$WarpsPerSM_{regBound} = \left\lfloor \frac{archMaxRegsPerBlock}{RegistersPerWarp \times archWarpAllocUnitSize} \right\rfloor \times archWarpAllocUnitSize \tag{5.19}$$

Finally, the maximum number of thread blocks per SM when constrained by registers can de determined dividing $WarpsPerSM_{regBound}$ by $WarpsPerBlock$. If the ratio is < 1, there are not enough registers for all the warps of one thread block. Equation 5.20 also multiplies the result by $\frac{archRegsPerSM}{archMaxRegsPerBlock}$. Note that $WarpsPerSM_{regBound}$ is calculated using the maximum number of registers available per thread block, *archMaxRegsPerBlock*. If *archMaxRegsPerBlock* is half of the total SM's register file size, *archRegsPerSM*, then the number of active threads can double.

Equation 5.20 includes three options. $WarpsPerSM_{regBound}$ was explained assuming the registers per thread do not exceed the architectural limit $archMaxRegsPerThread$. If the limit is exceeded, the kernel cannot be launched and effectively no blocks can be scheduled. If the threads do not use registers at all, then the number of blocks is only limited by *archMaxBlocksPerSM*.

$$BlocksPerSM\_regBound = \begin{cases} 0 & \text{, if } appRegsPerThread > archMaxRegsPerThread \\ \left\lfloor \frac{WarpsPerSM_{regBound}}{WarpsPerBlock} \right\rfloor \times \left\lfloor \frac{archRegsPerSM}{archMaxRegsPerBlock} \right\rfloor & \text{, if } 0 < appRegsPerThread < archMaxRegsPerThread \\ archMaxBlocksPerSM & \text{, if } appRegsPerThread = 0 \end{cases} \tag{5.20}$$

5.5.3 Shared Memory

Shared memory is on-chip memory available to all threads in a thread block. Each SM has *archSharedMemPerBlock* bytes of shared memory. In contrast to registers, shared memory is allocated on a thread block basis. The amount of bytes consumed by each thread block is application-dependent ($appSharedMemPerThreadBlock$).

Although the amount of shared memory used by a thread block is an application input parameter, there are two factors to consider that impact the actual amount of memory used. First, like register allocation, shared memory is assigned in chunks per thread block. The size of each

chunk is *archSharedMemAllocUnitSize* bytes. The second factor is that starting with Compute Capability 8.x, the CUDA runtime driver takes around 1KB of shared memory per thread block. The occupancy calculator [1] lists runtime driver's shared memory usage per driver version. The amount of memory taken depends on Compute Capability and CUDA runtime driver version. However, for the time being, all driver versions use 1KB and it is enough to check the architecture version — *archCudaSharedMemPerBlock*. Considering both factors, Equation 5.21 shows how to determine the number of shared memory bytes taken per thread block.

$$cudaSharedMemPerThreadBlock = \begin{cases} 0, & \text{if Compute Capability} < 8 \\ 1024, & \text{if Compute Capability} \geq 8 \end{cases}$$
$$SharedMemPerThreadBlock = \left\lceil \frac{appSharedMemPerThreadBlock + archCudaSharedMemPerBlock}{archSharedMemAllocUnitSize} \right\rceil \times archSharedMemAllocUnitSize \tag{5.21}$$

Finally, Equation 5.22 shows how to determine how many thread blocks can be active per SM assuming there is a constraint in shared memory, *ThreadBlocksPerSM_sharedMemBound*. Similarly to registers, besides the shared memory limit per SM, there is a limit per thread block and the approach is similar. If the thread block needs more than *archSharedMemPerBlock*, it is invalid. Otherwise, the only limiting factor is *archSharedMemPerSM*.

$$ThreadBlocksPerSM\_sharedMemBound = \begin{cases} 0, & \text{if } appSharedMemPerBlock > archSharedMemPerBlock \\ \left\lfloor \frac{archSharedMemPerSM}{SharedMemPerThreadBlock} \right\rfloor, & \text{if } 0 < appSharedMemPerBlock < archSharedMemPerBlock \\ archMaxBlocksPerSM, & \text{if } appSharedMemPerBlock = 0 \end{cases} \tag{5.22}$$

There is one additional detail in in Compute Capability 2.0-3.0 architectures as the L1 cache and Shared Memory reside on the same memory chip. In some architectures, the partitions are fixed, but others allow configuring the memory layout, i.e. how much memory is used as L1 cache and shared memory. There are pre-configured layouts. Hence, in Equation 5.22, *archSharedMemPerSM* is not necessarily the total memory size, but a pre-configured value accordingly to the selected memory layout. Nonetheless, the default is the maximum possible size.

5.5.4 Final remarks

The previous sections covered how different architectural properties limit the number of active thread blocks per SM. For each scenario, the number of possible thread blocks is calculated. The

actual number of active thread blocks is the minimum value obtained for each constraint, as shown in Equation 5.23.

$$\begin{aligned} BlocksPerSM = MIN(& \\ & BlocksPerSM\_warpBound, \\ & BlocksPerSM\_threadBlockBound, \\ & BlocksPerSM\_regBound, \\ & BlocksPerSM\_sharedMemBound \\ &) \end{aligned} \tag{5.23}$$

The number of active warps per SM is straightforward: the product of the number of warps per thread block by the number of active thread blocks per SM (Equation 5.24).

$$N = BlocksPerSM \times WarpsPerBlock \tag{5.24}$$

5.6 Summary

This chapter presented the analytical model proposed by Kim et al. [47], which is an essential basis for the analytical model used in this work. The analytical model was designed for the first CUDA-enabled GPUs from NVIDIA. Since then, architectures have evolved, and the analytical model needs refinements: Chapter 6 addresses some of its limitations. However, to identify the limitations in the original analytical model and address them, it is essential to described Kim et al.'s approach, which was the intent of this chapter.

Furthermore, this chapter presented how to calculate the number of thread blocks concurrently executed in each SM. Then, the number of active warps is derived, which is used several times in the analytical model [47].

Chapter 6

Improving the GPU Analytical Model

Since Kim et al. proposed the GPU analytical model [47], GPU architectures have evolved substantially. This chapter proposes some adjustments to the analytical model in order to reflect modern GPUs, addressing limitations in instruction cost estimations, number of memory transactions and warp-level parallelism. Finally, it presents how to collect input parameters for the model in a context where applications are offloaded with OpenMP, which reveals specific issues.

6.1 The OpenMP factor

The original analytical model counts the number of instructions and memory operations per thread, which is the natural approach given the SIMT CUDA programming model. In OpenMP offloading context, counting operations has to consider the *workshare region* concept and how *work items* are distributed among the threads in the GPU.

```
#pragma omp parallel for collapse(2)
for(int i = 0; i < NI; i++) {
   for(int j = 0; j < NJ; j++) {
      // begin of worksharing region
      ...
      // end of worksharing region
   }
}
```

Listing 6.1: A parallel OpenMP loop nest and the illustration of the worksharing region

The workshare region is the block of code that is executed in parallel by all participating threads. In Listing 6.1, the two outermost loops, `i` and `j`, are marked as parallel and are coalesced into a single iteration space. The body of the loop `j` is the workshare region. Thus, threads will execute the code of the loop body for one or more `(i, j)` pairs, each corresponding to a work item. The distribution of work items per threads depends on the total number of work items ($NI \times NJ$ in the example), and number of launched threads.

In the GPU context, the total number of threads depends on grid geometry used to launch the kernel function, which defines the number of thread blocks and their size (number of threads per thread block). Equation 6.1 shows the total number of launched threads.

$$NumTotalThreads = appThreadBlocks \times appThreadsPerBlock \tag{6.1}$$

Assuming the number of parallel iterations is higher than the number of launched threads, the compiler handles the OpenMP workshare region assigning multiple work items per thread. In practice, the assembly code in the SIMT model has two or more loop iterations. The number of work items is shown in Equation 6.2. As $NumTotalThreads$ is not necessarily multiple of $WorkItemsPerThread$, some threads may have fewer work items than others. In general, the thread's code may have a conditional statement to ensure it does not go beyond the problem size, which could cause issues such as writing in invalid memory addresses. However, that should affect just the last running warps, and for simplification, the issue is not handled in our work.

$$WorkItemsPerThread = \left\lceil \frac{NumParallelIterations}{NumTotalThreads} \right\rceil \tag{6.2}$$

The operations counted at the AST level represent the computation per work item. The total number of operations per thread is the cost of one work item times the number of work items assigned to each thread, as exemplified in Equation 6.3.

$$\begin{aligned} appCompInsts &= appCompInsts \times WorkItemsPerThread \\ appMemInstsCoal &= appMemInstsCoal \times WorkItemsPerThread \\ appMemInstsUncoal &= appMemInstsUncoal \times WorkItemsPerThread \end{aligned} \tag{6.3}$$

Chikin et al. [30] also identified the need to multiply operations counted at IR-level by an OpenMP factor, OMP_Rep. However, instead of multiplying the number of operations, they multiply the $Exec\_cycles$, the cost for executing N warps per Streaming Multiprocessor (SM), as shown in Equation 6.4. Although a more straightforward approach, it suffers of an incorrection.

$$\begin{aligned} Exec\_cycles = (Mem\_cycles \times \frac{N}{MWP} + \frac{Comp\_cycles}{appMemInstsTotal} \times MWP) \\ \times RepsPerSM \times \textbf{OMP\_Rep} \end{aligned} \tag{6.4}$$

For instance, consider $WorkItemsPerThread = 2$, i.e., each thread launched in the GPU computes two iterations of the workshare region. Therefore, the number of computation ($appCompInsts$) and memory operations ($appMemInstsTotal$) doubles. $Comp\_cycles$ is calculated as $Issue\_cycles \times appCompInsts$ (see Equation 5.9), which also doubles due to the increase of computation operations. However, the ratio $\frac{Comp\_cycles}{appMemInstsTotal}$ in Equation 6.4 is the same, as both numerator and denominator increase proportionally. Using the equation presented by Chikin et al. induces an incorrection. Therefore we argue that the correct approach, given the original analytical model, is to adjust the number of operations, as presented in Equation 6.3. In our approach, the remainder of the analytical model does not require any further adjustment.

6.2 Warp-level parallelism

Each Streaming Multiprocessor (SM) in recent NVIDIA GPUs is organized in partitions composed of warp schedulers, as shown in Figure 6.1. Each warp scheduler unit is responsible for executing warps, and therefore there is warp parallelism in the SM. The original analytical model [47] was outlined for the Tesla architecture (CC 1.x), where each SM has a single warp scheduler unit. Consequently, extending the model to account for the multiple schedulers available per SM in modern architectures is necessary.

SMs are hardware building blocks that combine processing and memory units. Each SM partition groups registers files, one warp scheduler unit and the execution units available for the scheduler. Other units such as L1 caches, Shared Memory and Texture Units are private to the SM, but shared by all the partitions. The remainder of this chapter refers to these partitions as *execution blocks* because a partition aggregates the actual units for computing warps — execution, scheduler and dispatch units — and therefore, it is a block responsible for executing warps.

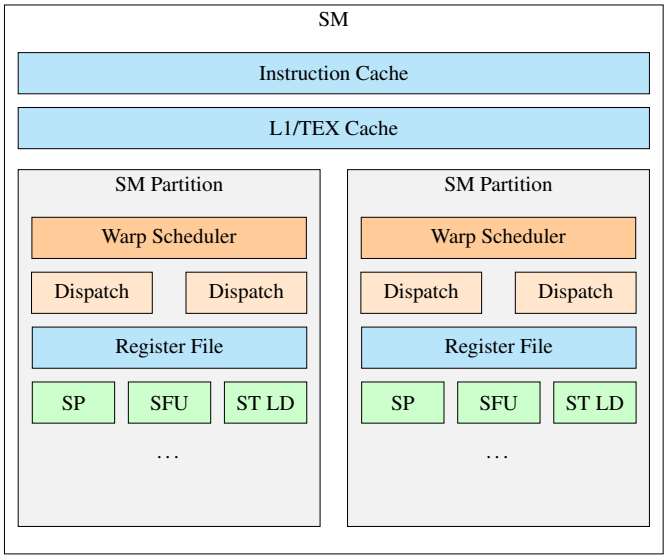


Figure 6.1: Partitions in the Streaming Multiprocessors

One possible approach to extend the Kim et al. analytical model [47], presented in Chapter 5, is related to the following question: can the SM notion used in the analytical model be translated to an execution block? Since the analytical model only addresses the computing units and global memory (which is off-chip memory), the notion of SM can be compared to an execution block. However, there is one exception: in Kepler's architecture, some execution units are shared between warp scheduler units [5]. To model Kepler, it would be necessary to discover the scheduling logic and how instructions are dispatched to the shared units. To the best of our knowledge, it is undocumented and would require a thorough analysis. Besides, the architecture is old and is entering the deprecated status as CUDA drivers no longer support Kepler devices. Architectures released afterwards have well-defined partitions, reducing the motivation to address this problem in our work.

Considering the presented hypothesis, several metrics are adjusted to become relative to an execution block rather than an SM. For instance, the number of active warps per SM, N, is transformed into active warps allocated per partition, $ActiveWarpsPerExecBlock$. Note that warps in SMs are evenly assigned and pinned to each partition before starting executing [12]. Therefore, the number of warps per execution block is determined with a straightforward division, as shown in Equation 6.5.

$$ActiveWarpsPerExecBlock = \left\lceil \frac{N}{archSchedulersPerSM} \right\rceil \tag{6.5}$$

The number of times each SM computes a group of N warps, $RepsPerSM$, is transformed in a equivalent metric concerning the number of repetitions per execution block — $ExecBlocksCount$ in Equation 6.6.

$$ExecBlocksCount = archSMCount \times archSchedulersPerSM \tag{6.6}$$

Metrics MWP and CWP become relative to the execution blocks as well, therefore they need to be updated so that neither is higher than the number of active warps per execution block. MWP_peakBw is the number of memory requests issued in one memory waiting period. The original analytical model [47] calculates it assuming global's memory bandwidth is evenly shared by all SMs. The assumption is extended to the execution blocks, as shown in Equation 6.7.

$$\begin{aligned}
BwPerWarp &= \frac{archSMFreq \times archMemTransactionSize}{MemLat} \\
MWP_{peakBw} &= \frac{archGlobalMemBandwidth}{BwPerWarp \times ExecBlocksCount} \\
MWP &= MIN(ActiveWarpsPerExecBlock, MWP_{peakBw}, MWP_{withoutPeakBw}) \\
CWP &= MIN(ActiveWarpsPerExecBlock, \frac{Mem_cycles}{Comp_cycles} + 1)
\end{aligned} \tag{6.7}$$

The last adjustment required is calculating the cycles-cost, $Exec\_cycles$, for executing a group of $ExecBlocksCount$ execution blocks. As explained in Section 5.4, in the original analytical model, an SM may execute a group of warps multiple times, and it was necessary to introduce a factor $RepsPerSM$. Similarly, an execution block may execute a group of warps multiple times. Equation 6.8 shows how to determine the repetition factor. Finally, Equation 6.9 determines the cost for executing all warps scheduled into an execution block.

$$RepsPerExecBlock = \frac{RepsPerSM}{archSchedulersPerSM} \tag{6.8}$$

$$Exec\_cycles = RepsPerExecBlock \times \begin{cases} Mem\_cycles \times \frac{ActiveWarpsPerExecBlock}{MWP} + \frac{Comp\_cycles}{appMemInstsTotal} \times MWP & \text{if } CWP \geq MWP, \\ MemLat + Comp\_cycles \times ActiveWarpsPerExecBlock & \text{if } MWP > CWP, \\ Mem\_cycles + Comp\_cycles + \frac{Comp\_cycles}{appMemInstsTotal} \times (N-1) & \text{if } MWP = CWP = ActiveWarpsPerExecBlock \end{cases} \tag{6.9}$$

6.3 Demystifying the coalesced and uncoalesced classification

Kim et al. [47] classifies memory accesses within the warp as coalesced or uncoalesced. Essentially, if all threads of the warp access contiguous memory addresses, then the requested data is coalesced and all threads are served with a single 128 B memory transaction. When threads access scattered memory positions, the GPU issues one memory transaction per thread, resulting in 32 transactions — the warp size — of 128 B. Uncoalesced memory requests are inefficient because more transactions are issued, reducing overall available bandwidth. Moreover, most of the data delivered by each transaction is unused because typical word sizes ranges from 1 B to 16 B. The original analytical model [47] was designed for the first architectures of Compute Capability 1.x, where the binary classification for memory instructions, — as coalesced or uncoalesced —, is correct and is documented by NVIDIA. However, the following architectures changed how global memory requests work, relaxing some constraints and enhancing the coalescing mechanism.

The following sections introduce some terminology and how global memory requests have changed over time. Due to contradictory information from NVIDIA documentation, it was necessary to conduct experiments to understand the sizes of the transactions at different levels in the memory hierarchy. After understanding the coalescing mechanisms in modern GPUs, we propose a new approach to estimate the number of transactions and consequently the data traffic. The analytical model [47] is adjusted accordingly to reflect our proposed approach.

6.3.1 Terminology

This section introduces NVIDIA's terminology [12, 18]:

Request: A command issued into the hardware memory unit to perform some action, e.g., load data from a memory address.

Sector: Aligned 32-byte chunk of memory in a cache line or device memory. An L1 or L2 cache line is four sectors, i.e., 128 bytes.

Transactions: Represents the smallest unit of data to be moved between two memory units. Regardless of the requested data, memory units always serve the requests with fixed-size chunks of data. Each request is served with one or more transactions depending on the access pattern and amount of data.

Wavefronts: It is the maximum unit of work per cycle going through the pipeline stages in the memory units. The number of cache lines or sectors that can be accessed in a single

wavefront may be limited due to the need "[...] for a consistent memory space [...] as well as various other reasons" [18]. A wavefront comprises work items that can be processed in parallel, but distinct wavefronts are serialized. "The L1TEX unit has internally multiple processing stages operating in a pipeline." [18].

6.3.2 Memory requests evolution across architectures

To better understand how coalescing evolved, the following list summarises the documented changes across architecture generations [12]. CC is short for Compute Capability.

CC 1.0 and 1.1: Coalescing is supported per half-warp, i.e., the groups of threads with identifiers $[0-15]$ and $[16-31]$. The accessed words within the half-warps to be of sizes 4, 8, or 16 bytes. Moreover, they must be perfectly aligned. Finally, the ith thread must access the ith word. When the requirements for coalescing are met, the number of transactions per half-warp depends on the word size: one 64 B memory transaction, one 128 B transaction, or two 128 B transactions for words of 4, 8 or 16 bytes, respectively. If the requirements fail, each half-warp receives 16 transactions from the memory units, one per thread, each 32 B.

CC 1.2 and 1.3: Some constraints from earlier architectures are relaxed. For instance, words can be accessed in any order, i.e. the ith thread does not have to access the ith word. Coalescing is still applied per half-warp, but implements an iterative algorithm that tries to coalesce the thread's requests as much as possible to a minimum 32 B sized transaction. In summary, the algorithm works as follows: *(i)* select the lowest active thread ID and its requested memory segment, $MemSeg$; *(ii)* find all other active threads that requested words falling within the memory segment $MemSeg$; *(iii)* successively reduce the transaction size in half, discarding the upper or lower half of the transaction if unused, until it is not possible to shrink any further or the transaction size reaches the minimum 32 B size; *(iv)* threads that are serviceable with the resulting transaction are marked as inactive; *(v)* the request is dispatched, and *(vi)* if there are active threads left, repeat the process until all threads in the half-warp are serviced.

CC 2.x and onwards: The Fermi architecture introduced L1 cache for global memory accesses. The data transferred between L1 and device memory goes through the L2 cache. Some architectures have opt-in caching in L1 and it may be enabled or disabled by default. Cache line sizes in L1 and L2 are 128 B, composed of 4 sectors. Each cache line maps to a segment of 128 B in global memory. L2 has an access granularity of 32 B, thus, it is possible to access a single sector within the cache lines. In contrast, L1 serves the entire cache line resulting in 128 B transactions. Requests are processed per warp rather than half-warp as in previous generations. Moreover, the number of requests varies with the word size being accessed: *(a)* a single memory request for words up to 4 B, *(b)* for 8 B words, two requests

> are issued, one per half-warp, and *(c)* finally, if the word size is 16 B, there are four memory requests, one per quarter-warp. The memory requests are broken down into cache line requests that are issued independently. The request is serviced at L1, or L2 throughput in a cache hit, otherwise at the throughput of device memory.

Concerning memory unit's access granularity, there is some ambiguity in documentation. While it is well established across all NVIDIA architectures that the L2 cache serves transactions with a granularity of 32 B, for L1 cache different sources point towards different values. CUDA Programming Guide [12] states a 128 B granularity on L1 for CC 2.0 and onwards. However, Kernel Profiling Guide [18] suggests a 32 B granularity[1]. One of the profiling metrics for L1 is described as: "Total number of bytes requested from L1. This is identical to the number of sectors multiplied by 32 byte, since the minimum access size in L1 is one sector.". Therefore, according to this quote, the access granularity in the L1 is not full cache lines, but 32 B sectors, just like L2 cache.

Moreover, both manuals suggest that the number of requests issued to the memory units is also different. As described earlier, the Programming Guide reports that the word size influences the number of issued requests. However, according to [18], one global LD/ST instruction issues a single request to L1 cache regardless of the word size. A request is processed in the memory unit to create work packages, the wavefronts. Items within the same wavefront are processed in parallel, but distinct wavefronts are serialised.

> "When an SM executes a global or local memory instruction for a warp, a single request is sent to L1TEX [unified L1-Data and Texture caches]. This request communicates the information for all participating threads of this warp (up to 32). [...] the request requires to access a number of cache lines, and sectors within these cache lines. The L1TEX unit has internally multiple processing stages operating in a pipeline." [18]

Concerning memory requests, the analytical model [47] does not take this level of detail into consideration, and is left for future work. Regarding the L1 access granularity, it is an important detail as it may influence the amount of memory that is fetched from global memory. This specific topic is further explored in the following sections.

6.3.3 The impact of mixed access granularity in memory hierarchy

Figure 6.2 illustrates how the transactions flow through memory hierarchy for a given memory request. In the example, the access granularities are 32 B, 64 B and 128 B for L1, L2 and DRAM, respectively. Moreover, it assumes the request accesses a single word sized 4 B.

When L1 receives a memory request, it checks if the data is present and valid in its cache lines. If it is, then L1 can respond to the request transferring a sector of 32 B. Even through the request

[1] The profiling guide is documentation for the Nsight Compute profiling tools, released in 2018 (https://developer.nvidia.com/nsight-compute-history), with support for Pascal (CC 6.x) and later GPUs (https://docs.nvidia.com/nsight-compute/2022.1/ReleaseNotes/index.html#gpu-support)

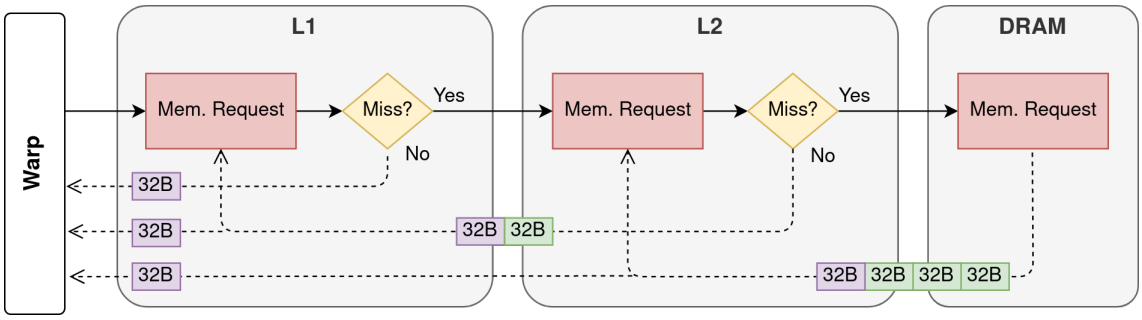


Figure 6.2: Example of transactions flow across the memory hierarchy for a memory request accessing a single memory address. In the figure, L1 has 32 B granularity, L2 has 64 B granularity and DRAM has 128 B granularity.

is a for single 4 B word, the minimum unit to read data in L1 is the 32 B. In this case, the data is transferred at L1 throughput, as no other units are involved. However, if the requested data is not in L1, then the request is routed to L2. L2 has a 64 B access granularity. Therefore, for each 32 B sector miss in L1, L2 sends one 64 B transaction, which maps to two 32 B sectors in L1. Similarly, if the data is not present in L2, the request is routed to DRAM, with higher latency times. In our example, DRAM transactions are 128 B in size.

Consider that the threads in the warp access a 4 B word, but the memory addresses are scattered. In other words, the threads are accessing memory positions that hit in distinct sectors or cache lines, consequently a different transaction is needed per thread. If there is a cache hit in L1, the maximum amount if data is $WarpSize \times L1SectorSize = 32 \times 32 = 1024$ B. However, if the requests are served by DRAM, then the transferred data is 4 times larger. Of the 4096 bytes of data, only 128 bytes are used.

Our example shows the importance of access granularity at caches and DRAM and how it can impact the data traffic in the memory hierarchy. The following sections discuss the granularities for L1 and DRAM memory, given the contradictory information in NVIDIA documentation.

