POLITECNICO DI TORINO

Master's Degree Course
in Computer Engineering

Master's Degree Thesis

A Comparative Analysis of Methods and Tools for Identification of GPU-friendly Algorithms

Supervisors
Prof. Alessandro SAVINO
Dr. Giulio GAMBARDELLA

Candidate
Jacopo PATI

Academic Year 2022-2023

Alla mia famiglia.
A Erika.
Grazie del vostro costante supporto.

Abstract

In solving advanced computational problems, GP-GPUs (General-Purpose Graphics Processing Units) have gained prominence in recent years. The adoption of GPUs not only for computer graphics allows to exploit their huge compute performance and high level of parallelism to lighten the CPU from burdensome executions. Aim of this thesis is to analyze automatic methodologies to help developers to find code for acceleration, studying ways to identify functions amenable to GPU acceleration without prior knowledge or assumption on the code-base.

The first parameter useful in identifying such loops is Arithmetic Intensity (AI). The higher is the value, the most likely the code will benefit from GPU offloading. Although AI is independent from the hardware characteristics, it can be related to the FLOPs/s of the machine through an analysis called Roofline, in order to also identify if the considered functions are memory or compute bound. We analyzed different tools, namely Intel VTune together with SDE, RRZE LIKWID and Intel Advisor, allowing to calculate AI with little effort on the developer side. A set of publicly available benchmarks (KernelGen Test Suite) has been used to evaluate and compare the tools in depth, relying on implementations from third party of the code base to GPU (e.g., CUDA) as a golden reference. We demonstrate how Intel Advisor provides the best evaluation, thanks to its ability to model GPU execution advantages.

Given the limit on the AI evaluation as standalone metric for GPU offload potential, we improved our analysis including the modeling of the speed up on GPU versus the CPU counterpart, providing further hints thanks to the Intel Advisor feature, using as end-to-end approach evaluation the open source Static Time Analysis (STA) tool OpenTimer. Finally, given the estimation provided by the benchmarked tools, we ported core function of OpenTimer to GPU with minimal code changes using OpenACC to validate the AI evaluation.

Despite being an important first step in assessing GPU offload opportunities, we conclude that AI evaluation is not enough information to completely estimate possible advantage of GPU execution. The evaluation is based on the CPU implementation without techniques, like batching or vectorization, that could increase GPU advantages.

Contents

List of Tables

List of Figures

Acronyms

AI Arithmetic Intensity. 7, 11, 17–20, 23, 30–33, 36–38, 41, 44–49, 51–53, 55–58, 60, 62–64, 66–69, 71–73, 79, 89, 90, 97, 108–110

CLI Command Line Interface. 25, 26, 36, 70

CPU Central Processing Unit. 11, 13, 15

DAG Direct Acyclic Graph. 80, 90

DFS Depth-First Search. 91, 92, 103, 110

EBS Event-Based Sampling. 24, 26

EDA Electronic Design Automation. 79

FLOP/s floating-point operations per second. 20

FLOPs floating-point operations. 17, 18, 23, 27–29, 31, 35, 37, 38, 41, 43, 46, 48, 51, 53, 55–58, 60, 62–64, 66, 68–73, 97, 107, 108

GPGPU General-Purpose Graphics Processing Unit. 11, 16, 113

GPU Graphic Processing Unit. 7, 11–19, 23, 24, 33–36, 38, 69, 71, 79, 82–89, 91–99, 101, 103, 104, 107–111, 113, 114

GUI Graphical User Interface. 24, 33, 34

HPC High Performance Computing. 11, 24, 29, 42, 94, 114

IMC Integrated Memory Controller. 26, 31, 107

ITT Instrumentation and Tracing Technology. 24, 38, 43

MSR Model Specific Registers. 30

PCI-Express Peripheral Component Interconnect Express. 13, 15

SIMD Single Instruction, Multiple Data. 15, 16

SIMT Single Instruction, Multiple Threads. 16

SM Streaming Multiprocessor. 7, 13, 14

STA Static Time Analysis. 11, 79, 109, 113

Chapter 1

Introduction

The use of Graphic Processing Units (GPUs) has always been prominent in the field of computer graphics with a growing demand for hardware also supported by the gaming industry. Lately, thanks to the constant innovation and progress leading to the continuous increase in their performance, developers are starting to integrate more and more GPUs use also in High Performance Computing (HPC), cluster computers able to provide extremely elevated performance, primarily used for scientific research [3].

The monsters that move on the playing field of a video game have been transformed into particles, and their smashing have become particle collisions. Hence the term General-Purpose Graphics Processing Unit (GPGPU) is born, broadens their field of use beyond that of graphics, processing large amount of data in parallel, which is something traditional CPUs are not well-suited for [46].