6.3.4 Experimental setup for determining L1 access granularity

This section describes a simple experiment conducted on two Pascal GPUs (CC 6.x) that should help clarify what is the granularity for L1 cache. The experiment consists in crafting a kernel that performs global read and write memory accesses with a configurable access stride inter-threads, i.e., let s be a striding distance, then thread t_i accesses the address $baseAddr$, and thread t_{i+1} the address $baseAddr + s \times wordSize$. The question is if L1 has a 32 B or 128 B access granularity. Therefore, we assume both cases and estimate by hand the number of transactions and transferred bytes. Our estimations are compared to profiling data to derive any conclusion.

Listing 6.2 shows the kernel in CUDA used for this experiment, moving one element from the `src` array to `dst`. The parameter `stride` configures the striding distance between consecutive threads. The array index, `tid`, is calculated such that each thread accesses a unique memory position per thread and thread block, preventing any cache hits inter-warps (which would affect

the number of transactions from DRAM). The grid geometry is large enough to let the profiler collect the hardware counter values accurately. The profiling is repeated several times to ensure negligible variation in the counters, suggesting the kernel is properly set.

```
#define DATA_TYPE float
__global__ void mem(DATA_TYPE *src, DATA_TYPE *dst, int stride) {
   int tid = blockDim.x * blockIdx.x * stride + threadIdx.x * stride;

   dst[tid] = src[tid];
}
void main() {
   ...
   mem<<<5,256>>>(dev_src, dev_dst, 1);
   ...
}
```

Listing 6.2: Simple experiment to analyse the number of transactions on different scenarios

The NVIDIA profiler `nvprof` is used to get the number of transactions and amount of transferred data. Table 6.1 presents the relevant metrics or events for this experiment. Unfortunately, the number of transactions from L1/Tex cache is not available for the GPUs at hand. Although there is a metric called `tex_cache_transactions`, described as "Unified Cache Transactions", it is not giving the expected information. The Nsight documentation [18] has a table that compares the metrics available in `nvprof` with the introduced metrics in Nsight, showing a migration path[2]. Figure 6.3 shows the equivalent metrics in Nsight for `tex_cache_transactions`. Essentially, it is an average percentage of peak rate achieved during load/store operations in the L1/Tex cache. Therefore, it is not the number of transactions in the unified L1/Tex cache serves, as the metric's name suggests[3]. It was not possible to find a substitute metric for the number of transactions L1 serves. However, it can be inferred from other metrics. The number of transactions served from L2 depends on the access granularity at L1 being 128 B or 32 B, and the rest of the experiment relies on this fact.

```
l1tex__lsu_writeback_active.avg.pct_of_peak_sustained_active
+ l1tex__tex_writeback_active.avg.pct_of_peak_sustained_active
```

Figure 6.3: Equivalent metrics in Nsight for the `tex_cache_transactions` in `nvprof`.

Assume an access granularity of 128 B at L1 for a word size of 4 B. Each warp makes a single memory request according to CUDA Programming Guide [12] and Nsight [18]. Furthermore, consider an access stride of 128; therefore, each thread performs a load and store in distinct cache lines, ensuring cache misses. Consequently, each warp has to be served with 32 transactions of 128 B, assuming L1 has the 128 B access granularity. Due to cache misses in L1, the requests

[2] https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html#nvprof-metric-comparison

[3] `nvprof` is used instead of Nsight because the latest release dropped support for Pascal GPUs, and older versions are not compatible with the latest CUDA Toolkits.

Metric name	Metric description
`l2_read_transactions`	Memory read transactions seen at L2 cache, which result from L1/Tex cache requests and other units
`l2_write_transactions`	Memory write transactions seen at L2 cache
`dram_read_transactions`	Device memory read transactions
`dram_write_transactions`	Device memory write transactions
`l2_global_load_bytes`	Bytes read from L2 for misses in Unified Cache for global loads
`l2_local_global_store_bytes`	Bytes written to L2 from Unified Cache for local and global stores
`dram_read_bytes`	Total bytes read from DRAM to L2 cache
`dram_write_transactions`	Total bytes written from L2 cache to DRAM

Table 6.1: Relevant profiling metrics in `nvprof` for the experiment.

are routed to L2. Access granularity for L2 is 32 B, thus the number of transactions from L2 to L1 increases by a factor of 4, because 4× 32 B sectors from L2 are needed per cache line in L1. Hence, the number of transactions per warp from L2 to L1 is 128. The requests in L2 also result in cache misses and are routed to DRAM. Assuming the DRAM access granularity is the same as L2 (more on this topic later), the number of transactions per warp served from DRAM to L2 is also 128, as imposed by L1. Given the kernel launch configuration `mem«<5,256»>`, the number of warps is $5 \times \frac{256}{32} = 40$. Hence, the total number of transactions from DRAM to L2 and L2 to L1 is 5120. The total bytes transferred from DRAM to L2, and from L2 to L1, is the number of transactions times the size of each transaction, i.e. $5120 \times 32 = 163840$ B. Note that the number of bytes needed for the kernel is $5 \times 256 \times 4 = 5120$ B, i.e. the total number of threads times the size of each requested word. The efficiency is $\frac{5120}{163840} \approx 3\%$.

Consider the alternative scenario where the access granularity to L1 is 32 B. Because the access stride is still 128 and coalescing is not possible, each warp memory request results in 32 transactions of 32 B. Note that the number of transactions has not changed compared to the previous scenario given there is one transaction per thread in the warp, but the size of the transactions between L2 and L1 is smaller. Since the access granularity of DRAM, L2 and L1 match, transactions do not have to be split or fused along the memory hierarchy. Therefore, the number of transactions served from DRAM to L2, L2 to L1, and L1 to the warp is $32 \times 40 = 1280$ (40 is the number of warps). Transactions size is 32 B, thus each unit in the hierarchy transfers 40960 B. Efficiency, in this case, is about 12.5%.

The next step is to profile the application with `nvprof` and collect the metrics in Table 6.1. The profiling results, shown in Table 6.2, approximate the scenario where L1 transactions have a granularity of 32 B. Interestingly, the number of write transactions from L2 to DRAM is higher than expected. It is particularly unexpected that the number of write transactions and amount of written data in DRAM is 3.3× higher compared to L2. If the ratio was approximately 4×, it could suggest that writing operations on DRAM require writing full cache lines or segments of

128 B, while read operations can access 32 B sectors within cache lines. Figure 6.4, from the profiler's GUI, shows that all writing operations in DRAM are coming from L2. Writing and reading operations between L1 (unified cache) and L2 are the same, as expected. One plausible explanation for such observations, is that both GPUs used in the experiment are not operating exclusively for running CUDA applications. They are also used to render the OS desktop interface. Therefore, it might interfere with the collected metrics.

Metric name	Value
`l2_tex_read_transactions`	1280
`l2_tex_write_transactions`	1280
`dram_read_transactions`	1286
`dram_write_transactions`	4245
`l2_global_load_bytes`	40960 B
`l2_local_global_store_bytes`	40960 B
`dram_read_bytes`	41152 B
`dram_write_bytes`	135840 B

Table 6.2: Number of memory transactions and total size on L1/Tex, L2 and DRAM

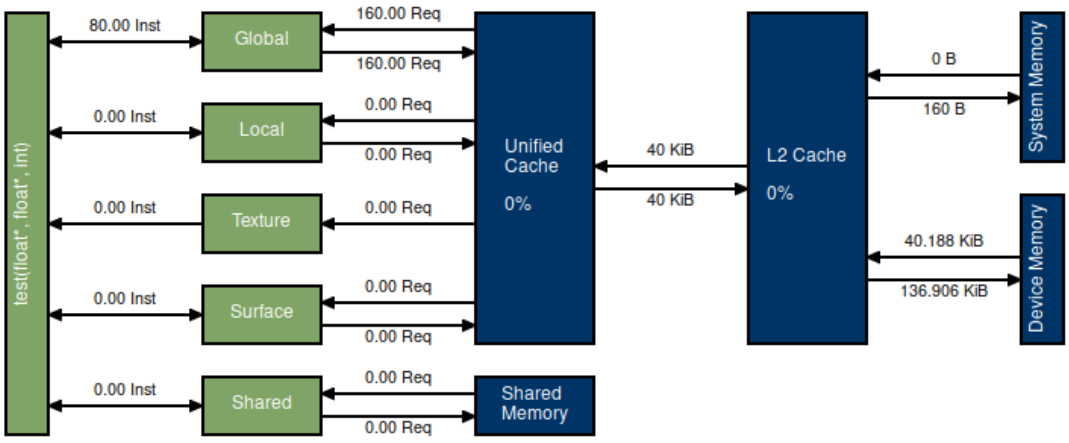


Figure 6.4: Memory transactions and moved data in physical units (blue) and logical units (green)

Despite the unexpected results in writing operations from L2 to DRAM, we can conclude that in Pascal the access granularity to L1 is 32 B. Our results are supported by Lia et al. [48] observations for L1 access granularity, listed below. In conclusion, starting with Maxwell, L1 access granularity is 32 B, the same as L2 cache.

- Kepler (CC 3.x): 128 B
- Maxwell (CC 5.x): 32 B
- Pascal (CC 6.x): 32 B

6.3.5 DRAM access granularity

The previous section discussed the memory access granularity for L1 cache and showed how it can impact the amount of data that is fetched from DRAM. It is well established that L2 is accessed in units of 32 B. However, DRAM is not mentioned in the documentation. DRAM is off-chip memory and one of the reasons why NVIDIA does not mention the granularity for DRAM is because it depends on the used memory technology for DRAM modules on the board.

For instance, data access granularity on GDDR5, GDDR5X and GDDR6 is 32 bytes [7]. However, HBM allows 32 or 64 byte transactions [16, 49]. Both Pascal GPUs used for the experiment have GDDR-based memory technologies, but some data-center Pascal GPUs use HBM. Therefore, DRAM access granularity is not an architecture parameter, but rather device dependent. An NVIDIA engineer in the forums support this conclusions and shares the results for an experiment in a GPU with HBM2 memory controller, showing it fetches 64 B from DRAM for every cache miss in a 32 B sector miss in L2[4].

Based on the experiment results in Table 6.2, the number of transactions between DRAM and L2 have a ratio 1:1, which means that DRAM in the tested devices has a 32 B access granularity.

6.3.6 Estimating number of memory transactions

Previous sections discussed how memory requests evolved across architecture iterations. It is possible to conclude that the distinction between coalesced and uncoalesced memory instructions is no longer suitable for current GPUs. Coalescing still happens to some extent, in the sense that if the threads within the warp access memory addresses that fall in a common sector, then that sector serves multiple threads, hence increasing efficiency. This section proposes a new approach to estimate the number of transactions to DRAM and data traffic. Our approach targets CC 2.x and later NVIDIA GPUs. Similarly to the original analytical model [47], our work does not model cache. Therefore, we are concerned with estimating traffic from/to DRAM only.

Let *accessGranularity* be the largest access granularity imposed by DRAM, L2 or L1. Furthermore, assume that allocated memory is naturally aligned (default for contiguous memory allocation, [12, Section 5.3.2]) and each memory address belongs to a memory segment. Memory fetches are done per segment rather than by word, similar to CPUs. Each thread within the warp, T, requests a memory address, $memAddr$, which falls in a memory segment with a base address $\lfloor \frac{memAddr}{accessGranularity} \rfloor$. The number of unique memory segments to be accessed is the number of transactions necessary for executing the memory instruction. Multiplying the number of transactions by the *accessGranularity* gives the total bytes transferred. Algorithm 2 illustrates this procedure.

6.3.7 Adapting for AST analysis and OpenMP offloading

Algorithm 2 assumes it is possible to calculate the physical/virtual memory address requested by each thread. However, that problem can be abstracted when the analysis is performed at AST level.

[4] https://forums.developer.nvidia.com/t/pascal-l1-cache/49571/18

Algorithm 2 Determine the number of transactions per warp's memory instruction

Require: $accessGranularity$
Require: $MemInstr$
1: $allMemSegments \leftarrow \{\}$ ▷ A set data structure for storing unique memory segments accessed when executing $MemInstr$
2: **for** thread T in warp W **do**
3: $memAddr \leftarrow MemInstr.getThreadAddr(T)$ ▷ Calculate the memory addressed requested by thread T
4: $memSegmentAddr \leftarrow \left\lfloor \frac{memAddr}{accessGranularity} \right\rfloor$ ▷ Calculate the memory segment base address
5: $allMemSegments.insert(memSegmentAddr)$ ▷ Add the memory segment address to the set
6: **end for**
7: $numMemTransactions \leftarrow allMemSegments.count()$
8: $totalMemTransf \leftarrow numMemTransactions \times accessGranularity$ ▷ One memory transaction per memory segment. Each transaction has $accessGranularity$ bytes

For instance, with respect to load/store operations on arrays, the memory address is calculated as $baseAddr + offset$. The base address is the starting memory position for the array, which corresponds to a contiguous memory block in memory. In C languages, the base address is abstracted by the array variable or a pointer. The offset depends on the array index and the array data type. Since the base address is common to all threads within the warp executing the instruction, the offset is enough to calculate the number of transactions needed to fulfil the memory request.

```
1  #pragma omp distribute parallel for collapse(2)
2  for (int i = 0; i < N; i++) {
3     for (int j = 0; j < N; j++) {
4        C[i * N + j] = 0.0;
5        for (int k = 0; k < N; ++k) {
6           C[i * N + j] += A[i * N + k] * B[k * N + j];
7        }
8     }
9  }
```

Listing 6.3: Matrix multiplication example with OpenMP

Consider the matrix multiplication in Listing 6.3 and assume array elements are floats. Our goal is to determine the number of transactions using AST analysis. In the innermost loop, `k`, there are memory accesses to matrices `A`, `B` and `C`.

Assume that a thread t_k computes the kth iteration, and t_{k+1} the $k+1$ iteration. Looking at the array subscripts, we can conclude that there is a load for `A[i*N+k]`, `A[i*N+(k+1)]`, `B[k*N+j]`, `B[(k+1)*N+j]` and `C[i*N+j]`. The striding distance for array accesses by consecutive threads is 1 for array `A`, N for array `B` and 0 for `C`.

Assuming that memory transactions from DRAM are 32 B, then the two loads to `A` fall in the same memory segment. Note that the base address is the same, and the offset between the two loads is $stride \times dataType = 1 \times 4$ B. With respect to array `B`, the offset is $4 \cdot N$. If it is larger than 32 B (access granularity), then both loads hit distinct memory segments. Let T be the number of

data elements that fit in a transaction. In this case, a transaction sized 32 B can move 4 floating-point scalars. Then, the number of words that fit in a single transaction is $\left\lceil \frac{T}{stride \times dataType} \right\rceil$.

Using the AST analysis it is necessary to determine the striding distance and know the data type associated to the array. With respect to the striding distance, one possible solution is to extract the expression in array subscript and use algebraic symbolic analysis. In the innermost loop `k`, we assume any other variable in the expression is constant throughout the loop. It is true for the control variables `i` and `j`, but assuming it for other variables may be a limitation. As the loop increments the variable `k` in steps of 1, the striding distance for `B` is $B_{k+1} - B_k = ((k+1) \cdot N + j) - (k \cdot N + j) = N$. Note that this approach only works for loops with constant steps.

So far, it was assumed that consecutive threads compute consecutive loop iterations. In the context of OpenMP, that is not necessarily true as it depends in the scheduling policy. For instance, with static scheduling, the iteration space is split in even chunks which are assigned to threads in order. Consider 1024 iterations and 8 threads, resulting in chunks sized 128 iterations. Thread t_0 computes the iteration range $[0, 127]$, t_1 the range $[128, 255]$ and so on. The chunk size is configurable. For instance, it can be set to 1 and each thread is assigned multiple chunks of size 1. Considering the same example, t_0 computes the iterations $\{0, 8, ...\}$, t_1 the iterations $\{1, 9, ...\}$ and so forth. Hence, it is to account both access stride and chunk size to determine the distance between memory elements accessed by consecutive threads in the same memory instruction.

Equation 6.10 summarises the steps to calculate the number of memory transactions per memory operation. $ThreadsServedPerMemTrans$ is the number of threads served with a single transaction of size $archMemTransactionSize$. It depends on array data type ($dataType$), striding distance between consecutive loop iterations ($stride$) and the chunk size in OpenMP scheduling ($chunckSize$). If the fraction is equal or less than 1, then each transaction serves a single thread. Finally, the total number of transactions necessary for an array access, is the division of number of threads in the warp by how many threads each transaction fulfils. $ThreadsServedPerMemTrans$ is the number of threads that are served with a single transaction of size $archMemTransactionSize$.

$$\begin{aligned} TransactionsPerMemInstr &= \frac{archThreadsPerWarp}{ThreadsServedPerMemTrans} \\ ThreadsServedPerMemTrans &= \left\lceil \frac{archMemTransactionSize}{dataType \times stride \times chunckSize} \right\rceil \end{aligned} \tag{6.10}$$

Our formulation for the problem assumes that $dataType$ is never larger than the transaction size. If that could be possible, an additional case would be necessary to indicate that a thread may need more than one transaction and the striding distance and chunk size would be irrelevant in that case. According to [12], the supported words sizes are 1, 2, 4, 8 and 16 bytes. Thus, as long as the array data type is a primitive data type in C/C++, our methodology works. However, further work is necessary to understand how compilers deal with structures in the OpenMP context, whether struct fields are aligned and how does it interfere with memory accesses.

6.3.8 Updating the model

This section discusses the necessary changes to Kim et al. analytical model [47] to reflect how memory requests work in modern GPUs, dropping the distinction between coalesced and uncoalesced memory requests.

In the original model, the memory latency was a weighted average because the latencies would vary depending on coalesced and unconcealed accesses. With our proposal, the distinction is dropped, as shown in Equation 6.11. With respect to round trip latencies ($archMemLatGlobal$), it should be constant for all transactions given the fixed-size and that all requests hit DRAM. However, different memory instructions result in varying number of transactions. Therefore, an average number of transactions is used to estimate the average memory latency.

$$MemLat = archMemLatGlobal + archMemDepartDelay \times avgTransPerMemInstr \tag{6.11}$$

The departure delay between consecutive memory transactions was distinguished in coalesced and uncoalesced memory accesses. Consequently, Kim et al. [47] proposed a weighted average sum. In our proposal, the departure is constant for all transactions and matches the device property $archMemDepartDelay$, obtained via micro-benchmarking (see Equation 6.12).

$$MemDepartureDelay = archMemDepartDelay \tag{6.12}$$

Finally, the total number of cycles spent in memory operations, shown in Equation 6.13, is the number of memory instructions multiplied by the average latency per memory instruction.

$$Mem\_cycles = MemLat \times appMemInsts \tag{6.13}$$

6.4 Modelling instructions cost

The original analytical model [47] uses a 4 cycle latency cost for all instructions. However, modern GPUs have a larger ISA and diversified execution units. For example, Arafa et al. [22] shows that in a Pascal GP100 Streaming Multiprocessor, single-precision floating-point (SFP) addition, subtraction and multiplication have latencies of 6 cycles, while a division on average costs 408 cycles. Besides, the number of execution units available per warp scheduler for different operation types is not equal. In GP100, each warp scheduler has 32 execution units for single-precision floating-point, 16 double-precision floating-point units and 8 special function units (SFU) for transcendental operations (e.g., trigonometric and logarithmic operations). The warp size in GP100 is 32. Therefore, when the warp scheduler dispatches a 32-bit floating-point instruction, there are enough units to execute that instruction in parallel for all the threads. In case of a transcendental instruction, the instruction has to be issued multiple times to the SFU units for sub-groups of 8 threads.

As discussed earlier for the CPU analytical model (Section 4.2.2), there are different metrics for cycle-accurate estimations concerning instructions cost: *(a)* latency, and *(b)* throughput or reciprocal throughput.

Similarly to the CPU analytical model, our approach uses throughput metrics. NVIDIA discloses the instruction throughput in the CUDA Programming Guide [12, Section 5.4] for different types of native instructions and Compute Capability architectures. The throughput values are described as the number of results per clock cycle per multiprocessor.

Consider again the Pascal architecture and the GP100 SM (Compute Capability 6.0). The throughput for addition/subtraction integer operations is 64 results per cycle per SM [12]. Considering the two warp schedulers per SM, then each partition outputs 32 results per cycle. Each partition has 32 CUDA Cores, thereby each execution unit calculates one integer result per cycle. Note that additions have a 6 cycle latency [48, 22]. However, floating-point and integer units in the CUDA Core are fully-pipelined [4], therefore values from NVIDIA are the peak number of results per clock cycle, assuming a long sequence of independent instructions of the same type that saturate the pipelines.

Equation 6.14 shows the updated equation for $Comp\_cycles$, the number of cycles each thread consumes issuing instructions. The $Troughput(i)$ is the throughput provided by NVIDIA for a given instruction i. The numerator $archThreadsPerWarp \times archWarpsSchedPerSM$ is the total number of threads per SM, as the throughput disclosed is per SM. Thereby, the equation accumulates the average cycle-cost per instruction at peak throughput.

$$Comp\_cycles = \sum_{i \in Instructions} \frac{archThreadsPerWarp \times archWarpsSchedPerSM}{Troughput(i)} \tag{6.14}$$

6.5 Collecting device-specific parameters

6.5.1 Data transfers between CPU host and GPU

In the typical CPU-GPU system, the host CPU runs the Operating System and applications. Whenever an application wants to use the graphical or computational capabilities of the GPU, the driver on the host-side has to communicate with the GPU and transfer the kernel code. Besides, CPU and GPU have private system memory. Therefore, kernel code executing on the GPU cannot directly access the host memory. Instead, data must be transferred to the GPU before the kernel starts, and the computation results are carried back to the host's memory. The required data movements may limit the benefits of offloading. Therefore, it is vital to consider data movement overheads in the analytical model.

CPU and GPU attached to the same board can communicate through the interconnects to transfer data back and forth and issue commands. In spite of the high theoretical bandwidths for the interconnects, the bandwidth in real-world usage is typically much lower. Multiple factors bottleneck the effective bandwidth. For example, on the host side, the memory controller frequency, the RAM speed, and the number of channels for the RAM. On the GPU side, how fast data can

be written on device memory is also a factor, although memory chips on GPUs have higher bandwidths than their CPU counterparts. Moreover, how the GPU is attached to the system is important as well. Most GPUs connect via the standard PCI-e connector [41], but recently NVIDIA unveiled their proprietary NVLINK interconnector; other alternatives exist as well [50].

Besides all the aforementioned hardware-related factors, the host's memory allocation method also impacts the effective throughput. In modern operating systems, memory is split into user and kernel spaces. In contrast to kernel space, user space is pageable memory by default [69]. Pages are basically well-defined memory units and allow the OS to swap them to secondary memory (e.g. disks), thus freeing physical memory which is necessary for running the active processes.

Data transmission between peripherals and the main memory is managed by a dedicated controller, known as the DMA (direct memory access). Pageable memory is troublesome for DMA. Therefore, DMA requires the drivers to allocate page-locked buffers before data transferring [32, Chapter 15]. Page-locked memory is allocated memory that permanently resides on physical memory and the OS cannot swap it to secondary memory. As a result, when an application wants to move data to the GPU, the NVIDIA driver needs to check if the target memory is locked. If it is not, it must create locked buffers, move the data from the source to the newly allocated buffer, and finally, it can start the DMA process to transfer data from host to device or vice-versa. Managing these operations introduces overheads and limits the throughput, especially if page misses occur in the process.

As a mitigation measurement, CUDA gives the option to allocate memory on the host as page-locked (also known as *pinned memory*) using the `cudaHostAlloc` runtime API [14] in place of the standard `malloc`. When an application transfers data to the GPU and the memory is pinned, the CUDA driver does not need to create additional buffers or clone data. Instead, it can initiate the data transfer right away. Therefore, communications between host and device become faster, at the expense of putting more pressure on the host's main memory (in the sense that pinned memory cannot be swapped out).

In the CUDA programming model, pinned memory usage is optional and requires the programmer to use the adequate runtime or driver APIs [13, 14] for allocating the memory on the host. However, our work focus on OpenMP offloading, which raises the question of whether compilers such GCC or Clang use pinned memory or not. It was impossible to find a definitive answer in the existing literature. Therefore, some empirical experiments were conducted and are described next.