Writing code that can run on GPUs, however, can be challenging for several reasons, including that GPU vendors have different programming model and libraries, which can take time to learn, and it is not easy to identify functions that are worth offloading without prior knowledge or assumption, especially in large code projects.

During my six months internship hosted by "Synopsys International LTD" in Dublin, an electronic design automation (EDA) company, we studied ways to identify functions amenable to GPU acceleration without prior knowledge or assumption on the code-base.

Arithmetic Intensity (AI) is the first parameter that will be evaluated to identify algorithms that could benefit from GPU acceleration. We will analyze different tools, namely Intel VTune together with SDE, RRZE LIKWID and Intel Advisor, allowing to calculate AI with little effort on the developer side. A set of publicly available benchmarks (KernelGen Test Suite [29]) will be used to evaluate and compare the tools in depth, relying on implementations from third party of the code base to GPU (e.g., CUDA) as a golden reference.

The analysis will be improved by including the modeling of the speed up on GPU versus the CPU counterpart, providing further hints thanks to the Intel Advisor feature, using as end-to-end approach evaluation the open source Static Time Analysis (STA) tool OpenTimer [43]. This application has been chosen because it is open source and operates

in a similar area as the Synopsys team I worked with. We'll try to implement one of its core functions to run on GPU, using OpenACC directives, also comparing two different approach of code compiling to enable acceleration.

Final aim of this thesis will be to analyze opportunities and obstacles of methodologies that can help developers to find code to be accelerated on GPU. We are interested in understanding whether it can be done effectively automatically, evaluating different tools, in order to simplify the developer's work.

Chapter 2

Background

2.1 GPU Architecture Fundamentals

The Graphic Processing Unit (GPU) is a specialized electronic circuit equipped with many parallel processing elements and an high bandwidth memory hierarchy which supports the Central Processing Unit (CPU) inside a computer for specific tasks.

Different manufacturers, such as AMD, NVIDIA, Intel, ASUS and many more, produce GPUs with slightly different characteristics and nomenclatures. To learn more, we chose to focus mostly on those produced by the supplier NVIDIA. They are composed by [34]:

- A large number of processing units called *Streaming Multiprocessors (SMs)* (in AMD GPUs denominated *Compute Units (CUs)* [1]). Within each SM, we find multiple instruction execution pipelines, each containing hundreds of cores, an instruction scheduler responsible of efficiently dispatching instruction to the processing units and a dedicated L1 cache;

- L2 shared cache part of the chip package, used to decrease memory latency and increase bandwidth of memory accesses;

- Fast private DRAM, also known as *global memory*. As shown in Figure 2.1, the CPU, which is in charge of running the main program, transfers a copy of the data from its main memory, via the PCI-Express interface, to the GPU one with an operation called *Host-to-Device copy* (H→D). Once present, the device will perform the required algorithms and can also allocate and deallocate local memory. When the computation is completed, results are brought back to the CPU via a *Device-to-Host copy* (D→H) [31].

Refer to Figure 2.2 for a simplified visual representation of a GPU structure.

Taking as an example their NVIDIA A100 GPU, released in the last quarter of 2020 specifically for data centers, it is equipped with 80 GB of DRAM, accessed by 40 MB L2 cache, and contains 108 SMs, including within it a total of 8192 FP32 cores, performing

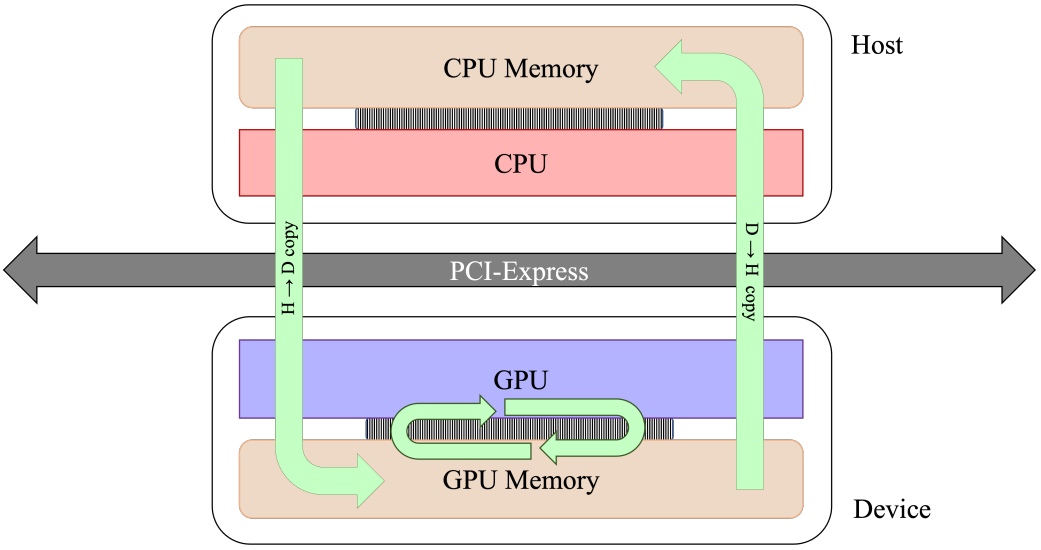


Figure 2.1: Memory transfers between host and device. The CPU moves data via an Host-To-Device copy, the GPU works on them, with the possibility of allocating local memory, and then can return the result back to the CPU via a Device-To-Host copy.

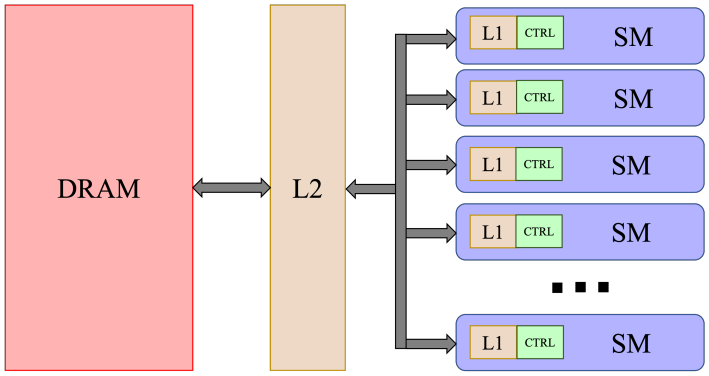


Figure 2.2: The GPU architecture in a simplified view, showing the shared DRAM and L2 cache along with the large number of SMs each with its own L1 cache and instruction scheduler. *Source: NVIDIA* [34].

mathematical operations on single-precision floating-point numbers, 8192 INT32 cores and 65536 32-bits registers [35].

Its technical characteristics highlight some advantages of the GPU programming model, including [42]:

- *Computational power*: the large number of optimized processing cores allows to perform a massive number of operations in a short amount of time;
- *High parallelism*: GPUs are designed to perform multiple operation simultaneously by splitting up large data sets into smaller chunks. Different processing cores execute

the same instruction but on different data and the results could be then combined together to produce the final output. This parallel processing technique is called Single Instruction, Multiple Data (SIMD);

- *Minimized data movement*: when working on data-intensive algorithms, the CPU may need to read large amount of data multiple time from the memory hierarchy to perform computation on different chunks. On the GPU instead, once you get past the PCI-Express interface data transfer bottle neck, which in its 5th generation is limited to 64 GB/s, the large number of processing cores can execute the computation on a larger portion of data simultaneously. This overcomes the need to read the same data multiple time, processing it more quickly and efficiently [31].

GPUs are a powerful tool for the high-performance computing field. However, not all applications are suitable for GPU parallelism, as it requires careful consideration on the data it is working with to take full advantages of the hardware features.

2.1.1 Comparison with the CPU model

In order to clearly and easily perceive the main differences between CPU and GPU model, Tolga Sotaya, author of [46], makes use of a straightforward analogy. Imagine a coconut harvesting competition in which three teams participate: a strong and experienced farmer, Arnold, two young farmers, Fred and Jim, with a less powerful tractor than Arnold's and finally Tolga, a farmer who instead of a tractor drives a bus with 32 young scouts, helping him pick the fruit from the trees. Who would win this race?

In the example Arnold represents a single-threaded 4 GHz CPU, Fred and Jim a dual-core CPU with each core running at around 2.5 GHz and Tolga a single CPU running a GPU with 32 small low-power cores. The victory of the competition depends on several factors, but it can be seen that if there are many coconuts trees and Tolga is able to coordinate his high number of no-experienced helpers efficiently, he can win the competition. This is certainly a conceptual simplification of the two devices but helps to imagine in a simple way how a CPU alongside a GPU can, in certain fields, have superior performance to a much more powerful CPU alone.

A comparison of the two architectural models is shown in Figure 2.3. The CPU is optimized for serial tasks and designed to run threads as fast as possible, thanks to its complex control logic and large caches. As showed in the figure, nowadays in a single processing chip it is possible to find several cores. Each one has its own ALU and registers that together are able to carry a thread execution independently from the others in the chip. Multiple threads can run together until there is available hardware for their execution. It may happen however in a CPU that the threads invoked are greater than the number of cores present. In this case a *Context Switch* takes place. The context of the current process needs to be saved and it includes several data such as the program counter and the contents of the registers, which will allow to resume execution at a later time. It is computationally very expensive and requires a considerable period to carry out all the necessary operations. Thanks to its architecture, the GPU is capable of executing thousands of threads simultaneously on the SMs and registers, that can still maintain the

status of the thread cancelling the context switch cost. This execution model is called Single Instruction, Multiple Threads (SIMT), where the Single Instruction, Multiple Data (SIMD) paradigm embraces multithreading [14].