The experiment uses kernels from the UniBench [56], an extension to Polybench [75] that adds OpenMP offloading support. If an application makes use of pinned memory, it must use the dynamic APIs `cudaHostAlloc` or `cuMemAllocHost`, from runtime and driver libraries respectively. The NVIDIA profiler, `nvprof` can trace all API calls, enabled with the CLI parameters shown in Figure 6.5. Therefore, the experiment consists in launching all the kernels in the GPU, parse the API trace, and look for the mentioned API calls.

```
$ nvprof --print-api-trace --profile-api-trace all <executable>
```

Figure 6.5: Using `nvprof` with API tracing enabled

Listings 6.4 and 6.5 highlight some of the API calls for Clang and GCC, respectively[5]. It is possible to conclude that neither Clang nor GCC use pinned memory in any Polybench kernels due the lack of calls to `cudaHostAlloc` or `cuMemAllocHost`. Therefore, memory allocation on the host side is done through the standard `malloc`. This experiment is not sufficient to conclude that Clang or GCC never use pinned memory, which would require inspecting the source code or finding some evidence in the documentation. However, experimental evaluation in this work uses the Polybench suite, and since neither compiler uses pinned memory in this particular suite, pinned memory is not addressed in this work.

```
cuStreamCreate // several streams are created
cuStreamCreate
...
cuDeviceGetAttribute // get information about GPU
...
cuMemAlloc // memory allocation on GPU
cuCtxSetCurrent
cuMemAlloc
...
cuMemcpyHtoDAsync // transferring from Host to Device
cuMemcpyHtoDAsync
...
// launch kernel
cuLaunchKernel (__omp_offloading_fd01_37df10a_mm3_OMP_l120 [80])
...
```

Listing 6.4: Polybench 3MM API call trace when compiled with Clang

[5] As a side note, the traces contain some interesting details. First, Clang seems to create several streams and transfer the data asynchronously. Secondly, GCC uses just-in-time compilation, proven by the calls to NVRTC library (`cuLinkCreate`, `cuLinkAddData`, etc.). In contrast, Clang compilation flow generates machine code for the target GPU and embeds PTX in the binary so that it is possible to use JIT on upcoming architectures.

```
cuDeviceGetAttribute // get information about GPU
...
cuLinkCreate // calls to NVRTC (runtime compilation library)
cuLinkAddData
cuLinkAddData
...
cuLinkComplete
...
cuMemAlloc // memory allocation on device
...
cuMemcpyHtoD // transferring from Host to Device
...
cuLaunchKernel (mm3_OMP$_omp_fn$0 [81])
...
```

Listing 6.5: Polybench 3MM API call trace when compiled with GCC

The remainder of this work focuses on data transferring with pageable memory. Since the effective throughput depends on several hardware characteristics, as mentioned earlier, a possible approach is to benchmark transferring speeds from host to device and vice-versa with sample applications. This task is simplified because NVIDIA includes a benchmarking example in the CUDA samples that does exactly that [6]. Running the benchmark should give a good approximation for the average observed throughput in real-world applications. The application allocates memory with `malloc` and pinned memory with `cudaHostAlloc`. Moreover, it performs transferring in both directions, i.e. host to device and device to host. Finally, it measures the bandwidths with varying block sizes.

The obtained results are summarized in Table 6.3, showing memory movements in both directions, i.e. device to host (DtH) and host to device (HtD). It is important to realise the larger the block sizes, the higher the throughput, until data blocks are sufficiently large and the bandwidth reaches a plateau. The presented averages are for blocks with sizes larger than 1GB. For small data blocks, the latencies involved in the DMA process are the primary factor, limiting considerably the observed bandwidth.

Memory allocation	DtH (GB/s)	HtD (GB/s)
Paged	11.76	9.37
Pinned	13.18	12.4

Table 6.3: Memory transferring bandwidth with paged and pinned allocated memory

6.5.2 Memory bandwidth

Another input parameter for the analytical model is the global memory bandwidth, *archGlobalMemBandwidth*. Usually, vendors disclose memory speed rates, clock frequencies and bandwidth

[6] https://github.com/NVIDIA/cuda-samples

for marketing purposes. In the case of the GPUs, vendors typically disclose one or more of the following metrics:

Effective memory speed (Gbps). The effective rate at which data is transferred per lane. It depends on the memory technology. For instance, GDDR3 and GDDR4 memories transfer two units per cycle, while GDDR5 transfers four units per cycle.

Device memory bandwidth (GB/s). It is a theoretical maximum bandwidth of the global memory, assuming all memory bus lanes are used at the same time at full speed. This metric is the one needed for the analytical model and corresponds to $archGlobalMemBandwidth$.

Memory clock. It is the memory chip clock frequency, which is not the same as the data rate because, as mentioned, the amount of data transferred per clock varies with the memory technology.

Bus width. The total number of lanes from the GPU memory controllers to the memory chips onboard.

All metrics are related to each other. To illustrate it, consider the information for an NVIDIA GTX 1070 card in Table 6.4. Consider the memory clock frequency. In GDDR5, each clock cycle in the data channels transfers 4 data units (or bits). Therefore, the effective data rate is $2002 \times 4 = 8008Mbps \approx 8Gbps$.

Memory clock frequency	2002 MHz
Effective memory speed	8 Gbps
Memory bus width	256 bit
Bandwidth	256.3 GB/s
Memory type	GDDR5

Table 6.4: NVIDIA GTX 1070 memory properties

For the memory bandwidth, it is necessary to account the total number of lanes available. The data rates for different GDDR type memories are presented in Table 6.5. Besides, the bandwidth is measured in GB (gigabytes), while the effective memory speed is in Gb (gigabits): dividing the later by 8 converts to GB. The bandwidth is calculated multiplying the effective memory speed by the bus width, and then convert bits to bytes: $\frac{2002 \times 4}{8} \times 256 = 256256MB/s \approxeq 256.3GB/s$.

The Equation 6.15 generalises the equation to calculate $archGlobalMemBandwidth$ when the disclosed information from vendors does not include the peak device memory bandwidth.

$$archGlobalMemBandwidth = \frac{MemClockFreq \times MemDataRate}{8} \times MemBusWidth \tag{6.15}$$

Memory type	Memory Data Rate (relative to clock frequency)
GDDR3, GDDR4	2x
GDDR5	4x
GDDR5X	8x
GDDR6	8x
GDDR6X	16x

Table 6.5: The memory data rates for GDDR memories [7]

6.5.3 Memory latency

The original analytical model [47] uses different latencies to model the memory accesses accurately. For example, it considers the round-trip latency to access the device memory and departure delays between consecutive transactions. The transactions are distinguished in coalesced and uncoalesced, and each has different departure delays.

Previous sections argued that the distinction between coalesced and uncoalesced memory requests is no longer proper for modern GPU architectures. Therefore, modelling memory accesses is simplified, and the necessary parameters are latency for each transaction, $archMemLatGlobal$, and the departure delay between successive transactions, $archMemDepartDelay$.

However, the original analytical model [47] does not explain how they measured the departure delays, only stating the value are obtained via micro-benchmarks. Besides, Chikin et al. [30] do not mention the value they use in the experimental evaluation, nor how they measured it. Given the missing details of the global memory hierarchy, it is assumed that transactions can be issued every clock cycle, i.e. $archMemDepartDelay = 1$.

Concerning memory access latency, the parameters are obtained via micro-benchmarking. The Jia et al. [48] describes how to obtain the parameters and lists values for the Pascal architecture, which are used for the experimental evaluation in this work.

6.6 Collecting application parameters

This section addresses collecting parameters for the analytical model that characterise the application under analysis. For each parameter, it discusses different possible approaches and the one implemented in this work. The necessary parameters are listed in Table 5.1.

6.6.1 Number of instructions and memory operations

For estimating the application's execution time, the number of operations per thread is needed for the analytical model, and different approaches are possible.

Concerning static analysis approaches, different approaches can be used. For instance, Chikin et al. [30] count the instructions at low-level representations (IR). Other possible paths for NVIDIA GPUs is analysing the pseudo-assembly PTX code, which is agnostic to a specific NVIDIA architecture, or using machine code instructions (SASS) for a more accurate analysis.

Our approach for this work is to collects metrics at AST level. The same approach is used for CPU and GPU analytical models. Details regarding operation counting are discussed in Section 8.?. The operations at AST, which inherit the semantics of the C language, are then associated to native instructions in the GPU. That way, we can use the instruction throughput table provided by NVIDIA, as discussed in Section 6.4.

6.6.2 Grid geometry

The grid geometry parameters are related to the configuration used to launch the kernel on the GPU: the number of thread blocks, $appThreadBlocks$, and the number of threads per thread block, $appThreadsPerBlock$.

In the CUDA programming model, the parameters are explicitly set by the user at the kernel launch. In OpenMP offloading context, the compiler automatically sets the launch configuration. OpenMP has some clauses that when applied to a GPU target it should control the grid geometry to some extent — `num_teams` and `num_threads`. However, GCC ignores the clauses in practice, and Clang does not always respect them. Nonetheless, selecting an optimal grid configuration is a problem that is not addressed in this work, and the grid selection is delegated to the compiler.

Consequently, the problem is how to find which configuration GCC, Clang or any other compiler uses for each workshare region. Accessing internal compiler information is not an option because this work concentrates on source-to-source techniques and one of our goals is to be compiler independent.

The compiler's CLI options to dump information regarding target offloading are very limited in both GCC and Clang; it was not possible to find a way to tell the compiler to report the selected grid geometry. The explored solutions include inspecting optimisation reports, using the verbose mode, saving intermediate files and logs. Nevertheless, none contains the selected grid geometry information needed.

An alternative assessed approach is inspecting the resulting binary file. Launching the kernel on the GPU implies invoking the runtime or driver APIs, which are dynamically linked. Therefore, disassembling the binary should expose the library calls.

Consider the CUDA vector addition example in Listing 6.6. The kernel function takes three parameters as input, the addresses of each array. Moreover, it is launched using $10240/512 = 20$ thread blocks, each with 512 threads.

```
1   const short N = 10240;
2
3   __global__ void vec_add(const int *dev_a, const int *dev_b, int *dev_c) {
4      // global thread id
5      unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
6      if (tid < N)
7         dev_c[tid] = dev_a[tid] + dev_b[tid];
8   }
9
10  int main(void) {
```

```
11      // Allocate device and host memory
12      ...
13      // Perform any data transfering
14      ...
15      // Call GPU kernel
16      vec_add<<<(N) / 512, 512>>>(dev_a, dev_b, dev_c);
17      // Bring the results from the GPU to Host and release memory
18      ...
19  }
```

Listing 6.6: Vector addition in CUDA using $10240/512 = 20$ thread blocks and 512 threads per block.

The kernel is launched using a *execution configuration* [12] syntax `vec_add<<<(N) / 512, 512>>>`. However, the compiler transforms it into a runtime or driver API call. The prototype for the runtime API is presented in Listing 6.7, and the driver API's equivalent is shown in Listing 6.8. Both function prototypes take the grid and thread block dimensions as parameters, the information needed for the analytical model.

```
1  cudaLaunchKernel(
2     const void* func,
3     dim3 gridDim,
4     dim3 blockDim,
5     void** args,
6     size_t sharedMem,
7     cudaStream_t stream
8  )
```

Listing 6.7: Runtime API function prototype for launching kernels

```
1  cuLaunchKernel(
2     CUfunction f,
3     unsigned int gridDimX,
4     unsigned int gridDimY,
5     unsigned int gridDimZ,
6     unsigned int blockDimX,
7     unsigned int blockDimY,
8     unsigned int blockDimZ,
9     unsigned int sharedMemBytes,
10    CUstream hStream,
11    void** kernelParams,
12    void** extra
13 )
```

Listing 6.8: Driver API function prototype for launching kernels

After compiling the application with NVIDIA's compiler, `nvcc`, the Linux utility *objdump* is used to display information from object files, including the disassembled machine instructions. The command used is shown in Figure 6.6. The option `-d` enables disassembling, `-C` is just

for demangle the symbol names to user-level names, and `-M` is used to customise the assembly language to Intel syntax.

```
objdump -d -C -M intel <object file>
```

Figure 6.6: Command used to disassemble the application binary file, the result of compiling the vector addition in Listing 6.6.

Listing 6.9 partially lists the disassembled instructions for the `main` function, showing some preparation ahead of the kernel invocation. For instance, there are calls to `dim3`, used to construct an object representing the grid or thread block dimensions. The function prototype is `dim3(unsigned int, unsigned int, unsigned int)`. Each parameter corresponds to an axis in a three-dimensional coordinate system, `(x,y,z)`. Such representation is intuitive when the kernels work on 2D or 3D problems. Due to the function call conventions [39], before each `dim3` constructor invocation, there is a sequence of `mov` instructions preparing the arguments list. In general, the order is right to left. For instance, the first call is `dim3(0x200, 0x1, 0x1)` and the second is `dim3(0x14, 0x1, 0x1)`, which is equivalent to `(512, 1, 1)` and `(20, 1, 1)`. It is precisely the information needed for the analytical model.

```
403c1a: mov ecx,0x1
403c1f: mov edx,0x1
403c24: mov esi,0x200
403c29: mov rdi,rax
403c2c: call 403f82 <dim3::dim3(unsigned int, unsigned int, unsigned int)>
403c31: lea rax,[rbp-0x10]
403c35: mov ecx,0x1
403c3a: mov edx,0x1
403c3f: mov esi,0x14
403c44: mov rdi,rax
403c47: call 403f82 <dim3::dim3(unsigned int, unsigned int, unsigned int)>
403c4c: mov rax,QWORD PTR [rbp-0x1c]
```

Listing 6.9: Partial list of disassembled instructions corresponding to the `main` function in Listing 6.6, demonstrating the initialisation of `dim3` structures that define the grid geometry.

However, rather than using the *execution configuration* syntax, `<<<...>>>`, the programmer can instantiate the `dim3` manually and use them for launching the kernel. The consequence is that the order in which `dim3` appear in the disassembled code is not necessarily the grid followed by the thread blocks, as shown in Listing 6.9.

```
403e02: mov rcx,QWORD PTR [rbp-0x38]
403e06: mov r8d,DWORD PTR [rbp-0x30]
403e0a: mov rdx,QWORD PTR [rbp-0x2c]
403e0e: mov eax,DWORD PTR [rbp-0x24]
403e11: push rdi
403e12: push rsi
403e13: mov rsi,rdx // 3rd parameter, thread block dimension
403e16: mov edx,eax // 2nd parameter, grid dimension
```

```
9   403e18: mov edi,0x403e5a // 1st parameter, address of function to be called
10  403e1d: call 403f2e <cudaError cudaLaunchKernel<char>(char const*, dim3, dim3,
        void**, unsigned long, CUstream_st*)>
```

Listing 6.10: Disassembled instructions corresponding calling the `cudaLaunchKernel` runtime API to launch the kernel

A solution could be finding calls to `cudaLaunchKernel` or `cuLaunchKernel` and performing a data-flow analysis to find what data is passed through the parameters corresponding to execution configuration. Listing 6.10 shows the call to `cudaLaunchKernel` at address `403e1d`. In the example, it would be necessary to trace the registers `rdx` and `eax` and memory addresses in a bottom-up order, reaching the `dim3` instances shown in Listing 6.9. Although it seems a possible approach, it can be laborious, especially because function call conventions vary with OS, target architecture and compiler [39]. Moreover, although the disassembling approach can be a solution when compiling CUDA, it remains to assess if it could work in the OpenMP offloading context with GCC and Clang.

A similar experiment is conducted with both compilers, GCC and Clang. Rather than using disassembling, one can use the `-S` flag to output the assembly code. Starting with GCC, the assembly code reveals a call to `GOMP_target_ext`, a routine from `libgomp` — the OpenMP runtime library used by GCC. It is responsible for creating the threads team on the target accelerator and launching the kernel. However, it is not clear if the number of teams and threads per team are calculated before calling the routine or if it is calculated internally in the `libgomp`. But, it seems it is calculated internally after inspecting the library implementation lightly. In Clang, the observations are similar. The generated assembly only contains dynamic calls to OpenMP libraries. Thus, not only the grid geometry seems to be calculated internally, but any call to NVIDIA APIs also happens in the OpenMP libraries.

In conclusion, despite the explored options to automatically and statically find both grid and block dimensions, it was impossible to find a feasible one, especially in the OpenMP context. It would be necessary to further investigate the compiler's OpenMP libraries and understand how they determine the grid geometry. This problem is not addressed and is left for future work. Concerning the experimental evaluations, the information is collected by profiling the kernels with `nvprof`.

6.7 Hardware resources utilization

Each kernel requires different finite resources, such as registers, shared memory or constant cache. As discussed in Section 5.5, resource usage may limit the number of thread blocks and warps simultaneously active in each SM. In order to calculate the number of active warps, it is necessary to know how many registers and the amount of shared memory each thread or thread block requires.

In the CUDA programming model, the programmer manages the shared memory usage. However, in the OpenMP context, to the best of own knowledge, there is no way to tell GCC or Clang where and how shared memory should be used. Nevertheless, according to profiling data, both

compilers try to take advantage of it. Regarding registers, it depends solely on the compiler, which is responsible for the register allocation and avoids spilling as much as possible. It also depends on the target architecture because different compute capability architectures have specific limits on registers per thread and register file sizes. In conclusion, in the OpenMP context, shared memory and registers usage depends on how the backend generates the code for the GPU. The question is how to get that information to feed the analytical models.

With the official NVIDIA toolchain, the information can be accessed after the compiling workflow. The compiler `nvcc` has an option `-res-usage` that dumps code generation statistics. In fact, it is a shortcut for PTX assembler, `ptxas`, and the linker `nvlink`. Both have a verbose mode that reports the statistics. A report example after compiling the CUDA 2MM kernel from the Polybench [42] suite is shown in Listing 6.11. For each kernel, the assembler and linker displays a summary line, e.g. "Used 20 registers, 368 bytes cmem[0]". `cmem` is a short for constant memory. The report omits unused resources. For instance, if shared memory were used, the summary lines would state `<number> bytes smem`, where `smem` stands for shared memory.

```
ptxas info : 0 bytes gmem
ptxas info : Compiling entry function '_Z11mm2_kernel2iiiiffPfS_S_' for 'sm_61'
ptxas info : Function properties for _Z11mm2_kernel2iiiiffPfS_S_
0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info : Used 20 registers, 368 bytes cmem[0]
ptxas info : Compiling entry function '_Z11mm2_kernel1iiiiffPfS_S_' for 'sm_61'
ptxas info : Function properties for _Z11mm2_kernel1iiiiffPfS_S_
0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info : Used 20 registers, 368 bytes cmem[0]
```

Listing 6.11: Report generated compiling the CUDA version of 2MM for a Compute Capability 6.1 architecture

The next step is to investigate whether Clang or GCC have options to enable dumping similar reports. In the case of Clang, it is possible to use the `-verbose` flag, which then is passed to the PTX assembler. It also has the `-Xcuda-ptxas` to pass additional options to the assembler. In Clang, this is possible because their approach generates machine code for the selected GPU architecture (besides embedding PTX code in the final binary for JIT compiling if necessary). As previously hinted in the API traces (see Listing 6.5), GCC only generates the machine code at runtime. At compile time, it only generates the PTX code. However, registers in the PTX are just virtual placeholders and allocation is performed when PTX is compiled to machine code. Setting the environment variable `GOMP_DEBUG` when the application is launched, GCC's OpenMP runtime dumps information about JIT and the assembler statistics, as shown in Listing 6.12. Nonetheless, this implies launching the application, which does not fit our goal for a fully static approach.

```
$ GOMP_DEBUG=1 <application>
...
Linking
Link complete: 0.000000ms
```

```
Link log warning : Stack size for entry function 'mm2_OMP$_omp_fn$0' cannot be
    statically determined
info : 192 bytes gmem, 701 bytes cmem[3]
info : Function properties for 'mm2_OMP$_omp_fn$0':
info : used 62 registers, 0 stack, 392 bytes smem, 344 bytes cmem[0], 40 bytes
    cmem[2], 0 bytes lmem
```

Listing 6.12: GCC's LIBGOMP dumping assembler statistics in debug mode when launching the application

Machine code generation is delegated to the NVIDIA toolchain. Clang and GCC just generate PTX code. Thus, another possible approach could be asking the compiler to output the PTX pseudo-assembly and compile it with the `ptxas` assembler, passing the necessary parameters for printing resource usage statistics. When compiling for CPU targets, compilers support a `-S` flag for dumping the assembly code to a file. Using the flag in an OpenMP offloading application does not seem to work as it is not possible to find any PTX code. No other options seem available for that purpose. In contrast, the `nvcc` compiler from NVIDIA offers the `-ptx` flag.

Despite our efforts, collecting grid geometry and resource usages is not automated in our work. Engineering this specific problem is left for future work and information is collected via profiling for evaluation purposes.

6.8 Summary

This chapter presented the necessary modifications to the original analytical model presented by Kim et al. [47] and discussed different approaches to collect the input parameters for the model.

The first modification in the model is for suiting our methodology. It added an OpenMP factor due to the work distribution mechanism among the threads. The following modifications addressed some identified limitations. Firstly, it addressed the warp-level parallelism in the Streaming Multiprocessors, enabling multiple warps to execute in parallel. Secondly, it discussed the memory coalescing problem, showing that a binary classification for memory instructions does not suit modern GPUs. Instead, the number of transactions depends solely on the access stride and word size. Transaction size depends on the access granularity of the memory units in the memory hierarchy. Thirdly, Kim et al. use constant latency costs for all instructions, not adequate for modern GPUs as some instructions take a handful of cycles and others hundreds of cycles [48, 22]. Therefore, it is essential to distinguish the operations and their respective cost.

The remainder sections discussed collecting the different architectural and application properties needed for the analytical model. Due to the OpenMP offloading context, we found some obstacles to determining resource usage (e.g., number of registers) and the grid geometry selected by the compiler. Collecting the metrics automatically is left for future work.

Chapter 7

Automatic hotspot detection and acceleration

This chapter presents our framework flow to process sequential C code and output a modified version with parallelised or offloaded hotspots. The first section describes the considered code regions as potential hotspots and our approach to determine if they are parallelisable. Next, our conservative strategy for parallelisation is presented. The subsequent section addresses the AST-based analysis to characterise hotspots. The last sections explain how OpenMP pragmas are added and related issues, as well as the analytical models integration for target selection.

7.1 Overview

Figure 7.1 illustrates the framework complete flow. It runs on top of Clava [25] for AST analysis and code transformations. The input is a sequential C program.

The first step is searching for potential hotspots at the AST level. Similarly to existing literature (Section 3.6), we only consider loop nests for parallelism opportunities, which are usually time-consuming code regions. For all loops in the application, our framework validates if the loop is in canonical form, as required by OpenMP [10], and uses AutoPar [21] to determine if the loop is parallelisable.

In the next phase, the framework analyses all loop nests validated in the previous step and applies a parallelisation strategy to define the *workshare region*, which represents the code region that is parallel and distributed by participating threads. Through loop coalescing, the iteration space may be formed by one or more perfectly nested loops. The workshare inner loop's body represents the code each thread executes one or more times.

After defining the workshare regions, the framework collects the necessary metrics required by the analytical models to estimate the execution time. We propose characterising the workshares and counting operations at the AST level, as in relative performance context it may be possible

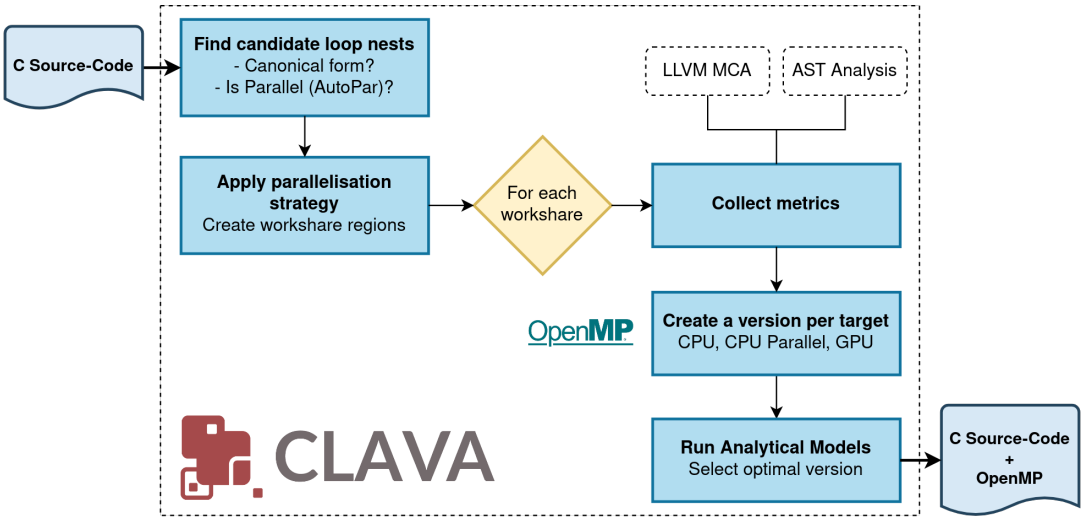


Figure 7.1: Framework flow

to relax the accuracy of the estimations without compromising the target selection optimality. Additionally, an LLVM MCA based approach is implemented for the CPU analytical models, first proposed by Chikin et al. [30] and discussed in Section 4.4. The framework automatically integrates the LLVM MCA tool for collecting cycle estimations and experimental results (Chapter 8) compares both approaches.

After collecting the metrics, we create three versions of the workshare region, one for each target: *(a)* CPU Sequential (same as original), *(b)* CPU Parallel, and *(c)* GPU. Each version is defined as a function containing the transformed code with the necessary OpenMP pragmas.

Finally, the framework evaluates the analytical models to determine the optimal target. The original sequential hotspot code is replaced with a call to the function that implements the code for the selected target.

The following sections address each phase in greater detail.

7.2 Finding candidate loop nests

Our approach considers any loop nest a potential hotspot that may benefit from parallelisation or offloading to the GPU. However, there are two pre-processing stages for filtering the loops. Firstly, the loop must be compliant with the OpenMP standard rules. Otherwise, the resulting application may be incorrect. Secondly, the loop has to be safe for parallelisation, i.e., must not have loop carried dependencies.

7.2.1 Motivation to enforce OpenMP rules

The OpenMP pragmas can only be inserted in loops that respect a canonical form. Inserting the parallelisation pragmas in a non-compliant loop may have two side effects:

- Compilers that enforce OpenMP rules will reject the framework's generated source code.
- The compiler accepts the incorrect code. Consequently, the application functionality will be incorrect.

Given our aim to create a tool that automatically parallelises applications, ensuring the generated application is correct is paramount.

7.2.2 Canonical loops in OpenMP

This section describes the canonical loops according to the OpenMP specification [10] and our approach to validate loops through static AST analysis. In its general form, a canonical loop has the structure presented in Listing 7.1.

```
1    for (init-expr; test-expr; incr-expr) structured-block
```

Listing 7.1: General form of canonical loops [10].

OpenMP specification states the loop header's expressions must have several properties. The following enumeration highlights some of the rules:

- The `init-expr` declares a variable `var` which is the loop control variable. In C, the variable's data type must be an integer or a pointer.
- The `test-expr` is a binary relational expression where one of the operands is `var` and the other is a loop-invariant expression, i.e., an expression whose value does not change throughout loop iterations. The control variable, `var`, cannot be present on both operands. Furthermore, it supports parent loops control variables: `var < outer-var` or `var < outer-var * a` are valid.
- The `incr-expr` updates the control variable, `var`. The stepping must be loop invariant as well. The supported arithmetic is limited to simple addition and subtractions. The expression `var = var + 4` is valid, but `var = var * 2` is not.
- The loop must be a structured-block, a compound of executable statements with a single entry at the top and single exit at the bottom. In other words, `return`, `goto` and `break` statements are not allowed.

Validating the initialisation and incremental expressions is straightforward at the AST level. In addition the framework ensures that all expressions operate in the same variable, inferred as the loop control variable.

Concerning the test expression, evaluating if an expression is loop-invariant is a well-known topic in compiler optimisations, allowing to move statements away from the loop to reduce overall computing cost. Our approach is limited. Let *Expr* be the second operand in the relational expression. For all variables in the expression *Expr*, we assert they are not modified in the loop's

body. In other words, their value is constant throughout the loop. If *Expr* has function calls, we assume the expression is not loop-invariant. Note that our approach can be improved with a more thorough analysis [63].

7.2.3 Determining if a loop can be parallelised

To determine which loops can be parallelised, we use the AutoPar [21] library integrated into Clava [25]. AutoPar relies on static analysis to determine if loops are safe to parallelise.

To determine if loops can be parallelised, AutoPar performs dependency analysis. Loops are considered safe for parallelisation when:

- Lack true dependencies;
- Have false dependencies that can be solved with the correct variable scoping (thread private variables);
- Have true dependencies that are reduction operations.

AutoPar can modify the source code automatically and insert OpenMP pragmas. The parallelisation is done on all parallelisable loops as AutoPar does not analyse the profitability of parallelisation. Furthermore, AutoPar does not perform loop-level transformations (e.g. tiling, interchange) to improve performance. Concerning the OpenMP pragmas, it adds the combined construct `omp parallel for` and inserts data-sharing clauses such as `private` and `firstprivate` to break false dependencies. Moreover, AutoPar can detect reduction operations on scalar and array variables, and inserts the appropriate `reduction` clause specifying the reduction operator and accumulator variable.

AutoPar is an important basis for our work to determine which loops can be parallelised and the necessary variable scoping to break dependencies.

7.3 Parallelisation strategy

Section 7.2 described our approach to find loops that are compliant with OpenMP specification and can be parallelised. As AutoPar [21] parallelises all loops without evaluating the profitability, we propose a conservative parallelisation strategy for the CPU and GPU, which is the topic for this Section.

Multiple parallelisable loops may belong to the same loop nest. Ideally, our framework should search for the optimal nest level for parallelisation, but searching for optimal solutions is left for future work. Instead, our approach is conservative and always selects the outermost parallelisable loop for parallelisation. Moreover, it tries to coalesce loops [66] to increase the parallel iteration space. Let $loop_{outer}$ be the outermost loop selected for parallelisation, then our approach coalesces all perfectly nested and parallelisable loops with $loop_{outer}$.

Listing 7.2 shows a loop nest of 3 perfectly nested loops. Assume that all loops are safe for parallelisation. Our approach selects the loop `i` for parallelisation — $loop_{outer}$. Considering that

`NI` is 1024, there is enough parallel work to distribute in a CPU physical cores, which usually range from 4 to 128 cores. Therefore, loop coalescing is not particularly beneficial for CPUs. However, modern GPUs feature thousands of streaming processors. Given that loops `j` and `k` are also parallelisable and are perfectly nested with each other, our approach coalesces the three loops to create a larger parallel iteration space ($NI \times NJ \times NK$) that makes better usage of GPUs highly-parallel architectures. Besides, GPUs constantly pre-empt threads to overlap memory accesses and synchronisation barriers with computation. Hence, having more parallel work units than the number of streaming processors is crucial to efficiently use the GPU.

```
for(int i=0; i<NI; i++) {
   for(int j=0; j<NJ; j++) {
      for(int k=0; k<NK; k++) {
         // body
      }
   }
}
```

Listing 7.2: Three perfectly nested loops, before loop coalescing

Listing 7.3 shows a possible implementation of coalescing the three loops into a single space of $NI \times NJ \times NK$ iterations.

```
for(int _i=0; _i < NI*NJ*NK; _i++) {
   int i = _i / NJ*NK; // note: integer division
   int j = (_i % NJ*NK) / NK;
   int k = (_i % NJ*NK) % NK;
   // body
}
```

Listing 7.3: Resulting loop after loop coalescing

We use the OpenMP `collapse` clause. For instance, inserting the clause `collapse(3)` in the loop `i` in Listing 7.2, instructs the compiler to coalesce the three loops.

7.4 Collecting metrics for analytical models

The previous section presented our parallelisation strategy, defining the loop nests for parallelisation or offloading. Our framework uses analytical models to guide target selection. Analytical models need the characteristics of the loop nest to estimate its execution time for a given target device. This section discusses our approach for collecting the necessary metrics, namely the number of executed computations and memory operations. As operations may be part of a loop, the total number of operations depends on the loop trip count. Determining loop trip counts with AST analysis is addressed first.

7.4.1 Estimating loop trip count

Precise trip count bounding is not always possible due to numerous reasons. Since data is generally unknown at compile-time, it is impossible to assess a precise trip count when it depends on data. Another problem is when the control variable depends on parent loops in the nest, resulting in non-rectangular or zero-trip loops, further complicating the estimation. Determining the number of iterations in a loop remains an active research problem [23].

Existing approaches in literature

Existing literature presents different approaches for trip count estimation. Some approaches use Integer Linear Programming (ILP) [31], formulating the problem as an equation system and defining precise bounds for variables. ILP solvers compute the problem and output the system solution — the number of loop iterations. However, ILP is only feasible if the constraints depend on constant values. For instance, some loops have a test expression such as `i < N` and the variable N is unknown at compile time.

An alternative approach is Abstract Execution [43]. The program is represented symbolically with abstract domains and values. The first step is analysing program flow to find scopes, regions of code that may repeat and all possible execution paths. The execution is simulated for each path. Variables are characterised by a domain representing the possible variables' values. Throughout the simulation, the domains are updated incrementally. Finally, variable domains are used to bound the loop trip count.

Other approaches build symbolic algebraic expressions and then use algebra solvers to calculate loop trip counts. It is advantageous when loop trip count depends on unknown parameters. This approach builds parametric expressions and can be evaluated at runtime [31, 44, 45].

Formulating the problem in OpenMP context

Given the loop constraints imposed by OpenMP, the problem for trip count estimation is simplified. Firstly, the loop must have single entry and exit points; therefore, statements such as `break` in the loop body are not allowed. Secondly, the OpenMP enforces loops in a canonical form, which simplifies the analysis at the AST level.

Let $initExpr$ be the initialisation expression for the control variable. The test expression is relational (see Section 7.2.2). Therefore, one operand is the control variable, and the other is a loop-invariant expression, $endExpr$. OpenMP also enforces that the stepping must be an arithmetic addition or subtraction. Multiplications, divisions and any other form are not allowed. Let $incr$ be an integer quantity for incrementing the control variable, such that at every loop iteration the control variable, $ctrlVar$, is updated as follows: $ctrlVar = ctrlVar + incr$.

Assuming a loop is in canonical form, the trip count can be stated as $\left\lfloor \frac{endExpr - initExpr}{incr} \right\rfloor$. The $initExpr$ and $endExpr$ expressions are not necessarily literal values. We use algebraic solvers to make the operations and simplify the result.

Non-retangular loops

OpenMP supports non-rectangular loops. The way we stated the trip count problem above is only valid for retangular loops. Listing 7.4 shows an example for non-retangular loops. Essentially, the inner's loop trip count depends on the outer loop. The inner loop iteration space decreases as the outer control variable increases, thereby the non-rectangular iteration pattern. In these cases, the loop trip count cannot be estimated independently. In other words, to compute the inner loop trip count, we need to consider the loop nest as a whole.

```
1  for(int i = 0; i < N; i++) {
2     for(int j = i; j < M; j++) {
3        // ...
4     }
5  }
```

Listing 7.4: Non-retangular loop illustration.

Healy et al. [44, 45] propose formulating the problem as a summation and then use the Bernoulli formula to solve the nested sums. For instance, the non-rectangular loop in Listing 7.4 can be formulated as $\sum_{i=0}^{N}\sum_{j=i}^{M} 1$. Since our approach uses algebraic solvers, we construct a sum expression that characterises the non-rectangular loop, and the solver outputs the number of iterations for the loop. For example, using Symja[1], the expression $Sum(1,\{i,0,N-1\},\{j,i,M-1\})$ gives the total trip count for the loop `j` in Listing 7.4.

Generalising the problem: our final approach

We generalise the non-rectangular approach to rectangular loops as well. However, it was possible to notice that the algebra solver is substantially slower when the lower and upper bounds are literal values. Consider a loop nest of 3 perfectly nested loops. Assume the control variables range is $[0,4095]$. The total iteration count for the innermost loop can be computed as $Sum(1,\{i,0,4095\},\{j,0,4095\},\{k,0,4095\})$. However, the solver takes a considerable amount of time as if it implements an $\mathcal{O}(n^3)$ algorithm to compute the sum's result. The same behaviour was noticed in other solvers besides Symja.

Our workaround is to prepare a sum expression with symbolic upper and lower bounds. The summation is evaluated resulting in a symbolic expression that is saved temporarily. Then the lower and upper bound variables are defined, and the summation result can be re-evaluated to output the constant result.

Listing 7.5 shows our implementation in Clava. Considering the same loop nest example, the solver computes the sum expression with symbolic bounds, as shown in line 6. The result is stored in a variable, `clavaSum`. Then we define the lower and upper bound variables with the information collected at the AST level. Finally, the sum result expression is re-evaluated, giving the trip count for the innermost loop, k.

[1] https://github.com/axkr/symja_android_library

```
// construct the Java object for evaluating expressions in Symja
const ExprEvaluatorJava = Java.type("org.matheclipse.core.eval.ExprEvaluator");
const solver = new ExprEvaluatorJava(true, 100);
// list of expressions to evaluate in Symja
const exprs = [
   "clavaSum=Simplify(Sum(1, {i, iLower, iUpper-1}, {j, jLower, jUpper-1}, {k,
       kLower, kUpper-1}))",
   "iLower=0",
   "iUpper=4096",
   "jLower=0",
   "jUpper=4096",
   "kLower=0",
   "kUpper=4096",
];
// execute each expression individually
for (e of exprs) solver.evaluate(e);
// re-evaluate the Sum's symbolic result
const tripCount = solver.evaluate("clavaSum");
```

Listing 7.5: Approach to use Symja in Clava, executing Sum expressions with symbolic lower and upper bounds to prevent performance hits

7.4.2 Counting operations at AST

Using OpenMP for parallelisation and offloading implies that the same code is compiled to different targets. We hypothesise that it should be possible to relax the accuracy of the estimations in a relative performance context without compromising the target selection optimality. Therefore, our analysis is done at AST rather than analysing machine code or other low-level representations, being compiler and target agnostic. This section details our approach.

Starting at the workshare region, the AST is traversed recursively. Each operation in the AST is characterised by a boolean that indicates if the operation type is integer or floating-point, the bit-width and the operator itself. Memory operations are also associated with a bit-width, and we distinguish read and write operations.

Our analysis makes some assumptions. For instance, it assumes scalar variables are stored in registers. As such, only array accesses are counted as memory operations. Operations in the loop headers and array subscripts are ignored. In some array subscripts, compilers are able to apply code motion such that the expressions are not re-computed every loop iteration. Instead, array indices are stored in registers and increment by some constant every iteration.

When traversing the AST, some nodes are handled specially. For instance, when function calls are found and the implementation is available, the analysis context switches to the function definition to count its operations. System or library calls are ignored. Our approach can be improved to count mathematical library calls, such as `pow` or `sqrt`. Operations in repeating code blocks, such as loops, the operations are counted in the loop's body and then multiplied by its trip count.

Counting operations at the AST level closely relates to language semantics. For instance, how does a float division in C translate to hardware instructions in GPUs? Some GPUs may not have native support to compute floating point divisions and use approximation algorithms instead. It

is up to the analytical models to map the counted operations at AST to native device instructions before using available latency and throughput tables to calculate the computational cost.

7.4.3 Getting LLVM MCA cycle estimations

An alternative approach for estimating elapsed cycles in the CPU is using the LLVM Machine Code Analyser (MCA), as Chikin et al. [30] proposed. Section 4.4 discussed the challenges found using the tool in a source-to-source approach and proposed a methodology that minimises the interference in compiler code generation, as well as a set of procedures to improve the cycle-cost estimation for the workshare region.

With the workshare regions defined, our LARA scripts insert the necessary assembly directives at the AST level to mark the regions of interest for LLVM MCA. Note these changes are temporary and only for analysis purposes. The resulting C code with the directives is written in a file and compiled with standard compilers (GCC or Clang). Since LLVM MCA expects assembly code, the `-S` flag is added to the compilation flags.

Our LARA script launches the `llvm-mca` command to analyse the resulting assembly file and generate a report. Then, our framework parses the information and calculates the total cost for one parallel iteration of a workshare region. All stages are done automatically using LARA scripts.

7.4.4 The workshare region structure

Listing 7.6 illustrates the internal structure for workshare regions. After the metric collection phase, every attribute is filled. Our framework can proceed with evaluating the analytical models for target selection and make the necessary code transformations to insert OpenMP pragmas.

```
1  class WorkshareRegion {
2     $outerLoopJp; // reference to the outermost loop AST node
3     $innerLoopJp; // reference to the innermost loop AST node. This loop's body
          is the workshare region. If it is the same as $outerLoopJp, loops were
          not coalesced
4     numParallelIters; // the size of the parallel iteration space
5     llvmMcaCycles; // estimated cycles for the workshare region using LLVM MCA
6     compOps = {
7        'flop-add-32': 5,
8        'flop-mul-64': 2,
9        'iop-add-32': 1
10    }; // number of computational operations counted at AST, starting at
          $innerLoopJp loop body. each operation is floating-point (flop) or
          integer (iop), has an operand type, and a bitwidth
11    memOps = {
12       'write-32': 2,
13       'write-64': 1,
14       'read-32' 2,
15       'read-64': 1
16    }; // number of memory operations organised per access type (read or write)
          and the word size in bits
```

```
17      ompInfo; // information about the OpenMP context defined by AutoPar. it
            specifies the data-sharing clauses, e.g. 'private', 'first-private'. it
            also has the collapsing depth, decided based in our parallelisation
            approach
18  }
```

Listing 7.6: The internal structure that represents workshare regions

7.5 Annotating code with OpenMP pragmas

Previous steps in our framework selected code regions for parallelisation and offloading, defining the workshare regions. This section presents our approach for inserting the OpenMP pragmas in the source code and addresses specific issues concerning offloading to the GPU.

Our approach iterates workshare regions by function. A function with one or more workshare regions is cloned, generating a new version per target. One version is an exact copy for running sequentially in the CPU. A second version is annotated with OpenMP pragmas to run all workshare regions in parallel in the CPU. A third version offloads workshare regions to the GPU. Listing 7.7 illustrates the three copies created for a function `mm2` with two workshare regions. The original function's code is replaced to integrate the analytical models and invoke one of the versions.

```
1   // version for CPU sequential
2   void __mm2_cpu_seq(float *A, float *B, float *C, float *D, float *E) { ... };
3   // version for CPU parallel
4   void __mm2_cpu_parallel(float *A, float *B, float *C, float *D, float *E) { ...
        };
5   // version for GPU offloading
6   void __mm2_gpu_parallel(float *A, float *B, float *C, float *D, float *E) { ...
        };
7   // original function
8   void mm2(float *A, float *B, float *C, float *D, float *E) {
9      // if GPU is optimal ...
10     __mm2_gpu_parallel(A, B, C, D, E);
11  }
```

Listing 7.7: Illustration of the different function declarations targetting different devices

Assuming all information is static, the analytical models can generate estimates during our framework analysis and the optimal assessed target is known. Instead of creating three versions, an alternative approach is making the necessary transformations in the original function. However, there is some motivation for creating the three versions. Firstly, one advantage of source-to-source techniques is allowing the user to modify the code further. Therefore, our approach makes the three different versions available. If our models guide non-optimal target selection, the user can easily correct the code by adjusting the function call. Secondly, our approach anticipates future extensions such as:

- Target-specific optimisations and tuning, justifying the need for different versions per target;

- Delay target selection decisions for runtime, when one or more parameters are unknown statically.

The remainder of this section discusses the insertion of OpenMP pragmas for running the workshare regions in parallel in the CPU or offloading to the GPU.

7.5.1 Parallel running in CPU

As explained before in Section 7.2.3, we use AutoPar [21] to determine if loops can be parallelised. We also use AutoPar's internal information, such as variable scoping, to insert the OpenMP pragmas on loops selected with our parallelisation strategy.

Listing 7.8 shows the parallelisation for the first workshare in the `2MM` kernel from UniBench [56].

```
1  int i, j, k;
2  #pragma omp parallel for collapse(2) default(shared) private(i, j, k)
       firstprivate(A, B)
3  for(i = 0; i < 4096; i++) {
4     for(j = 0; j < 4096; j++) {
5        C[i * 4096 + j] = 0.0;
6        for(k = 0; k < 4096; ++k) {
7           C[i * 4096 + j] += A[i * 4096 + k] * B[k * 4096 + j];
8        }
9     }
10 }
```

Listing 7.8: Partial `2MM` kernel [56] with the OpenMP constructs for shared-memory parallelism (N=4096)

7.5.2 Offloading to GPU

AutoPar [21] does not support the accelerator model introduced in recent OpenMP standards [10]. Although the basic pragmas used for shared-memory systems are also available when offloading to accelerators, additional directives are needed. For instance, code regions to be offloaded have to be marked explicitly for offloading and it is necessary to control data transferring between the host CPU and accelerators.

OpenMP constructs for offloading

To offload computation to accelerators, the OpenMP specification introduced the `target` directive. All parallel code within the target region is relocated to a given device.

In shared-memory systems, there is a single degree of parallelism. The work is distributed evenly by available CPUs and their cores. Some accelerators, such as GPUs, offer multiple degrees of parallelism. The workload is split into independent units of work that can be executed in any order in the Streaming Multiprocessors (SM) — the thread blocks. In turn, thread blocks are

organised in groups of threads (warps) that execute concurrently in the same SM. The OpenMP specification introduces the concept of *teams* to support multiple levels of parallelism. Threads within the team can synchronise and share memory. When teams are created, they have a master thread. Upon work distribution routines, the team's master thread launches additional threads in its team, and then the work initially assigned to the team is distributed among the threads. The exact interpretation of teams and the threads within is implementation-dependent and may vary from target to target. In the context of NVIDIA GPUs, a team resembles a thread block, and the number of threads in the team corresponds to the thread block size (which is expected to be multiple of 32, the warp size).

Listing 7.9 shows our automatic GPU parallelisation for the first workshare region in `2MM` from UniBench [56]. The first directive, `omp target`, is applied in the outermost loop; hence it is offloaded to the GPU. The `map` clauses determine the data environment. In other words, it defines data to be transferred to the GPU and back to the host. The array dimensions are explicitly set. Determining array boundaries and data mapping is discussed in further detail below. The second directive initialises the league of teams. For simplification, we use a composition of constructions, `teams distribute parallel for`, meaning the loop iterations are distributed among the teams and participating threads in each team. The remainder clauses are the same ones used in the CPU parallel version.

```
1  int i, j, k;
2  #pragma omp target map(to: A[0:16777216], B[0:16777216]) map(tofrom:
       C[0:16777216])
3  #pragma omp teams distribute parallel for collapse(2) default(shared)
       private(i, j, k) firstprivate(A, B)
4  for(i = 0; i < 4096; i++) {
5     for(j = 0; j < 4096; j++) {
6        C[i * 4096 + j] = 0.0;
7        for(k = 0; k < 4096; ++k) {
8           C[i * 4096 + j] += A[i * 4096 + k] * B[k * 4096 + j];
9        }
10    }
11 }
```

Listing 7.9: Partial `2MM` kernel [56] with the OpenMP constructs for target offloading (N=4096)

Transferring data between CPU and GPU

With offloading support in OpenMP, the memory model is extended to handle different devices. When an OpenMP program launches, a data environment is implicitly created for each device. When host CPU offloads an OpenMP region to the GPU, data in the hosts memory must be mapped to the device's internal memory, and data transferring might be necessary.

In OpenMP, there are six memory mapping types, with the more relevant for this work being:

- `to`: Move data from the host to device.

- `from`: Allocate buffer on the device, uninitialized; at the end of computation, data is copied back to the host memory.
- `alloc`: Allocate buffer on device, uninitialized; no movement between host-device needed. Useful to allocate memory regions that are just auxiliary buffers for computation.
- `tofrom`: A combination of `to` and `from`; data is moved from host to device, and then from device to host.

There are different options for creating data mapping environments. Our approach appends the `map` clause to the `target` construct. The `map` clause defines a list of variables to be mapped to the target environment, according to the mapping types enumerated above.

Determine the data mapping type

The data mapping type is determined using AST analysis and a conservative approach. All array accesses in the workshare region are queried. Each array access is classified as a read or write operation. After iterating all arrays accesses in the workshare region, each array is classified as *(a)* read-only, *(b)* write-only, and *(c)* read-write. Read-only arrays are mapped with `to` and read-write with `tofrom`. With respect to write-only arrays, we are very conservative. There might be a chance where the GPU only writes in a sub-set of the memory region. Therefore, the original information must be preserved. Arrays classified as write-only are mapped with `tofrom`.

As our analysis is limited to a workshare region, it limits the opportunities for more efficient data management. With intra and inter-procedural analysis, out framework could detect that some arrays are temporary or find data-reuse opportunities to prevent redundant data transfer. Our work can benefit from existing literature for more efficient data management [59, 72].

Array boundaries

One additional issue concerning the data mapping is when arrays in the workshare region are accessed via pointers. Consequently, the memory region size is unknown to the compiler. In these cases, the programmer must specify the number of elements in the arrays to be mapped to device memory.

Our approach to overcome this problem is using AST analysis to bound the accessed array indexes. Since the array accesses happen inside loops of the workshare region, in general, array positions are determined by loop control variables and other constants. Our approach analyses the array subscript expression and finds all variables in it. For variables that are control variables, we can infer their lower and upper range by knowing the initialisation expression and the loop trip count. Any other variable is assumed to be constant, which is a significant limitation. It is also possible to have array accesses as subscripts, e.g., `A[B[i]]`, which we do not support. After bounding the variables accessed in the subscript, we can determine the array's lower and upper bounds, replacing the variables with their respective boundaries.

7.6 Target selection with analytical models

The final step in the framework flow is to estimate the best target for executing the workshare regions within a function. The analytical models discussed in Chapters 4-6 are implemented in LARA and can be evaluated throughout our framework analysis. Assuming the collected metrics in previous steps are constant values, the analytical models output a time estimation in seconds. The target with the lowest anticipated running time is selected as the optimal target.

The original function body is replaced with a call for a function with the modified code for the selected target. Consider the example in Listing 7.7, presented earlier. If the optimal target is the GPU, then the original function's body, `mm2`, is replaced with a call to `__mm2_gpu_parallel`.

In case the analytical model estimations cannot be evaluated due to missing parameters, the original function is not modified. In future work, support for runtime decisions may be added.

7.7 Summary

This chapter presented our overall framework flow. First, it addressed our methodology for finding the hotspots using AST analysis with Clava. We consider loops as candidates if they are in canonical form and safe for parallelisation. Next, we presented our conservative parallelisation strategy. We traverse the AST, and the first valid loop is selected for parallelisation. Besides, we coalesce all perfectly nested loops to increase the degree of parallelism, which is advantageous for GPUs. The subsequent sections presented our approach for collecting metrics, considering the two approaches: AST analysis and LLVM MCA. Concerning code transformations, we create one function version per target. We discussed our approach for determining array boundaries and the data mapping necessary for transferring data between CPU and GPU. Finally, the original function code is replaced with a call for the optimal target as estimated by the analytical models.

Chapter 8

Experimental results

This chapter presents the experimental results. The first section describes the platforms used for benchmarking and the environment setup. The following sections present and analyse the results.

First, we compare different methodologies when using the LLVM MCA to show the importance of recursively processing loops and multiply its cost by a precise trip count (Section 4.4). The following section evaluates the CPU analytical models, showing estimate errors for our AST and LLVM MCA approaches for computational cost calculation. Afterwards, we compare the original analytical model (Chapter 5) the extended version with our contributions (Chapter 6). Next, we consider all models and assess the quality for guiding target selection for the various hotspots in benchmark suite kernels. The following section compares our automatic parallelisation strategy to manually transformed OpenMP kernels (Chapter 7). Finally, we evaluate the overall framework showing the achieved speedups with respect to the original sequential version.

8.1 Experimental methodology

8.1.1 Platforms

The experiments and measurements were performed in one platform whose characteristics are presented in Table 8.1, also showing additional details concerning the software used and its versions.

8.1.2 Platform configuration

Prior launching the kernels on the platforms to measure time execution, some preparing steps are performed to ensure an environment that suits the work under analysis and prevent external factors to affect results interpretation.

Property		Antarex
CPU	Name	2x Intel Xeon E5-2630 v3
	Base Clock	2.4 GHz
	Physical/Logical Cores	8/16 (per physical CPU or NUMA node)
	RAM	128 GB (8x16GB, 64 GB per NUMA node) DDR3 @ 1866 MT/s
GPU	Name	NVIDIA GeForce GTX 1070 (Pascal)
	Max Clock	2012 MHz
	RAM	8 GB GDDR5
Operating System		Ubuntu 20.04 (kernel 5.11.0)
Compilers	Clang	10.0.0
	GCC	9.3
NVIDIA Toolchain	CUDA version	11.3
	Driver version	465.19.01

Table 8.1: Platform characterisation used for evaluation.

Frequency scaling

CPUs, GPUs and other components that assemble the system operate on varying power modes and varying frequencies. For instance, the i5-8300H in the Laptop can operate between 1.2 GHz and 3.2 GHz. The NVIDIA 1070 clock frequencies ranges between 139 MHz and 2012 MHz. The clock frequencies are an important parameter for the analytical model. Besides, the analytical models used do not model frequency scaling. Since the CPU and GPU run at varying frequencies, which value should be used as an input parameter? Fortunately, it is possible to configure the hardware and operating system to ensure a more steady and predictable frequency.

Starting with the CPU, the following measures are taken:

Disable turbo boost. In general, CPUs technical sheets indicate a *base clock* and one or more *max turbo frequency*. According to Intel, the Turbo Boost feature has different power states and is designed to last for a short period of time. However, some vendors unlock the turbo frequency limits, and as long as there is enough cooling, the turbo boost can be sustained for long periods of time. The base clock, by definition, is the normal operating frequency for the CPU. Therefore, vendors design the system such that the CPU can operate on the base clock indefinitely without constraints on power or thermals. For our experiments, the turbo boost is disabled on the operating system such that the clock frequency does not exceed the base clock (Table 8.1). In Linux operating systems with the `intel_pstate`[1] driver, turbo boost can be disabled by writing `1` to `/sys/devices/system/cpu/intel_pstate/no_turbo`.

CPU Governor. Disabling the turbo boost makes the average clock frequency tend to the base clock. However, when the application is launched for the first time, it is likely the CPU is operating at lower clock frequencies and might cause interference for kernels that conclude

[1] https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt

in less than a second. One possible solution is to configure the CPU governor. According to Linux Kernel documentation [2], setting the governor to `performance` sets frequency to the highest value statically. Since the turbo boost is disabled, all CPU cores run at base frequency at all times.

For NVIDIA GPUs similar techniques are explored. However, the GPUs used for evaluation are consumer-grade GPUs and the possibility to adjust clock speeds and power limits is restricted. GPUs tailored for data-center and HPC are more flexible. For instance, the NVIDIA System Management Interface (`nvidia-smi`) has two options: `-lock-gpu-clocks` and `-applications--clocks`. They allow setting static clocks for memory and graphics cores. Unfortunately, it is not supported in the NVIDIA 1050 and NVIDIA 1070 (Table 8.1).

The best workaround we found is setting performance modes, which are pre-defined operating profiles that control GPU clock ranges, memory clocks and memory transfer clocks (i.e., the PCI-e interface to communicate with host CPU). Running `nvidia-settings -query GPUPerfModes` lists the available modes and respective parameter values. However, neither mode sets a static graphics clock, it just sets lower and upper bounds for frequency scaling. On the other hand, transferring and memory clocks can be fixed. Running `nvidia-settings -assign GPUPowerMizerMode=1` enables the maximum performance mode.

Hyper-threading (HT) and Simultaneously Multi-Threading (SMT)

Modern CPUs use different approaches to maximize the CPU resources usage, such has instruction level paralelism and out of the order execution. Nonetheless, due to data dependencies, unavailable execution units or mispredictions, full utilization of the CPU resources is not always achieved. CPUs that support HT and SMT allow two threads to execute in each core and it is an additional step to improve resource utilization and increase overall throughput. With SMT/HT, each physical core has two maintain architectural states (in registers) for each thread. The number of active threads simultaneously in each core corresponds to the number of logical cores and it is typically two per core. It is important to understand that with HT/SMT the threads are not executed in parallel but rather concurrently. Besides, Core resources have to be shared — queues, schedulers, execution units, register file, etc [54]. Some are statically partition while others are dynamically allocated. Therefore, HT or SMT with two logical cores per physical core does not double performance. In some scenarios it might not increase performance at all [53].

The analytical models implemented in our work do not model HT/SMT. Therefore, the kernels are launched with a number of threads that corresponds to the physical cores rather than logical cores.

Thread Affinity

Another step to mitigate operating system "noise" is setting the thread affinity. Operating System (OS) schedulers are constantly seeking to increase system performance and the OS scheduler tries

[2] https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

to even the load across the CPU cores by moving runnable threads to different cores as needed. Thread scheduling has introduces overheads. Firstly, there is the context switching overhead. Secondly, when a thread moves from one Core to another, cached data in the initial Core's private caches (e.g. L1) is lost.

Thread affinity consists in pinning threads to a specific CPU Core. Thus, the threads start executing on a given Core and are never move to other Cores through execution. Given that the analytical models do not model the costs associated to Operating System scheduling, our experimental setup pins one thread per physical core. With OpenMP, the affinity can be configured using the environment variable `OMP_PROC_BIND` [10].

NUMA

Although the Antarex system is a NUMA platform (Table 8.1), modelling the intrinsics of NUMA is out of scope in our work. Therefore, the kernels execute in a single node and memory allocation is restricted to the node's local memory.

Summary

The following is a summary of the setup used:

- Disable turbo boost in the CPU, limiting the maximum clock to base clock.
- Set the kernel governor to `performance`. It increases the CPU power state, and sets a static frequency. Assuming the turbo boost is disabled, the frequency matches the base clock.
- GPU power state set to maximum performance, ensuring stable memory clocks and memory transference rate.
- In Antarex, applications run in a single NUMA node and only access local memory.
- OpenMP:
  - Set thread affinity, pinning each thread to a different physical core.
  - Number of threads on CPU matches physical cores to mitigate HT or SMT effects.
  - Configure the thread scheduling policy to static, `OMP_SCHEDULE=static`. Dynamic and other policies are not studied in our work.

8.1.3 Benchmarks

The UniBench [56] suite is used to evaluate the analytical model estimations precision and the automatic parallelisation mechanism. UniBench is an adaptation of the original Polyhedral Benchmark Suite (Polybench) [75] to support OpenMP offloading and contains 14 kernels in total. Each kernel has two versions. One is the original kernel from Polybench without any modification,

targeting sequential execution on the CPU. A second variant is manually annotated with OpenMP pragmas and can execute in parallel on the CPU or offload to the GPU.

UniBench has predefined problem sizes which are used for all kernels and define array sizes and respective computation workload:

- `RUN_TEST`: 1100
- `RUN_BENCHMARK`: 9600

Our evaluation uses custom problem sizes as $N = 9600$ is too large for the experimental platforms, with most kernels crashing due lack of memory in the GPU. Besides, it is desirable to have more testing sizes to evaluate the analytical models behaviour as problem sizes increase gradually. Therefore, four categories are defined and the values are inspired in the original Polybench.

- `EXTRA_SMALL`: 512
- `SMALL`: 1024
- `MEDIUM`: 2048
- `LARGE`: 4096

It is also important that the suite is representative, i.e., the optimal target varies for different kernels and problem sizes. Table 8.2 summarises the number of times each target is optimal to run a given kernel. It is possible to see that the optimal target varies with the problem size. Except for $N = 512$, there are always kernels benefiting from all targets under analysis.

Size	CPU Sequencial	CPU Parallel	GPU
512	4	10	0
1024	5	4	5
2048	2	4	8
4096	2	4	8

Table 8.2: Number of times each target is optimal to run a given kernel in Antarex compiled with `-O3`, considering the 14 kernels of the UniBench benchmark suite.

8.1.4 Measuring kernels execution time

UniBench [56] kernels are compiled with Clang or GCC with optimisation flags that are specified in the results discussion. Execution times are measured using `rtclock`, which measures the time to invoke the kernel function. Time spent in memory allocation is not accounted. However, when workshare regions are offloaded to the GPU, data movements contribute to the kernel's execution time, as intended. To prevent code elimination, the matrices that are the kernel's computation output are reduced (summing all values) and the result is printed to `stderr`. Therefore, the compiler will not eliminate the kernel function call.

Depending on the device target, the UniBench kernels are compiled with different flags that enable/disable OpenMP, and enable/disable offloading. Table 8.3 summarises the flags for Clang and GCC. For running the kernel sequentially in the CPU, no additional flags are needed because by default OpenMP is disabled. To enable parallel running in the CPU, it is necessary to add the `-fopenmp` flag. When OpenMP is enabled and GCC finds a `target` environment, by default it compiles the OpenMP parallel regions for CPU and GPU (and any other available targets). Therefore, the offload is disabled explicitly. Finally, for enabling offloading to the GPU, the target is explicitly set. The GPUs used for evaluation are Compute Capability 6.1. Unfortunately, GCC only supports old compute capabilities, hence the target is set to `nvptx-none=-misa=sm_35`. The stack protection for GPU offloading is disabled to prevent the PTX assembler of crashing. Lastly, in GCC the optimisation flag has to be explicitly managed. In other words, if the code is compiled with `-O3`, the flag is not passed to the backend that generates the PTX code. The workaround is setting the optimisation flag via the `-foffload` parameter.

Compiler	Target	Additional flags
GCC	CPU Parallel	`-fopenmp -foffload=disable`
	GPU	`-fopenmp -foffload=nvptx-none=-misa=sm_35 -foffload="{OPT_FLAG}" -fcf-protection=none -fno-stack-protector`
Clang	CPU Parallel	`-fopenmp`
	GPU	`-fopenmp -fopenmp-targets=nvptx64-nvidia--cuda -Xopenmp-target -march=sm_61`

Table 8.3: Compiler flags used to enable OpenMP and control offloading to GPU

Each kernel application is launched 5 times. We noticed that for simpler kernels that complete in less than a second, the times were not very consistent, despite our efforts to configure the platforms for a stabler environment. Therefore, 25 samples are collected for those kernels. The times are processed and outliers are removed using interquartile range. The average is used to represent the kernel's execution time and it is compared to the analytical model estimations.

8.2 Analysing LLVM MCA techniques

This section evaluates the different techniques explored for using `llvm-mca` as discussed in Section 4.4. The techniques can be summarised as follows:

Workshare Region (A). It is the baseline approach where `llvm-mca` is used to estimate the cost of one *work item*, i.e. one iteration of the parallel code region, without any particular treatment. The cost is multiplied by the number of work-items assigned to each thread to estimate the number of computational cycles per thread. In this approach, the assembly directives are inserted at the beginning and ending of the scope that represents the parallel region.

Handling Loops (B). `llvm-mca` does not simulate control flow structures, such as a loop. Our proposed approach splits the parallel code region in *basic blocks* (see Section 4.4) to isolate sequential code from repeating code sequences. Each *basic block* is analysed individually with `llvm-mca`. For basic blocks associated to loops, their cost is multiplied by loop's trip count. Trip counts are estimated in the source-code, resulting in a hybrid approach.

Loop Optimisations (C). Loop optimisations impact the assembly code sequence which is analysed with `llvm-mca` and may impact the number of times the sequence is repeated. For instance, if the loop is unrolled by a factor of two, the assembly sequence analysed by `llvm-mca` represents two loop iterations. Consequently, the trip count estimated in the source-code has to be adjusted accordingly. The approach B is extended to incorporate the loop optimisations performed by the compiler and adjust the loop trip count based on vectorisation, interleaving and unrolling factors.

Assembly directives inside loop's body (Asm_{in}). To use the `llvm-mca`, assembly directives are inserted at the source-code to define the basic blocks. However, the approach for inserting the directives can affect the compilation outcome. For instance, inserting the directives in the loop body creates an inter-iteration dependency that prevents loop optimisations.

Assembly directives wrapping the loop (Asm_{wrap}). Assembly directives are added around the loop as an attempt to minimize interferences in the compilation.

Note that Asm_{in} and Asm_{wrap} are strategies for inserting the assembly directives inside the parallel code region. Therefore, they are always paired to the main strategies — B and C.

Figure 8.1 shows the absolute errors for CPU estimations using different `llvm-mca` configurations. Approach A simply extracts the parallel code region without further processing, therefore is not paired to any other technique. Approach B is tested with both techniques for inserting the assembly directives — Asm_{in} and Asm_{wrap}. The improvement C is only added to (B, Asm_{wrap}), because with Asm_{in} the directives are inserted in the loop body and block compiler optimisations.

As expected, the approach A results in higher error magnitudes and more optimistic estimations suggested by the negative absolute errors. Most kernels from UniBench [56] have nested loops within the parallel region. Therefore, B reduces error magnitudes in all cases, with the exception of `FDTD-2D`, where parallel regions are composed of a single compute statement — which justifies the constant error for all tests.

With the configuration (B, Asm_{wrap}), the assembly directives are inserted around the loop, which enables loop optimisations, and therefore the analysed assembly is closer to what is executed when UniBench [56] is compiled. However, the trip count is not adjusted accordingly. For the majority of the kernels, specially long-running ones, the error drops significantly compared to (B, Asm_{in}). In spite of that, it is not possible to conclude the approach is better. The kernels `2MM`, `3MM`, `COVAR`, `GEMM`, `GRAMSCHM` are memory bound due to cache misses. With (B, Asm_{wrap}), loops are unrolled and in some cases vectorised, increasing the cost of the assembly sequence

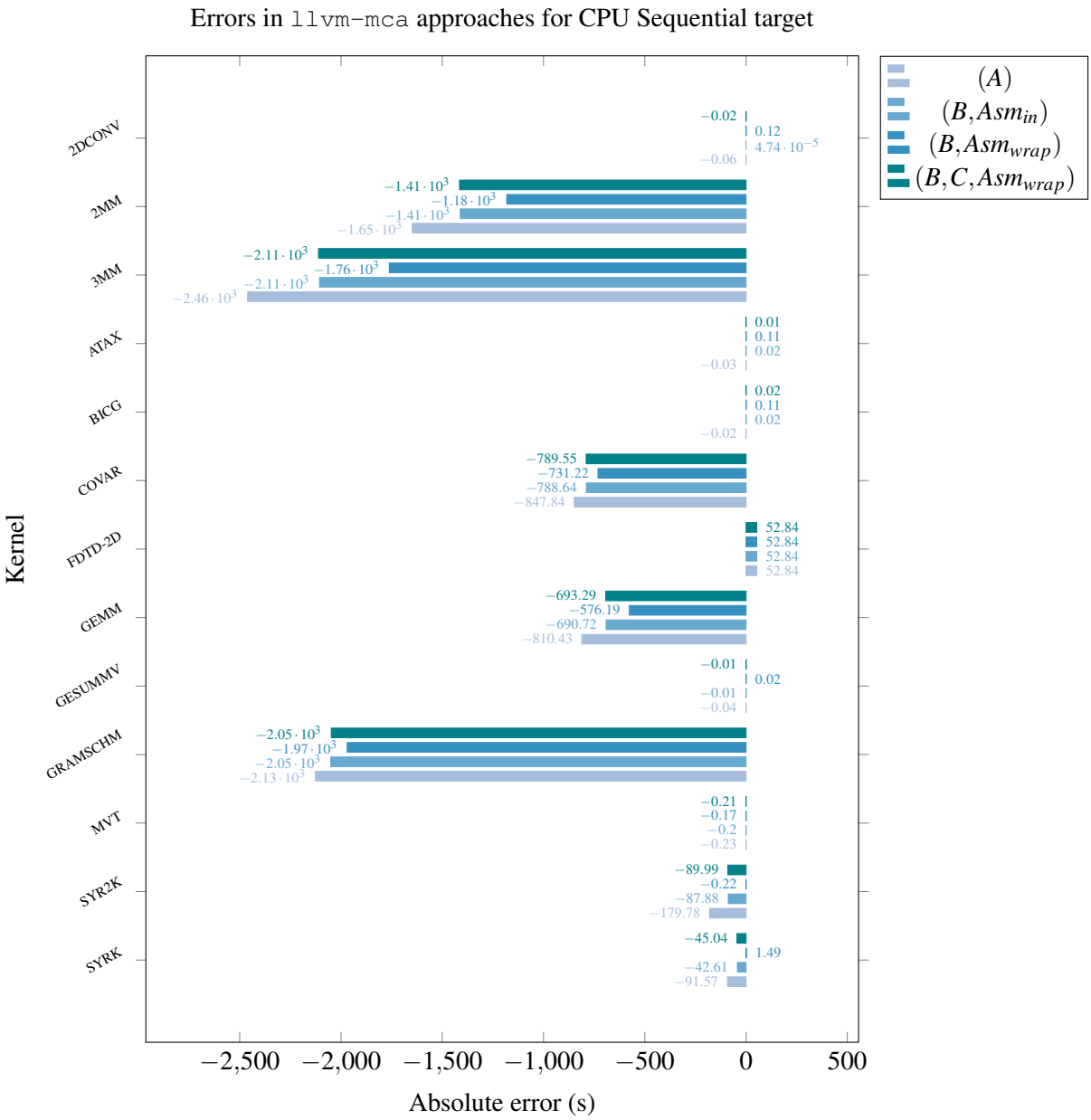


Figure 8.1: Comparison of absolute error in `llvm-mca` approaches for CPU Sequential. Measurements obtained with Clang, `-O3`, and problem size set to $N = 4096$.

analysed by `llvm-mca`. That additional cost shadows the CPU stalls due cache misses, hence the smaller error magnitude. For faster kernels like `2DCONV`, `ATAX` and `BICG`, (B, Asm_{wrap}) is worst than (B, Asm_{in}), as expected. In these kernels the memory effects are less relevant and are better for comparing (B, Asm_{wrap}) and (B, Asm_{in}).

For the remainder of this chapter, experiments using `llvm-mca` approach apply the methodology (B, C, Asm_{wrap}), which is expected to be the correct approach, although the results do not support the expectation. The experiment can be improved by comparing the `llvm-mca` estimations to elapsed cycles in the CPU back-end. Using execution times includes elapsed time in memory accesses which is not modelled, complicating the analysis of the results. Furthermore, the `llvm-mca` estimations may suffer from our source-to-source approach as it can affect the compiler outcome (see Section 4.4). Therefore, it might be important to include estimations obtained manually, i.e., adding the `llvm-mca` directives at assembly level to not affect the compilation and then run `llvm-mca` to get the estimations.

8.3 Comparing AST and LLVM MCA approaches

In our work, two approaches are explored to estimate the computational cost of a code region, a parameter used internally in the CPU analytical model:*(a)* counting operations at AST level, mapping to x86 instructions and accumulating the average cycle-cost per instruction, and *(b)* a hybrid approach using `llvm-mca` that simulates the sequence of assembly instructions and outputs a cost, alongside an AST analysis to estimate how many times a code region simulated in `llvm-mca` repeats. This section compares the estimative errors using both approaches for sequential and parallel workloads in the UniBench suite [56].

Figure 8.2 shows the absolute errors in CPU estimations using the two approaches. The absolute errors are the difference between the estimated time necessary to execute the kernel against real-world measurements in Antarex. The kernels are compiled with Clang using the `-O3` optimisation flag.

The first conclusion is that the absolute errors achieved in the distinct approaches are very similar. It is a very interesting remark given that `llvm-mca` has the ability to analyse the assembly code, which is fully transformed and optimised by the compiler. Aside from the problem of the problem of inserted assembly directives that may affect the compilation outcome, the assembly code analysed with `llvm-mca` is expected to match what is executed on the hardware. Besides, `llvm-mca` simulates the pipelining stages of superscalar processors, modelling port usage, instruction dependencies, and other fine-grained details. In contrast, our AST approach counts the operations at AST level and has no information concerning how the code is transformed by the compiler. Moreover, our approach assumes there are no dependencies. Finally, with respect to instruction execution costs, it simply accumulates the average cost per instruction, assuming maximum throughput. Therefore, the expectation was `llvm-mca` estimations would have much smaller error magnitudes, as our AST approach is quite optimistic. Despite the error proximity, in general, the magnitude of the errors in the `llvm-mca` approach is smaller as expected.

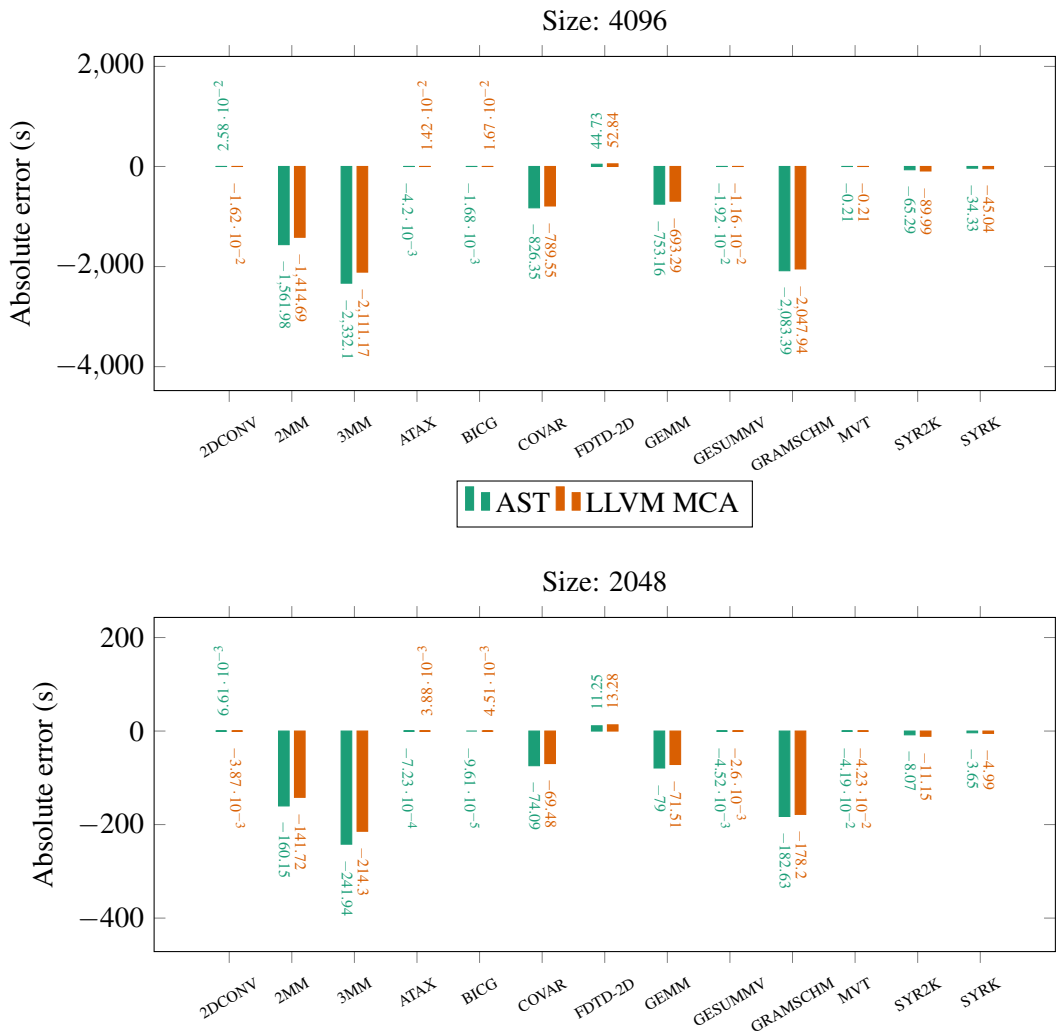


Figure 8.2: Absolute error comparison for `llvm-mca` and AST approaches in CPU Sequential estimations, using Clang and `-O3`.

Table 8.4 summarises the average estimation errors for varying problem sizes. The conclusions are consistent regardless of the problem size.

Problem sizes	Average absolute error w.r.t execution times (s)			
	Sequential		Parallel	
	AST	`llvm-mca`	AST	`llvm-mca`
1024	-2.4	-1.8	-0.27	-0.2
2048	-56.79	-52.16	-7.31	-6.73
4096	-585.54	-549.15	-80.5	-75.95

Table 8.4: Average absolute error on different problem sizes.

The same experiment is performed for parallel execution, as shown in Figure A.3. The observations are the same as for for sequential execution in Figure 8.2.

Another important advantage of `llvm-mca` is it naturally models the compiler's configuration because it analyses the assembly code, the compiler's backend output. In other words, the `llvm-mca` estimations should be consistent regardless of the compiler flags used. Our AST approach is agnostic to the compiler configuration and the estimations are always constant, regardless of the optimisation flags. In the future, our approach could be extended to estimate classic compiler optimisations and adapt the analysis accordingly.

To evaluate the resilience of `llvm-mca` and AST approaches to different optimisation flags, analytical model estimations are generated for three compiling configurations:

- `-O3` optimisation flag, used in earlier experiments.
- `-O2` with vectorisation disabled. In this test, our `llvm-mca` approach is adjusted accordingly to only update loop trip counts based on unrolling, which is still enabled in `-O2`.
- `-O0`, disabling all optimisations. Our `llvm-mca` approach no longer adjusts loop trip counts.

Although `-O0` might be rarely used in production, it is included for two reasons:

- UniBench [56] kernels's execution times in Antarex do not change significantly from `-O3` to `-O2` with no vectorisation, as shown in Figure A.1. The reason is vectorisation is mostly applied in short running kernels or in workshare regions with small footprint relative to the total kernel execution time (see Table A.1). Therefore, `-O0` is added as it exposes a larger gap in execution times compared to optimised versions.
- With `-O0` compiler optimisations are disabled and allows to evaluate the effectiveness of the two approaches on different compiler setups.

Since the kernels execution times vary depending on the optimisation flags, for this evaluation we use the relative error between the analytical model time estimations and the measurements collected in Antarex. Figure 8.3 shows the relative errors trend for the different optimisation flags. On top, the error evaluation for the AST approach and below for `llvm-mca`.

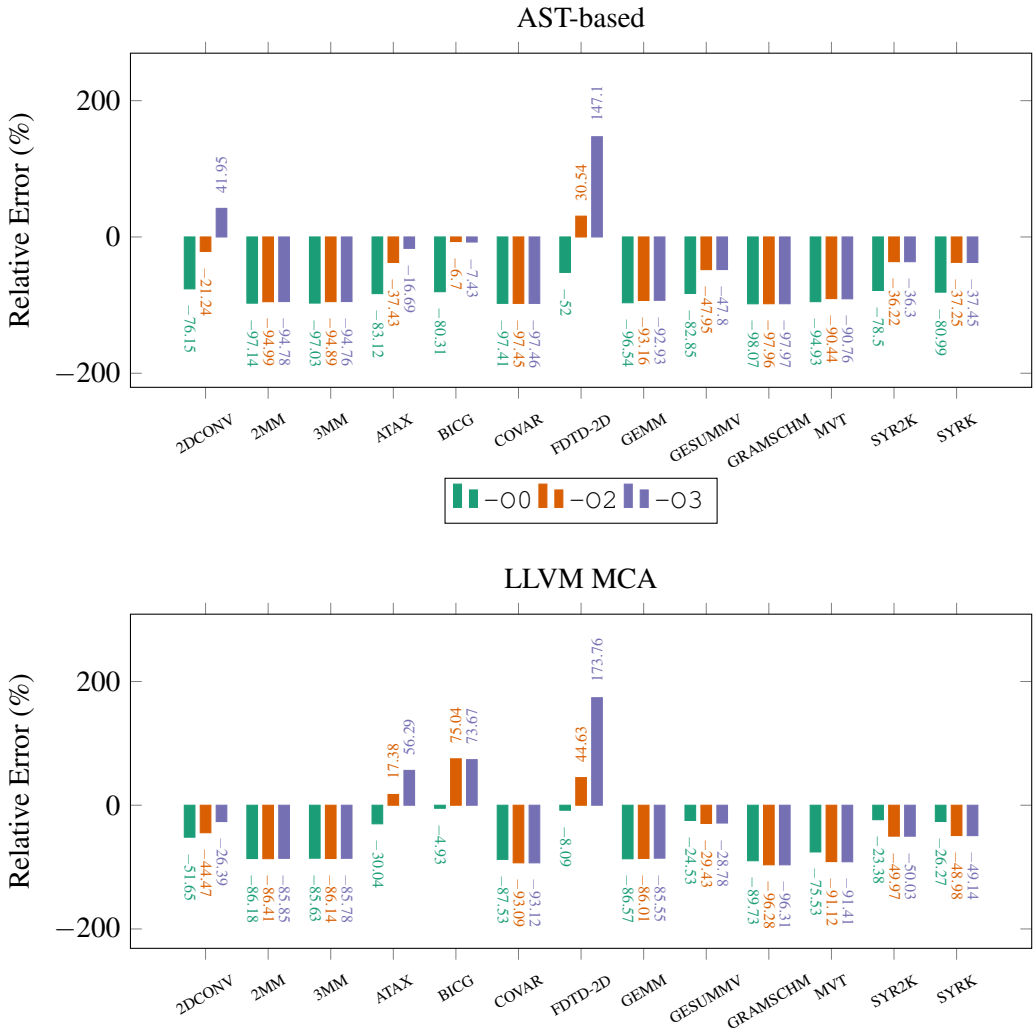


Figure 8.3: Relative error variation with different optimisation flags for $N = 4096$. Kernels are compiled with Clang. In `-O2` the vectorisation is disabled with `-fno-slp-vectorize` and `-fno-vectorize`.

Concerning the AST approach, the error magnitudes are larger with `-O0`. Considering our approach, that ignores operations in array subscripts and loop headers, it is expected to approximate more the `-O2` and `-O3` optimisation levels, rather than `-O0`.

Error values for some kernels present fluctuation. Interestingly, the fluctuations trends are similar in both approaches. The kernel `2DCONV` has a particularity. It is optimised with vectorisation, but its trip count is not multiple of the vector width (Table A.1). Therefore, Clang creates a second version of the loop which is unrolled by a factor of two to complete the reminder iterations. Given $N = 4096$, the loops in `2DCONV` iterate 4094 times. 4092 iterations are performed in the vectorised loop version, and the reminder 2 iterations in the unrolled loop version. The assembly analysed by `llvm-mca` will include both versions and therefore should increase the error. Interestingly, in the `llvm-mca` approach the error is higher in `-O0` than in `-O3`, when it should be the other way around. `2DCONV` has high arithmetic and memory intensity (see Table A.1). Each loop iteration does 9 memory accesses. Even though the access stride is sequential with opportunity for caching, given the small execution time, perhaps the error fluctuation is due memory side effects.

The kernels `ATAX`, `BICG` and `MVT` have similarities: equal arithmetic and memory operations intensity, and the only optimisation applied to loops is unrolling (Table A.1). However, `MVT` and `ATAX` have one workshare region that is not cache oblivious as one matrix is accessed in column-major order. Nonetheless, `MVT` has stabler errors in AST and `llvm-mca` approaches compared to the other two kernels. One additional difference between `ATAX` and `BICG` is that both have a loop that initialises one matrix to zero before the OpenMP context. With `-O2` and `-O3`, the loop is eliminated and replaced with a system call to `memset`[3]. Therefore, that loop is ignored in our experimental setup. However, it may contribute to the large error jumps from `-O0` to `-O2`. In `BICG` the error is stable from `-O2` to `-O3` in both approaches, while `ATAX` still fluctuates from `-O2` to `-O3`. In that case, given that `ATAX` is not cache oblivious, the error may be due to memory accesses. As `2DCONV`, `ATAX`, `BICG` and `MVT` execute in a fraction of a second, some of the error deviations may just be noise.

Concerning long-running kernels, the results are quite consistent with some exceptions. All kernels are memory bound due to high cache miss rates. The exceptions are `SYRK` and `SYR2K` and present a small variation from `-O0` to `-O2` in both approaches.

The kernel `FDTD-2D` also presents considerable variations in both approaches. This particular kernel has an outermost loop with 500 iterations that aggregates four workshare regions, a unique detail compared to other kernels where workshare regions are outer loops. The results indicate a possible limitation in our experimental setup or performance modelling approach, but the issue is not identified for the time being.

8.4 Evaluating GPU analytical model

This section evaluates the GPU analytical model estimations. Our work implemented the original analytical model proposed by Kim et al. [47] and an improved version to address potential

[3] https://www.man7.org/linux/man-pages/man3/memset.3.html

limitations regarding (Chapter 6):

- Warp-level parallelism inside the Streaming Multiprocessors
- Use reciprocal throughput for instructions costs, rather than a fixed cost of 4 cycles
- Adapt the memory coalescing modelling for modern GPU architectures

Figure 8.4 compares the baseline and our improved version in UniBench. The chart shows the absolute errors for $N = 4096$, and for kernels compiled with `-O3` and `-O0`. Our implementation for the GPU analytical model relies solely in AST analysis. It has the same limitation as our approach for CPU time estimations: it is agnostic to compiler optimisations. Unfortunately, we could not find a similar tool to `llvm-mca` designed for NVIDIA GPUs that can analyse PTX or NVIDIA native assembly.

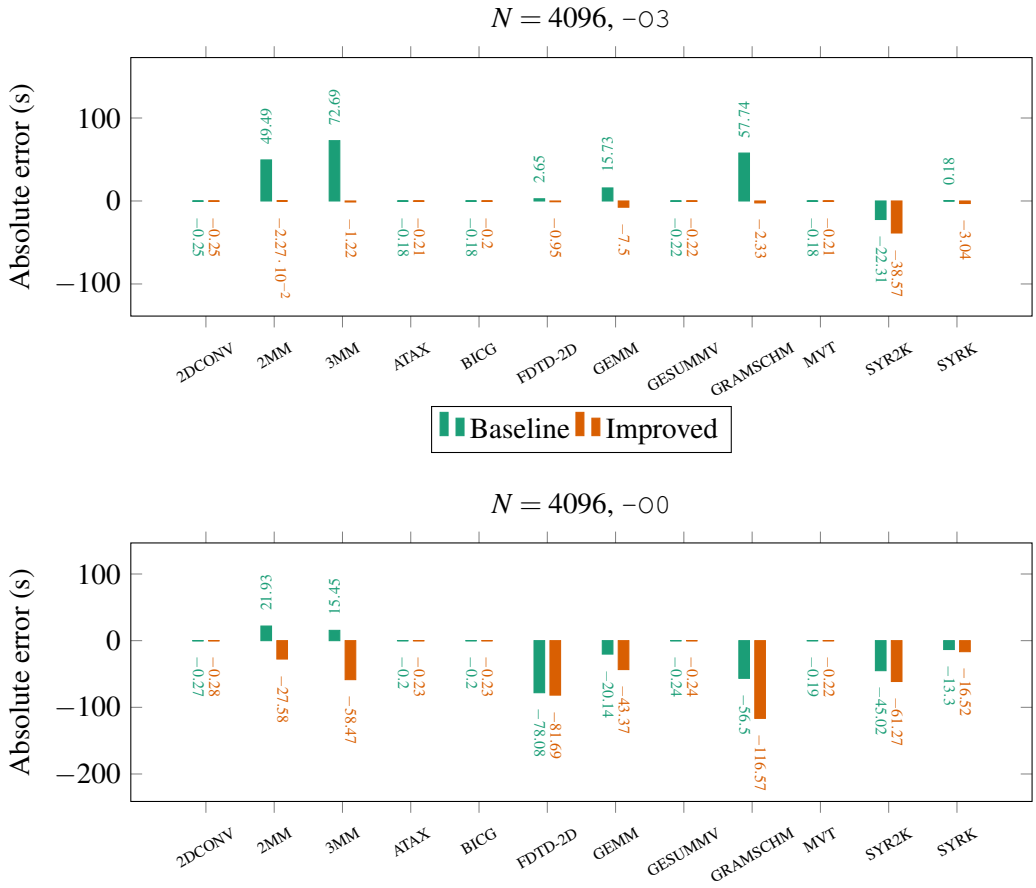


Figure 8.4: Comparing the baseline and improved GPU analytical models with $N = 4096$. On top, the results for `-O3`. The bottom has the results for `-O0`.

The results for `-O3` show that our approach is an improvement over the original analytical model [47], particularly in more computational intensive kernels. The exception is `SYRK`, where the error magnitude in our approach is higher.

Regarding the `-O0` results, our implementation is worse. A similar observation was done for the CPU analytical models, where the AST approach was better on `-O3` results than `-O0`. A possible justification can be the fact that using instruction throughput is too optimistic and approximates the slower execution times of `-O3`, hence the smaller error in `-O3`.

8.5 Assess offloading decisions

The previous sections evaluated the CPU and GPU analytical models individually, comparing different scenarios to assess the estimations precision. However, our work uses the analytical models simultaneously to guide offloading decisions for hotspots in the application code. This section evaluates the offloading decisions driven by analytical models when working together.

Figure 8.5 shows the maximum speedup possible for the UniBench [56] kernels and the achievable speedups if the offloading decisions were driven from the analytical models. Two speedups are presented for the analytical models due to the AST and `llvm-mca` approaches used to estimate execution times in the CPU.

The speedup is relative to the baseline target, i.e., running the kernel in the CPU sequentially, as shown in Equation 8.2. The maximum or optimal speedup is determined from real-world measurements, selecting the target that executes the kernel faster (Equation 8.1), and then calculating the ratio relative to the baseline.

$$Optimal\_Target_t = min(CPU\_Seq_t, CPU\_Par_t, GPU_t) \tag{8.1}$$

$$Speedup = \begin{cases} \frac{Optimal\_Target_t}{CPU\_Seq_t}, & \text{if } Optimal\_Target_t \geq CPU\_Seq_t \\ -\frac{CPU\_Seq_t}{Optimal\_Target_t}, & \text{otherwise} \end{cases} \tag{8.2}$$

The analytical models are used to estimate the total time to execute the kernel in the CPU, CPU Parallel or GPU. The lowest estimated time determines the offloading target driven by the analytical models. Note that the achieved speedups in Figure 8.5 are calculated using the measured times rather than estimated times. In other words, if the analytical models guide offloading to the GPU, then $Optimal\_Target_t$ is the real measured execution time for the GPU. There are three different interpretations for the achieved speedup:

Optimal. The analytical models guide the optimal offloading decision if the achieved speedup matches the maximum speedup.

Improvement. The achieved speedup is smaller than the maximum, meaning the guided offloading decision is not the optimal decision, but still accelerated the kernel compared to the baseline, i.e., running the kernel sequentially in CPU.

Slowdown. The achieved speedup is negative. By definition, the speedup is a ratio and therefore it is a positive value. However, Equation 8.2 makes an adjustement for slowdowns,

converting it in a negative value for easier interpretation. For instance, a speedup of 0.05 is translated to −20 and it means the application is 20× slower than the baseline.

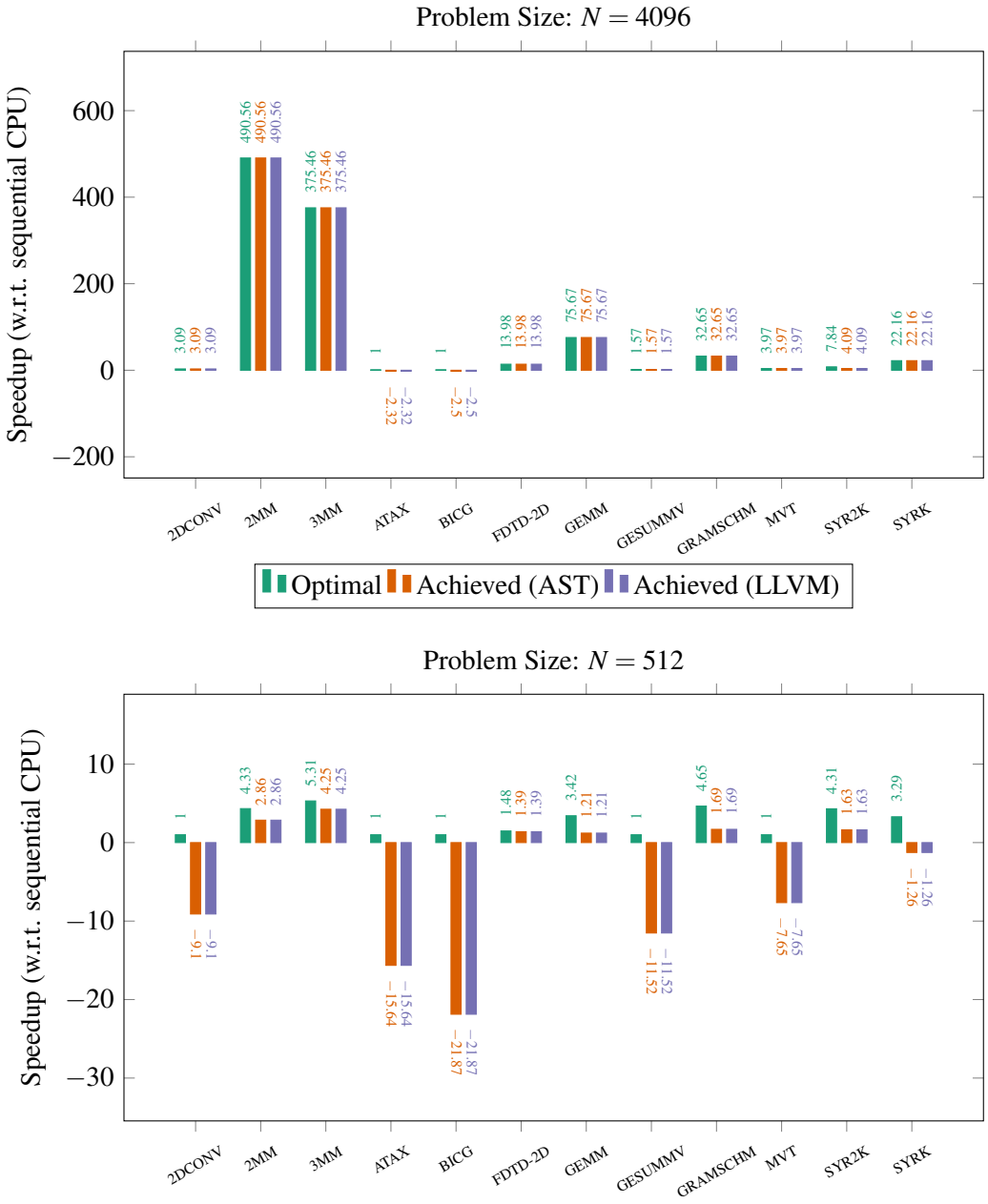


Figure 8.5: Speedups achieved driven by analytical model estimations.

Figure 8.5 compares the speedups for two problem sizes and using `-O3`. For $N = 4096$, except for `BICG` and `ATAX`, the analytical models guide offloading decisions that accelerate the kernel relative to the baseline. Besides, all offloading decisions are optimal, except for `SYR2K`. Regarding `BICG` and `ATAX`, the kernels finish in under a second. Despite the $\approx 2\times$ slowdown, the application is slower in the 10^{-2} seconds magnitude. The results for $N = 512$ are a different picture, with many kernels suffering slowdowns. Only 6 kernels are improved, but none has the optimal decision.

The remainder 6 kernels result in slowdowns. Despite the high slowdown values, the kernels in $N = 512$ complete in less than a second. Therefore, the slowdowns add $[10^{-1}, 10^{-3}]$ seconds to the optimal execution time.

The results in Figure 8.5 also show that both approaches (AST and `llvm-mca`) have the same efficiency. In fact, they always derive the same offloading decisions. It was somewhat expected since previous experiments shown that the absolute errors between the two approaches were always close to each other. For $N = 1024$, there are 5 optimal decisions, 2 improvements and 5 slowdowns, as shown in Table 8.5. The potential acceleration lost is also in the range $[10^{-1}, 10^{-2}]$ seconds. The offloading decision results for $N = 2048$ are the same as $N = 4096$.

The evaluation is repeated for `-O0` and summarised in Table 8.5. The parity between AST and `llvm-mca` in offloading decisions remains and there is a similar trend for more incorrect decisions on smaller computational kernels. For $N = 4096$, only `FDTD-2D` does not have an optimal offloading decision, but it is improved nonetheless.

Problem sizes	**-O0**			**-O3**		
	Optimal	Improve	Slowdown	Optimal	Improve	Slowdown
512	3	4	5	0	6	6
1024	7	3	2	5	2	5
2048	10	2	0	9	1	2
4096	11	1	0	9	1	2

Table 8.5: Offloading decisions classification summary. Equal results for AST and `llvm-mca` approaches.

In addition to the quality of the target selection decisions guided by the analytical models, it is important to evaluate their relative errors. If the relative errors for the three targets are very close to each other, then our methodology is more likely to guide good offloading decisions. However, if the relative errors are very scattered for all targets, it means our approach is not robust enough, despite the promising offloading results presented in Table 8.5. Figure 8.6 presents the standard deviation of relative errors on the three targets: CPU sequential, CPU parallel and GPU. A null standard deviation means the relative errors is the same for all targets. That is the ideal scenario, as the correct offloading decision is granted.

To conclude, the analytical models performed well overall in the UniBench kernels [56]. There is room for improvement, specially in short running kernels where sub-optimal decisions are quite common. Although the penalty is small and unnoticeable to a user, it occupies accelerators unnecessarily. One possible quick fix introducing a threshold when comparing the analytical model time estimations, rather than doing a strict comparison.

8.6 Evaluating automatic parallelisation

One central part of our work is to identify parallelisable code regions automatically. This section compares the automatic parallelisation against the manually annotated code in Unibench kernels

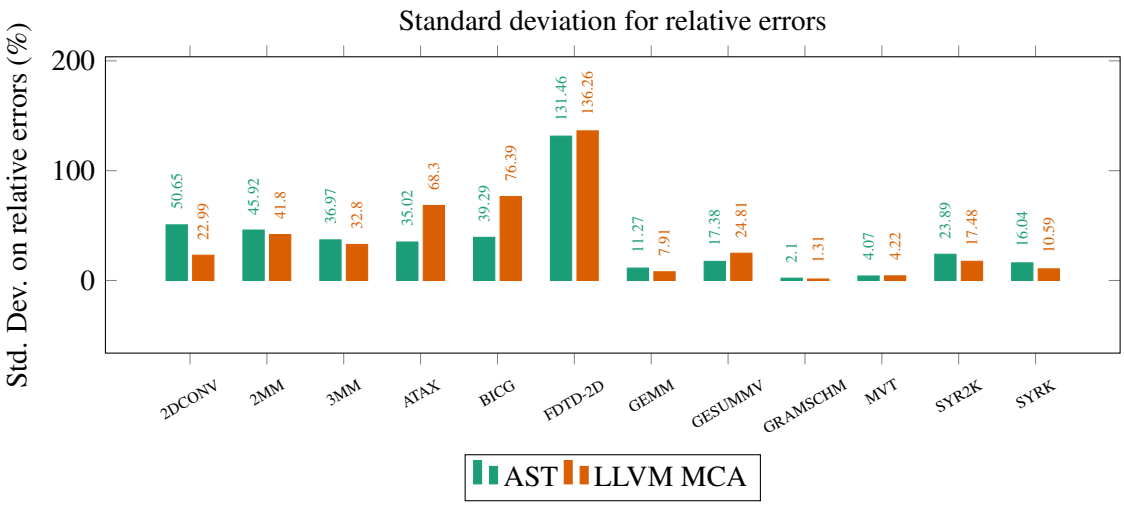


Figure 8.6: Standard deviation on the relative errors for the three targets. The chart shows the standard deviation for our AST and LLVM MCA approaches. Problem size set to $N = 4096$ and compiled with Clang and `-O3`. Execution times measured in Antarex.

[56].

Our experiment generates the modified code for CPU Parallel and GPU without any analytical models guidance. We measure the execution times achieved with our automatic parallelisation on each target and compare it with UniBench. This experiment demonstrates whether our conservative parallelisation and offloading strategy leave performance on the table compared to manually annotated code by experts.

Figure 8.7 and 8.8 compare the speedups achieved relative to baseline sequential execution on the CPU. Note that we do not transform the sequential code; hence the results are relative to the same source.

The first issue is that in some kernels, AutoPar [21] does not mark some loops as safe for parallelisation. In `ATAX`, only one innermost loop is marked for parallelisation. Compared to UniBench, the two outermost loops are annotated with OpenMP. The impact is substantial for GPU, as there is data transferring for every outermost loop iteration. The total amount of data transferred is unnecessarily large, transferring over 256 GB of data. Consequently, the execution time is 50 seconds slower w.r.t. UniBench. Kernel `BICG` suffers from similar issues, where the primary hotspot is not parallelised. However, the impact is more residual, being 45 ms slower. In `CORR`, the primary hotspot with $\mathcal{O}(n^3)$ complexity is not considered safe for parallelisation either.

Consequently, these three kernels have no speedup compared to the baseline or are much slower than UniBench versions. These kernels were successfully parallelised in the original AutoPar paper. The difference is that the AutoPar paper used the original Polybench benchmark suite, while our evaluation uses UniBench. Although the kernels are equivalent, they are written slightly different. For instance, kernel functions in UniBench use pointer parameters, while Polybench uses standard array syntax. Therefore, it is likely due to an AutoPar limitation in the code analysis.

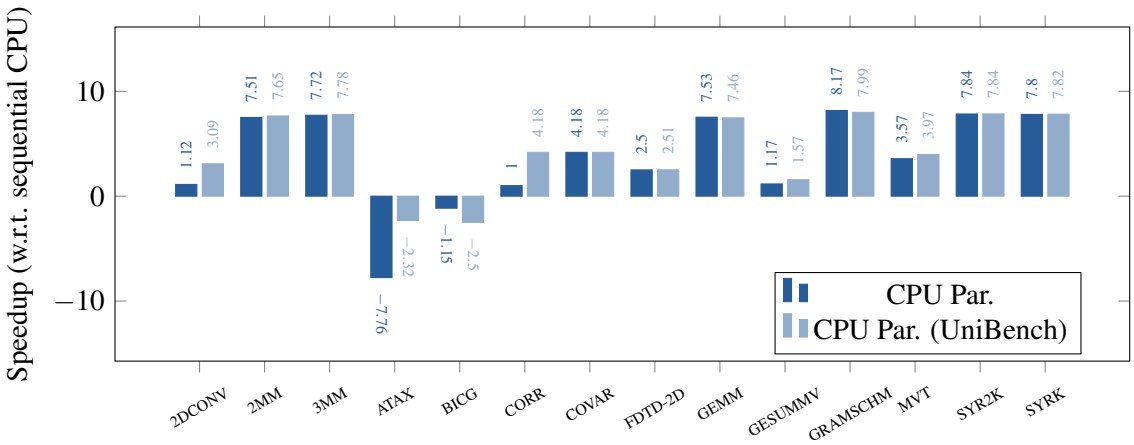


Figure 8.7: Speedups comparison of UniBench (CPU Parallel) and our automatic parallelisation for CPUs. Problem size set to $N = 4096$ and compiled with Clang and `-O3`. Execution times measured in Antarex.

Concerning parallelisation for CPU, Figure 8.7 shows similar speedup levels, the exception being the kernels mentioned above. `GRAMSCHM` is slightly improved compared to UniBench, reducing the execution time of 266 seconds by 5 seconds. The small gains are due to the parallelisation of three workshare regions compared to one in UniBench.

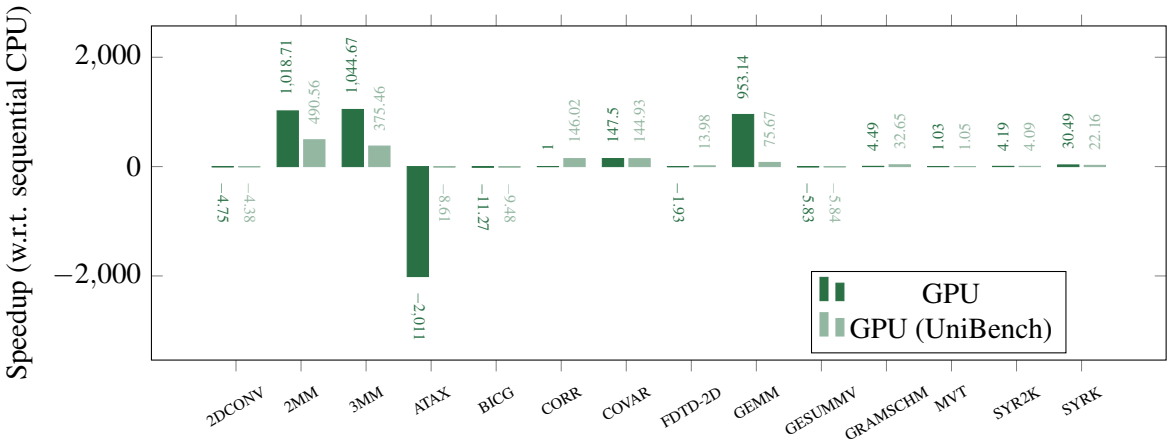


Figure 8.8: Speedups comparison of UniBench (GPU) and our automatic parallelisation for GPUs. Problem size set to $N = 4096$ and compiled with Clang and `-O3`. Execution times measured in Antarex.

Figure 8.8 reveals larger speedups in `2MM`, `3MM` and `GEMM`. Our advantage over UniBench is due to loop coalescing. It reduces execution time on GPU by 1.7, 4.2 and 9.9 seconds, respectively. `SYRK` execution time is also reduced by 1 second by offloading the first loop nest. Concerning `FDTD-2D`, although the offloaded code regions are the same as in UniBench, the lack of data mapping optimisation throws away any benefit in offloading. Kernel `GRAMSCHM` suffers from the same issue. In both `FDTD-2D` and `GRAMSCHM`, the workshare regions are nested in a loop that

runs sequentially on the CPU. Hence, the multiple unoptimised data transfers have more impact than in other kernels.

8.7 Framework overall

Throughout this chapter, previous sections evaluated the efficacy of the analytical models estimating execution times using our AST analysis approach and compared our automatic parallelisation to the manually annotated kernels in UniBench [56]. This section considers the complete framework. In other words, the sequential kernel versions of UniBench are automatically transformed by our framework, and the analytical models guide the target selection.

Figure 8.9 compares the speedups achieved. The bar *Achieved (auto)* represents the speedups achieved after running the transformed applications. The speedups are measured with respect to executing the kernels sequentially on the CPU. The bar *Optimal (auto)* shows the maximum speedup achievable with our parallelisation approach. As demonstrated before, in some kernels, speedups are limited due to problems in parallelisation safety checking. Besides, some kernels are affected due to unoptimised data transferring. Therefore, if the *Achieved (auto)* bar is smaller than *Optimal (auto)*, then the analytical models guided a non-optimal target selection. Finally, the third bar, *Optimal (UniBench)*, shows the maximum possible speedup with UniBench.

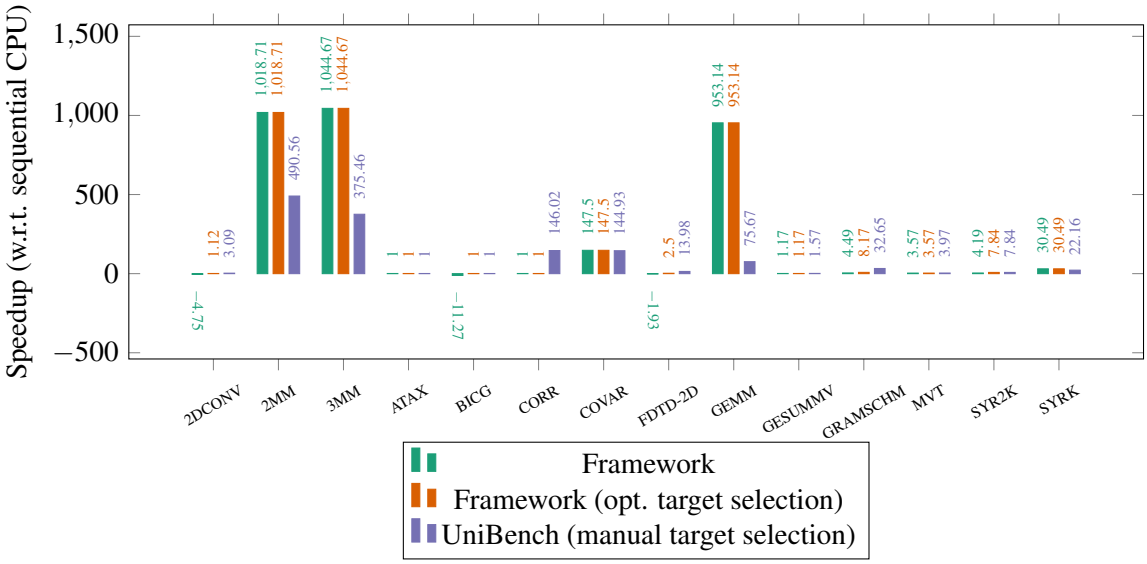


Figure 8.9: Speedups comparison between the framework automatic target selection, framework with optimal target selection and UniBench with optimal target selection. Problem size set to $N = 4096$ and compiled with Clang and `-O3`. Times measured in Antarex.

Overall, the results are encouraging. The maximum achievable speedup in UniBench kernels has a geometric mean of 18.2×, while the best case for our framework is 12.9×. As the analytical models made some non-optimal target selection, the achieved geometric mean speedup is 7.9×. From a time standpoint, the baseline for running UniBench kernels sequentially in the CPU takes 9E3 seconds. With our framework, it takes 1.44E3 seconds. However, UniBench transformed

code with manual optimal target selection completes in 1.3E2 seconds, one order of magnitude faster compared to our framework. The main culprit for the near 10× difference are `CORR` and `GRAMSCHM`. In `CORR`, the main hotspot is not offloaded, and the sequential running on the CPU lasts 848 seconds. Offloading to the GPU reduces the execution time by 5 seconds. In UniBench, offloading `GRAMSCHM` to the GPU reduces execution time to 65 seconds. The optimal decision considering our parallelisation strategy would be running parallel in the CPU, and takes 260 seconds, but the analytical models guide offloading to the GPU, which takes 474 seconds. These two kernels represent approximately 1.2E3 seconds (83%) of total execution time achieved with our framework, 1.44E3 seconds.

Excluding the kernels — `CORR`, `ATAX` and `BICG` — where some code regions were incorrectly classified as unsafe for parallelisation, the optimal speedups for UniBench and our automatic approach are 25.5× and 25.8×, respectively. Nonetheless, due to sub-optimal target selection decisions, the achieved geometric mean speedup is be 17.2×.

8.8 Summary

This chapter presented the evaluation results for our work. It first described the platforms and experimental setup. Then it evaluated the analytical models on manually annotated OpenMP kernels from UniBench [56], considering the different approaches studied in our work. The results show the models' capability to guide good offloading decisions, with optimal decisions in most cases. Besides, incorrect decisions that would lead to slowdowns only add fractions of a second compared to the baseline. However, the standard deviation of relative errors is significant in some kernels. Therefore, further enhancements in the analytical models are needed. Finally, we evaluated our framework as a whole, considering automatic parallelisation and the ability to select the adequate target to accelerate code regions. We achieved a 7.9× geometric mean speedup without any user intervention, reducing the total execution time from 9E3 to 1.44E3 seconds.

Chapter 9

Conclusions

Heterogeneous systems are emerging as one possible solution to answer the increasing computational power demand. To take advantage of those systems is necessary to migrate the existing code base, written initially as sequential applications. However, programming for heterogeneous systems is challenging. A heterogeneous system developer has to know many tools, learn vendor-specific programming models, and to have expertise in diverse accelerator's architectures to achieve maximum performance. Moreover, the developer has to partitioning workloads, which is a very time-consuming process. Innovations at the software level that assist developers or automate the transition to heterogeneous systems are needed.

Our literature review exhibits framework proposals that can ease the adoption of heterogeneous systems with little to no user intervention, particularly significant for common users that may not have the necessary background. However, we found some shortcomings in existing approaches that we aimed to address in our proposal, enumerated below.

Binary outcomes. Recent academic research has been embracing the LLVM infrastructure. The infrastructure is feature-packed with analysis often used in compiler research work. However, LLVM based approaches are often forced to generate a binary application. In contrast, source-to-source techniques output the modified source code. We argue it is much more flexible for the user, as he can inspect and further adjust the code for tuning. Existing source-to-source tools can expedite several steps, such as hotspot detection, necessary code transformations to use some programming models, and target selection. Knowledgeable users can work on the transformed code to achieve more performance.

Compiler enforcing. Approaches that use compiler infrastructures, such as LLVM, are tied to a specific compiler. However, using other compilers to generate the machine code may deliver better performance on some platforms — for instance, the platform vendor's toolchain. In contrast, source-to-source approaches output source code. The user can choose any compiler that supports the programming models used for parallelisation or offloading.

Target selection. A common limitation in the existing literature is the lack of automatic target selection. Naive parallelisation or offloading to GPU may result in substantial slowdowns in some applications. Chikin et al. [30] used analytical models to guide offloading decisions. Although HTrOP [67] proposes a runtime solution for dynamically selecting the accelerator, the compilation is done on the fly, adding overheads to the application execution time.

9.1 Concluding remarks

We proposed a fully static and compiler agnostic framework that automatically transforms code regions with OpenMP pragmas for running in parallel in the CPU or offloading to the GPU. Furthermore, we integrate analytical models for guiding target selection between CPU sequential, CPU parallel and GPU.

For performance modelling, we use the same base analytical models as Chikin et al. [30]. The GPU model is adapted for the OpenMP context and some limitations are addressed for modelling modern GPUs. Our approach differs from the literature as we explore the possibility of collecting metrics at the AST level rather than lower-level representations (e.g. low-level IR or machine code). Our reasoning is that each parallel loop is compiled for different targets and our focus is on relative performance estimations rather than accurate absolute timing estimations for each target. Using Clava [25] for the AST-level analysis has various advantages, as enumerated below. One obstacle is some compiler information not being accessible, such as the grid geometry, which is an input parameter for the GPU analytical model. As the OpenMP support matures in compilers, we expect future compiler versions to report such information.

- It is compiler agnostic, i.e., it is not tied to a particular compiler infrastructure such as LLVM.
- The LARA [28, 27] scripting language is more accessible than learning and setting up complex compiler infrastructures.
- Analysis at AST are closer to the source-code, which facilitates the reasoning and debugging.

For comparison, this dissertation also implements an LLVM MCA [2] approach to estimate CPU computational costs, firstly proposed by Chikin et al [30]. As LLVM MCA does not simulate control flow, it was identified the need to manually handle loop nests in the code under analysis and consider loop-level optimisations. Integrating the LLVM MCA in a source-to-source approach presented several challenges. Firstly, due to loop versioning and the assembly code structure not matching the source-code, causing the delimiting directives to appear in unexpected places and inverted orders that resulted in errors. Secondly, inserting the inline assembly directives may affect the compiler outcome, i.e., the assembly output may not be representative of the actual instructions executed in the CPU and affect the estimations accuracy. In conclusion, an LLVM MCA approach may integrate better with lower-level analysis with already applied optimisations.

Regarding code transformation for automatic parallelisation, we use AutoPar [21], a library integrated in Clava, to determine which loops are safe for parallelisation with static analysis. AutoPar generates OpenMP shared-memory pragmas for all parallelisable loops. We use AutoPar's information to determine the variable scoping necessary to break false dependencies. Then we apply a conservative parallelisation strategy rather than parallelising all loops. As AutoPar does not support the OpenMP offloading specification, our work extends the AutoPar capabilities. Using Clava and AutoPar enables our framework to output source-code. Compared to state of the art approaches that generate executables, our approach lets the user further modify the code and use any compiler with OpenMP support.

Our experimental results demonstrated a 7.9× geometric mean speedup over 14 kernels of UniBench [56]. The manually annotated code in UniBench with manual target selection can accomplish a 18.2× geometric mean speedup. Our framework's achieved speedup is limited by our automatic parallelisation strategy and non-optimal target selection in a few cases. The kernels were correctly parallelised and produced the same results as the baseline version. Our results indicate that using AST-level analysis to guide target selection may be feasible and worth researching. Analyses at the AST level may be less accurate, as shown in our comparison against LLVM MCA, but can guide optimal target selection in a relative performance modelling context. While LLVM MCA only supports CPU, our AST-level analysis is device agnostic supporting, supporting CPUs, GPUs and any other device with the proper analytical model. Despite the promising results, some kernels have a high standard deviation on relative errors for the three considered targets indicating the need to improve the analytical models and AST analysis.

9.2 Future work

We enumerate here some future work that we find more relevant.

- Workarounds for collecting metrics for the GPU analytical model. Our fully automatic approach suffers from the impossibility to collect some metrics, such as hardware resource usage and the grid geometry selected by the compiler. Our experimental evaluation collected the metrics via profiling and then inserted them via pragmas. As the compilers mature, it may be possible to dump the necessary data and collect it automatically in our framework.
- Extend the CPU analytical model in the following ways:
  - Estimate costs in memory accesses and cache hits/misses. Our experimental results show large absolute errors in memory-bound applications. Modelling the memory hierarchy should reduce those errors.
  - Model other parallelisation side effects, such as false-sharing and thread binding. Among other benefits, thread binding can improve data locality and spatiality by pinning threads that access the same data elements in closer cores to share the L2 cache.
- Extend the GPU analytical model as follows:

  - GPUs released afterwards NVIDIA Pascal architectures (CC > 6.0) introduced Tensor cores, which are specialised cores for matrix operations in the form $D = A \cdot B + C$. Using these cores can increase the throughput compared to equivalent implementations for Pascal. Besides, tensor cores can operate on smaller precision formats, increasing further throughput, and are beneficial for AI workloads in general. Therefore, correctly modelling these operations is essential for the analytical model's accuracy.
  - Model the dual-dispatching in warp schedulers. When a warp has two independent instructions, dispatch units can issue them for distinct execution units in the same clock cycle.
  - Memory hierarchy. Model the global memory hierarchy (DRAM and caches), and other specialised memories such as constant and shared memory.

- Evaluate our approach using more complex and irregular applications and possibly extend it to deal better with those applications.

- Use the analytical models for target-specific optimisations and improve the parallelisation strategy, e.g. considering task-level parallelism. For instance, consider the matrix multiplication with column-major accesses. The algorithm can be improved by interchanging loops or applying loop tiling, making it more cache-oblivious. The analytical models could guide these types of transformations. Besides, selecting the first parallelisable loop in a nest is not always ideal, and it is an important limitation in our approach.

- Support runtime decisions. Our current approach assumes that the number of operations and loop trip counts can be determined statically. However, the expressions may be parametric and depend on runtime values. Our approach can be extended to collect most of the information statically and prepare the mathematical expressions. The analytical models are implemented as C libraries and are invoked at runtime when all parameters are known.

References

[1] CUDA Occupancy Calculator. Available at http://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html.

[2] llvm-mca - LLVM Machine Code Analyzer — LLVM 13 documentation. Available at https://www.llvm.org/docs/CommandGuide/llvm-mca.html.

[3] TOP500. Available at https://www.top500.org/.

[4] NVIDIA GF100. World's Fastest GPU Delivering Great Gaming Performance with True Geometric Realism. Technical report, NVIDIA, 2010. Available at https://www.ece.lsu.edu/gp/refs/gf100-whitepaper.pdf.

[5] NVIDIA GeForce GTX 680: The fastest, most efficient GPU ever built. Technical report, NVIDIA, 2012. Available at https://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf.

[6] NVIDIA Tesla P100. The Most Advanced Datacenter Accelerator Ever Built. Technical report, NVIDIA, 2016. Available at https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[7] GDDR6: The Next-Generation Graphics DRAM. Technical report, Micron, 2017. Available at https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tned03_gddr6.pdf.

[8] The OpenACC Application Programming Interface, November 2020. Available at https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf, version 3.1.

[9] The OpenCL Specification, December 2020. Available at https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf, version 3.0.

[10] OpenMP Application Programming Interface, November 2020. Available at https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, version 5.1.

[11] CUDA C++ Programming Guide, January 2021. Available at https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, version 11.2.

[12] CUDA C++ Programming Guide. Technical report, NVIDIA, August 2021. Available at https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[13] CUDA Driver API - API Reference Manual. Technical report, August 2021. Available at https://docs.nvidia.com/cuda/pdf/CUDA_Driver_API.pdf.

[14] CUDA Runtime API - API Reference Manual. Technical report, August 2021. Available at https://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf.

[15] HCC API Documentation, December 2021. Available at https://github.com/RadeonOpenCompute/ROCm/blob/rocm-4.5.2/AMD_HIP_Programming_Guide.pdf.

[16] Integrating and Operating HBM2E Memory. Technical report, Micron, 2021. Available at https://media-www.micron.com/-/media/client/global/documents/src/product-information/micron_hbm2e_memory_wp.pdf.

[17] Intel® 64 and IA-32 Architectures Optimization Reference Manual. Technical report, Intel, June 2021. Available at https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html.

[18] Kernel Profiling Guide. Technical report, July 2021. Available at https://docs.nvidia.com/nsight-compute/pdf/ProfilingGuide.pdf.

[19] Software Optimization Guide for AMD Family 17h Processors. Technical Report 3.01, AMD, February 2021. Available at https://www.amd.com/system/files/TechDocs/55723_3_01_0.zip.

[20] Andreas Abel and Jan Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 673–686, Providence RI USA, April 2019. ACM.

[21] Hamid Arabnejad, João Bispo, João M. P. Cardoso, and Jorge G. Barbosa. Source-to-source compilation targeting OpenMP-based automatic parallelization of C applications. *The Journal of Supercomputing*, 76(9):6753–6785, September 2020.

[22] Yehia Arafa, Abdel-Hameed A. Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. Low Overhead Instruction Latency Characterization for NVIDIA GPGPUs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8, September 2019. ISSN: 2643-1971.

[23] Mark Bartlett, Iain Bate, and Dimitar Kazakov. Accurate Determination of Loop Iterations for Worst-Case Execution Time Analysis. *IEEE Transactions on Computers*, 59(11):1520–1532, November 2010.

[24] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In Rajiv Gupta, editor, *Compiler Construction*, pages 244–263. Springer Berlin Heidelberg, 2010.

[25] João Bispo and João M.P. Cardoso. Clava: C/C++ source-to-source compilation using LARA. *SoftwareX*, 12:100565, July 2020.

[26] J. Mark Bull, Fiona Reid, and Nicola McDonnell. A Microbenchmark Suite for OpenMP Tasks. In David Hutchison and et al., editors, *OpenMP in a Heterogeneous World*, volume 7312, pages 271–274. Springer Berlin Heidelberg, 2012. Series Title: Lecture Notes in Computer Science.

[27] João M. P. Cardoso, José G. F. Coutinho, Tiago Carvalho, Pedro C. Diniz, Zlatko Petrov, Wayne Luk, and Fernando Gonçalves. Performance-driven instrumentation and mapping strategies using the LARA aspect-oriented programming approach. *Software: Practice and Experience*, 46(2):251–287, 2016.

[28] João M.P. Cardoso, Tiago Carvalho, José G.F. Coutinho, Wayne Luk, Ricardo Nobre, Pedro Diniz, and Zlatko Petrov. LARA: an aspect-oriented programming language for embedded systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 179–190, New York, NY, USA, March 2012. ACM.

[29] Barbara Chapman, Deepak Eachempati, and Oscar Hernandez. Experiences Developing the OpenUH Compiler and Runtime Infrastructure. *International Journal of Parallel Programming*, 41(6):825–854, December 2013.

[30] A. Chikin, J. N. Amaral, K. Ali, and E. Tiotto. Toward an analytical performance model to select between gpu and cpu execution. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 353–362, 2019.

[31] A. Colin and G. Bernat. Scope-tree: a program representation for symbolic worst-case execution time analysis. In *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*, pages 50–59, Vienna, Austria, 2002. IEEE Comput. Soc.

[32] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, and Alessandro Rubini. *Linux device drivers*. O'Reilly, Beijing ; Sebastopol, CA, 3rd ed edition, 2005.

[33] William J. Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Commun. ACM*, 63(7):48–57, June 2020.

[34] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro*, 37(2):52–62, March 2017.

[35] Johan Enmyren and Christoph W. Kessler. Skepu: A multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications*, HLPP '10, page 5–14, New York, NY, USA, 2010. ACM.

[36] Steffen Ernsting and Herbert Kuchen. Algorithmic skeletons for multi-core, multi-gpu systems and clusters. *International Journal of High Performance Computing and Networking*, 7(2):129–138, 2012.

[37] August Ernstsson, Lu Li, and Christoph Kessler. SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming*, 46(1):62–80, February 2018.

[38] Jianbin Fang, Chun Huang, Tao Tang, and Zheng Wang. Parallel programming models for heterogeneous many-cores: a comprehensive survey. *CCF Transactions on High Performance Computing*, 2(4):382–400, December 2020.

[39] Agner Fog. Calling conventions for different C++ compilers and operating systems, 2021. Available at `https://agner.org/optimize/calling_conventions.pdf`.

[40] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs. Technical report, August 2021. Available at https://www.agner.org/optimize/instruction_tables.pdf.

[41] T Fountain, A McCarthy, and F Peng. PCI Express: an Overview of PCI Express, Cabled PCI Express and PXI Express. page 10, 2005.

[42] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 innovative parallel computing (InPar)*, pages 1–10. IEEE, 2012.

[43] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 57–66, Rio de Janeiro, Brazil, 2006. IEEE.

[44] Christopher Healy, Mikael Sjödin, Viresh Rustagi, David Whalley, and Robert van Engelen. Supporting Timing Analysis by Automatic Bounding of Loop Iterations. *Real-Time Systems*, 18(2/3):129–156, 2000.

[45] Christopher Healy, Robert Van Engelen, and David Whalley. A general approach for tight timing predictions of non-rectangular loops. In *IEEE Real-Time Technology and Applications Symposium*, pages 11–14. Citeseer, 1999.

[46] John L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers, Cambridge, MA, sixth edition edition, 2019.

[47] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture - ISCA '09*, page 152, Austin, TX, USA, 2009. ACM Press.

[48] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *CoRR*, abs/1804.06826, 2018.

[49] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. HBM (High Bandwidth Memory) DRAM Technology and Architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4, May 2017.

[50] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, January 2020.

[51] Chunhua Liao and Barbara Chapman. Invited Paper: A Compile-time Cost Model for OpenMP. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, Long Beach, CA, USA, 2007. IEEE.

[52] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[53] William Magro, Paul Petersen, and Sanjiv Shah. Hyper-threading technology: Impact on compute-intensive workloads. *Intel Technology Journal*, 6(1):1, 2002.

[54] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):1, 2002.

[55] Matt Martineau and Simon McIntosh-Smith. The Productivity, Portability and Performance of OpenMP 4.5 for Scientific Applications Targeting Intel CPUs, IBM CPUs, and NVIDIA GPUs. In Bronis R. de Supinski, Stephen L. Olivier, Christian Terboven, Barbara M. Chapman, and Matthias S. Müller, editors, *Scaling OpenMP for Exascale Performance and Portability*, volume 10468, pages 185–200. Springer International Publishing, Cham, 2017.

[56] Luís Felipe Mattos, Rafael Cardoso, and Márcio Pereira. Unibench, 2015. Available at https://github.com/ompcloud/Unibench.

[57] Cameron McNairy and Don Soltis. Itanium 2 Processor Microarchitecture. *IEEE MICRO*, page 12, 2003.

[58] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. Dawncc: Automatic annotation for data parallelism and offloading. *ACM Trans. Archit. Code Optim.*, 14(2), May 2017.

[59] Alok Mishra, Abid M. Malik, and Barbara Chapman. Data transfer and reuse analysis tool for gpu-offloading using openmp. In Kent Milfeld, Bronis R. de Supinski, Lars Koesterke, and Jannis Klinkenberg, editors, *OpenMP: Portable Multi-Level Parallelism on Modern Systems*, pages 280–294, Cham, 2020. Springer International Publishing.

[60] Ken Mitchell and Elliot Kim. AMD RYZEN™ CPU OPTIMIZATION. Technical report, AMD, March 2017. Available at https://en.wikichip.org/w/images/4/42/amd-ryzen-cpu-optimization.pdf.

[61] Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4), July 2015.

[62] Gordon E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, September 2006.

[63] S. Muchnick, Morgan Kaufmann Publishers, M. Associates, and Elsevier (Amsterdam). *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[64] Cedric Nugteren and Henk Corporaal. Introducing'bones' a parallelizing source-to-source compiler based on algorithmic skeletons. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 1–10, 2012.

[65] Gabriel Poesia, Breno Guimarães, Fabrício Ferracioli, and Fernando Magno Quintão Pereira. Static placement of computation on heterogeneous devices. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[66] C D Polychronopoulos. Loop coalescing: A compiler transformation for parallel machines. 1 1987.

[67] Heinrich Riebler, Gavin Vaz, Tobias Kenter, and Christian Plessl. Transparent acceleration for heterogeneous platforms with compilation to opencl. *ACM Trans. Archit. Code Optim.*, 16(2), April 2019.

[68] John Shalf. The future of computing beyond Moore's Law. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 378(2166):20190061, March 2020.

[69] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating system concepts*. Wiley, Hoboken, NJ, 9th edition, 2013.

[70] David Suggs, Mahesh Subramony, and Dan Bouvier. The AMD "Zen 2" Processor. *IEEE Micro*, 40(2):45–52, March 2020.

[71] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013.

[72] Farui Wang, Weizhe Zhang, Haonan Guo, Meng Hao, Gangzhao Lu, and Zheng Wang. Automatic translation of data parallel programs for heterogeneous parallelism through OpenMP offloading. *The Journal of Supercomputing*, October 2020.

[73] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining Loop Transformations Considering Caches and Scheduling. *International Journal of Parallel Programming*, 26(4):479–503, 1998. Number: 4.

[74] K.C. Yeager. The Mips R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–41, April 1996.

[75] Tomofumi Yuki and Louis-Noël Pouchet. Polybench 4.0, 2015. Available at `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`.

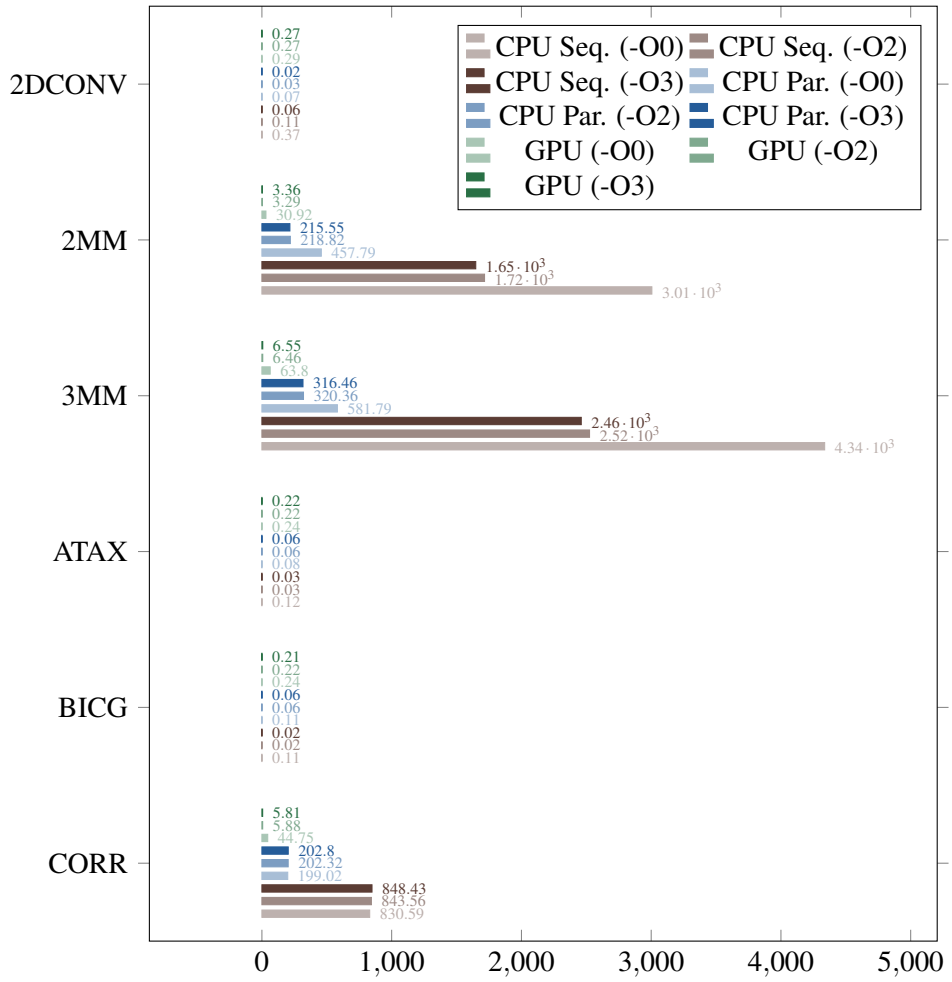
[76] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. page 7, 2020.

[77] Tomas Öhberg, August Ernstsson, and Christoph Kessler. Hybrid CPU–GPU execution support in the skeleton programming framework SkePU. *The Journal of Supercomputing*, 76(7):5038–5056, July 2020.

Appendix A

Benchmark characterisation and additional experimental results

A.1 UniBench kernel execution times



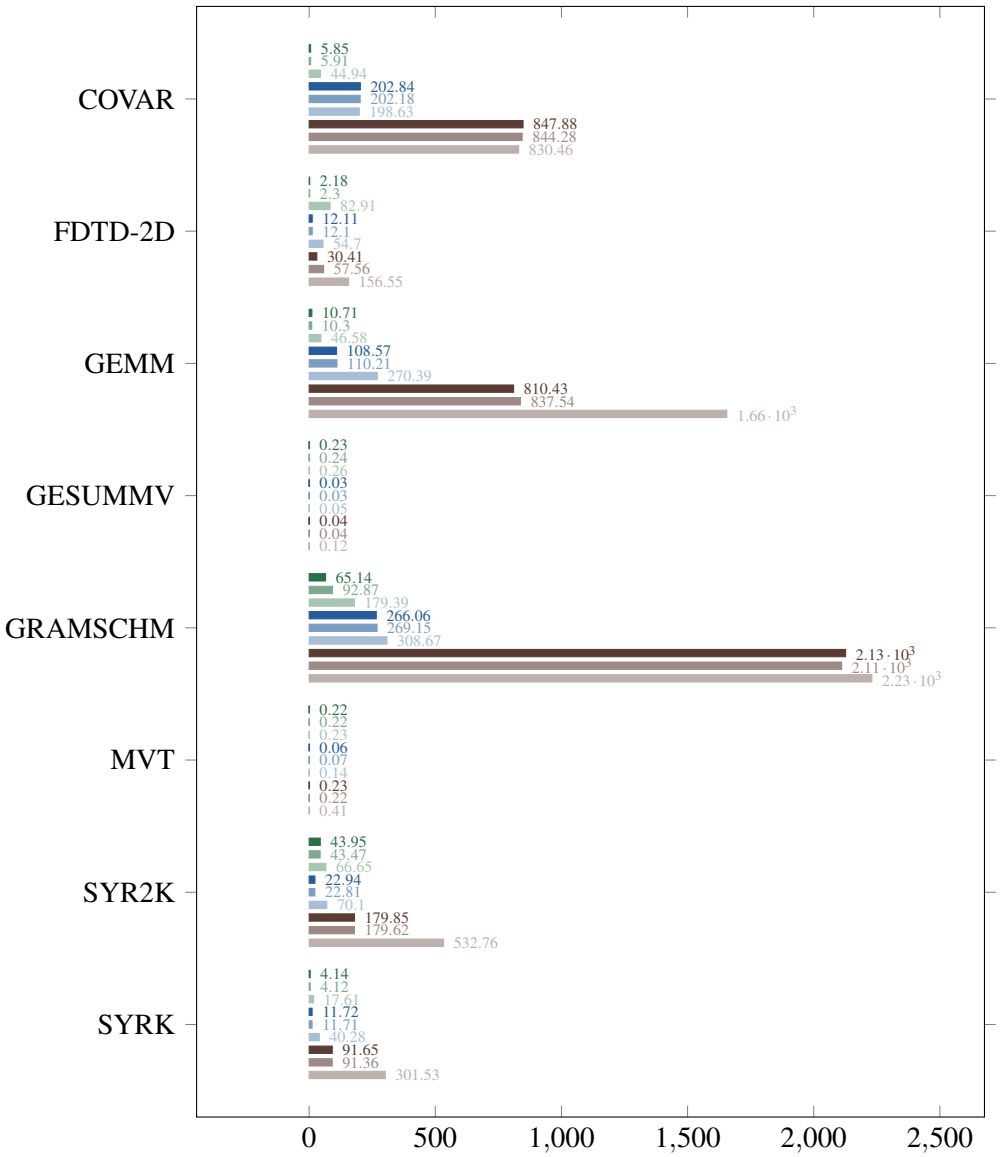


Figure A.1: Unibench kernels execution times for different targets and varying optimisation flags. For the `-O2`, vectorisation is disabled.

A.2 UniBench kernel characterisation

Table A.1 characterises the kernels in UniBench [56], which is useful to interpret evaluation results. Each kernel application in UniBench may have one or more OpenMP *workshare regions*. The table shows the characteristics for each *workshare region*. For instance, the matrix multiplication kernel, `2MM`, has two *workshare regions*, listed in the table in order of appearance in the source code: `2mm_1`, and `2mm_2`.

The columns arithmetic and memory operations show the number of operations per workshare region as a function of the problem size N. For arithmetic operations, loop headers and array subscripts are ignored. Memory operations only count array accesses and are split in read and

write operations. It is assumed that the scalar variables are stored in registers.

The columns *Host To Device (HtD)* and *Device to Host (DtH)* indicate the number of elements transferred explicitly between the host and the accelerator. The number of elements corresponds to memory positions in the arrays being transferred and in general is a function of the problem size, N. The effective number of bytes transferred depends on array's data types. Workshare regions in UniBench are grouped in data environments [10]. A data environment in OpenMP manages memory in a data sharing context. For instance, when the computation is offloaded to an accelerator (e.g., a GPU), the data environment specifies which variables or memory regions have to be transferred from the host to the accelerator and vice-versa, as well as specify variables to be allocated in the accelerator. In UniBench, most kernels move data to the accelerator before executing the workshare regions. When the computation completes, the results are transferred back from the accelerator to the host. In such cases, and assuming the kernel has multiple workshare regions, the number of data elements transferred is reported just for the first and last workshare region. For example, in `3MM` all workshare regions are grouped in the same data environment, therefore data movements are reported for the first and last workshare regions.

The rightmost columns in Table A.1 report compiler optimisations when the kernels are compiled with Clang and `-O3`. For vectorisation, both vector width and interleaving factors are reported. If `inter` is omitted then interleaving is not applied. A workshare region may have nested loops. Therefore, the optimisations indicate a rank to specify which loop is optimised. The rank is relative to the function.

Table A.1: UniBench kernel's characterisation.

Kernel	**Arith. Ops**	**Mem. Ops**		**Data Transf.**		**Loop Opt.**	
		R	**W**	**HtD**	**DtH**	**Vectorisation**	**Unrolled**
`2dconv`	$17 \times N^2$	$9 \times N^2$	N^2	N^2	N^2	Rank=1_1, width=4	No
`2mm_1`	$2 \times N^3$	$2 \times N^3$	N^3	$3 \times N^2$	0	No	Rank=1_1_1, factor=2
`2mm_2`	$2 \times N^3$	$2 \times N^3$	N^3	0	$2 \times N^2$	No	Rank=2_1_1, factor=2
`3mm_1`	$3 \times N^3$	$2 \times N^3$	N^3	$4 \times N^2$	0	No	Rank=1_1_1, factor=2
`3mm_2`	$3 \times N^3$	$2 \times N^3$	N^3	0	0	No	Rank=2_1_1, factor=2
`3mm_3`	$3 \times N^3$	$2 \times N^3$	N^3	0	$3 \times N^2$	No	Rank=3_1_1, factor=2
`atax_1`	$3 \times N^2$	$2 \times N^2 + N$	N	$N^2 + 3 \times N$	0	No	Rank=2_1, factor=2
`atax_2`	$3 \times N^2$	$2 \times N^2$	N	0	$2 \times N$	No	Rank=3_1, factor=2
`bicg_1`	$3 \times N^2$	$2 \times N^2$	N	$N^2 + 3 \times N$	0	No	Rank=2_1, factor=2
`bicg_2`	$3 \times N^2$	$2 \times N^2$	N	0	$2 \times N$	No	Rank=3_1, factor=2
`corr_1`	$N^2 + N$	N^2	N	$2 \times N^2 + 2 \times N$	0	No	Rank=1_2, factor=4

Continues...

Table A.1: UniBench kernel's characterisation.

Kernel	**Arith. Ops**	**Mem. Ops**		**Data Transf.**		**Loop Opt.**	
		R	**W**	**HtD**	**DtH**	**Vectorisation**	**Unrolled**
`corr_2`	$2 \times N^2 + 3 \times N$	$N^2 + N$	N	0	0	No	Rank=2_2, factor=2
`corr_3`	$4 \times N^2$	$N^2 + 2 \times N$	N^2	0	0	Rank=3_4, width=4	No
`corr_4`	$N^3 + 5 \times N^2 + 8 \times N + 4$	0	0	0	N^2	No	Rank=4_1_1, factor=2
`covar_1`	$N^2 + N$	N^2	N	$2 \times N^2$	0	No	Rank=1_1, factor=4
`covar_2`	N^2	$2 \times N^2$	N^2	0	0	Rank=2_1, width=4, inter=2	Rank=2_1, factor=2
`covar_3`	$2 \times N^3$	$N^3 + 4 \times N^2 + 5 \times N + 2$	$(N^2 + 3 \times N + 2)/2$	0	N^2	No	Rank=3_1_1, factor=2
`fdtd-2d_1`	0	1	N	$500 + 3 \times N^2$	0	Rank=1_1_1, width=4, inter=2	Rank=1_1_1, factor=8
`fdtd-2d_2`	$3 \times N^2$	$3 \times N^2$	N^2	0	0	Rank=1_2_1, width=4	Rank=1_2_1, factor=2
`fdtd-2d_3`	$3 \times N^2$	$3 \times N^2$	N^2	0	0	Rank=1_3_1, width=4	No
`fdtd-2d_4`	$5 \times N^2$	$5 \times N^2$	N^2	0	N^2	Rank=1_4_1, width=4	No
`gemm`	$3 \times N^3 + N^2$	$2 \times N^3 + N^2$	N^2	$3 \times N^2$	N^2	No	Rank=1_1_1, factor=2
`gesummv`	$5 \times N^2 + 3 \times N$	$3 \times N^2$	N	$2 \times N^2$	N	No	Rank=1_1, factor=2
`mvt_1`	$3 \times N^2$	$2 \times N^2 + N$	N	$N^2 + 2 \times N$	N	No	Rank=1_1, factor=4
`mvt_2`	$3 \times N^2$	$2 \times N^2 + N$	N	0	N	No	Rank=2_1, factor=2
`syr2k`	$8 \times N^3$	$2 \times N^3 + N^2$	N^2	$3 \times N^2$	N^2	No	Rank=2_1_1, factor=2
`syrk_1`	N^2	N^2	N^2	$2 \times N^2$	0	Rank=1_1, width=4, inter=2	Rank=1_1, factor=2
`syrk_2`	$3 \times N^3$	$2 \times N^3 + N^2$	N^2	0	N^2	No	Rank=2_1_1, factor=2

A.3 LLVM MCA approaches evaluation

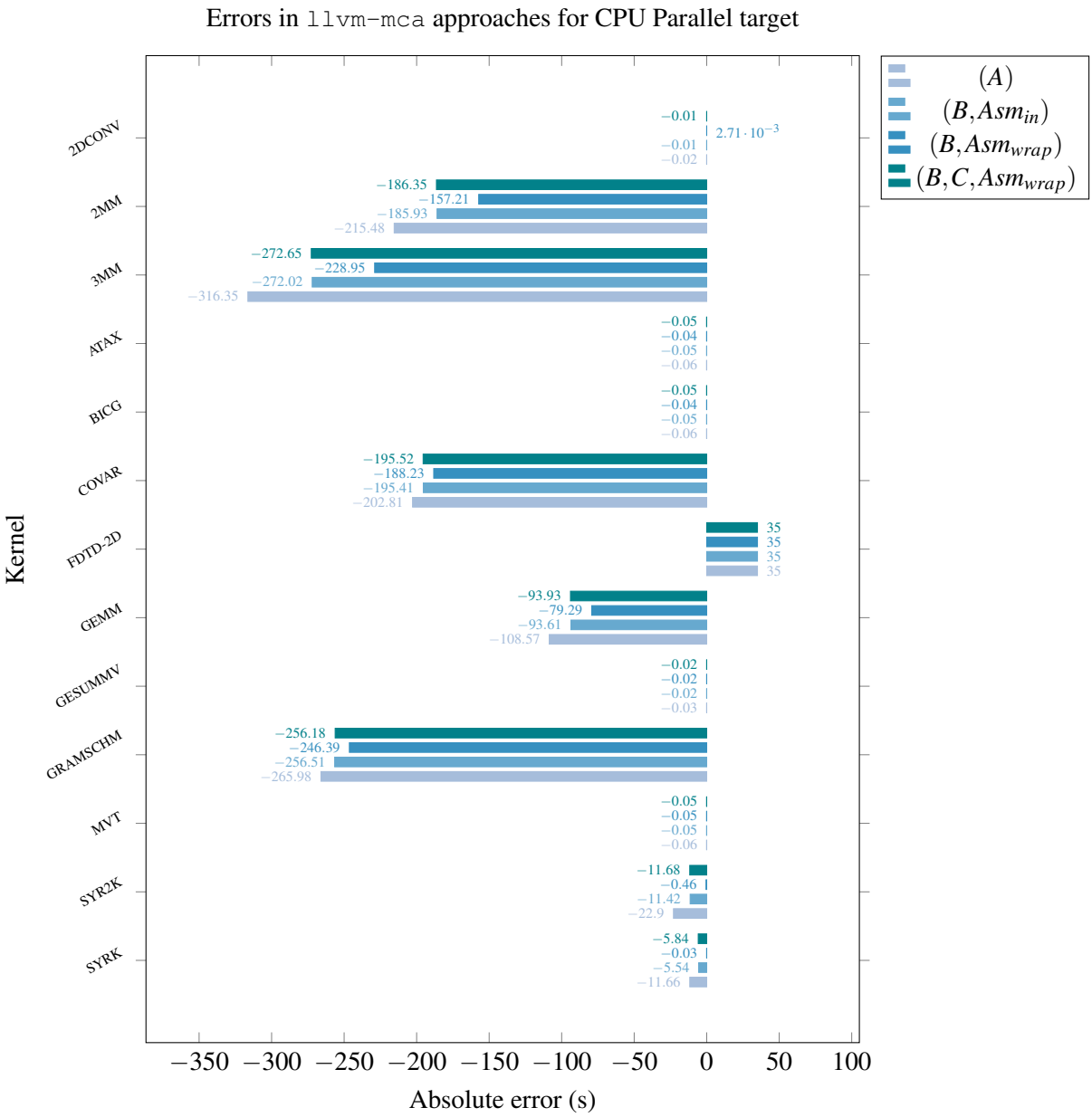


Figure A.2: Absolute error in `llvm-mca` approaches for CPU Parallel. Measurements obtained with Clang, `-O3`, and problem size set to $N = 4096$.

A.4 Comparing AST and LLVM MCA approaches

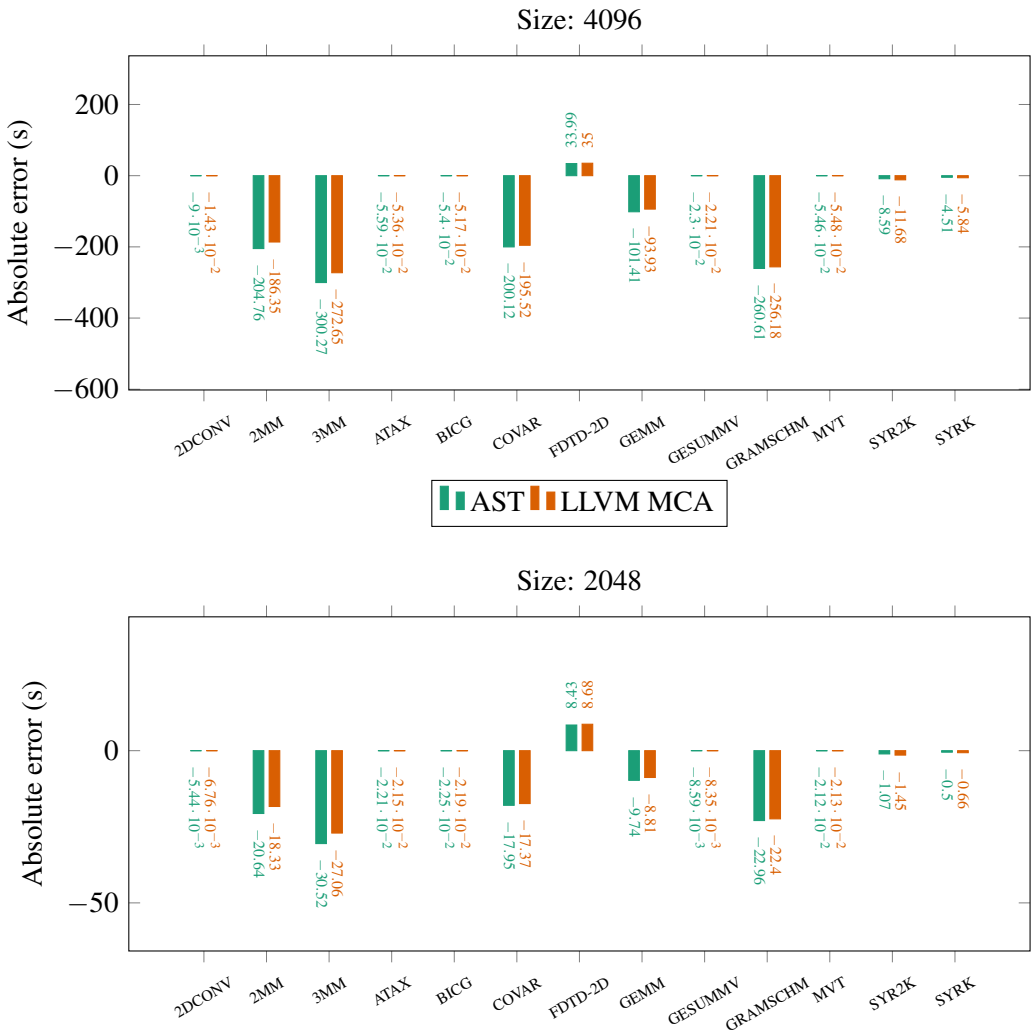


Figure A.3: Absolute error comparison for `llvm-mca` and AST approaches in CPU Parallel estimations, with Clang and `-O3`.