Finally, it should be remembered that the CPU is in charge of coordinating and managing the work, sending the data to the GPU for computation, as the GPU is just an executor that returns the results back. The purpose of GPGPU programming model is to make two separate devices with different characteristics to work together and achieve best possible performance. However, this model has some limitations that may prevent the acceleration of algorithms, as detailed in the next paragraph.

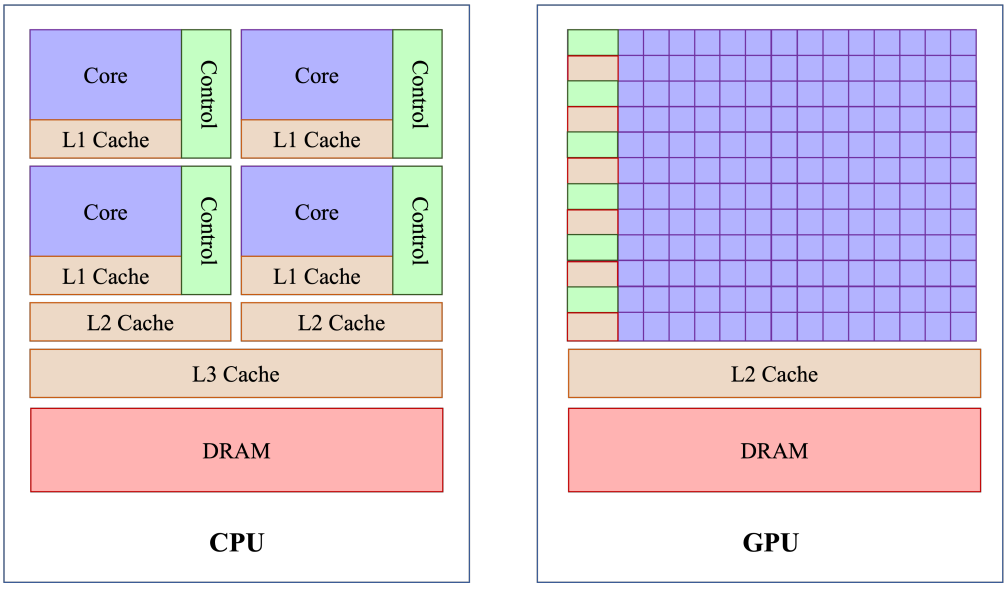


Figure 2.3: Overview of the differences between the architecture of a CPU and a GPU. Note in the case of the GPU a much higher number of processing cores.

2.2 GPGPUs Programming Challenges

Taking full advantage of the GPU parallelism, especially in fields outside of computer graphics, is not always simple and can be difficult for several reasons. Listed below are some common challenges that a developer may encounter when attempting to accelerate their code:

1. *Identification of GPU-friendly algorithms*: working on a large existing code base, and perhaps written by different developers, it can be very complex to identify algorithms to be offloaded on the GPU. Also, not all code can be accelerated or is worth it, for example, if it requires sequential processing where each steps depends on the previous one, or if it involves frequent small data transfers;

2. *Data transfers*: in Section 2.1 we introduced how data is synchronized from the CPU main memory to the global memory of the GPU. This aspect often turns out to be the bottleneck of the offloading operation and therefore the goal is to minimize the data movement between the host and the device, to take full advantage of performance;

3. *Programming models*: writing code that can run on GPGPUs can be complex for a developer who is new to this world and the learning curve can be steep. It involves a different programming model that could make use of libraries like NVIDIA *cuBLAS*, directives such as OpenACC, or programming languages as NVIDIA *CUDA* or AMD *HIP*. The source code is then enriched with one of these methods, which imply a different degree of difficulty, in order to highlight the algorithms to be offloaded on a GPU;

4. *Portability*: code written in certain GPU programming languages, such as CUDA or HIP, is often optimized for a specific GPU architecture and can only run on proprietary device. Porting code on different architectures or on a device of a different vendor could be complex. Universal languages such as OpenCL or libraries such as OpenACC have been developed to try overcome this obstacle.

While all of these aspects will be taken into consideration, the focus of our studies will be primarily on the first point of the list, looking for methods and tools that could help the developer to identify algorithms suitable for offloading. We will analyze whether this is achievable via modeling the speed up of the code on the GPU versus the CPU counterpart or also by evaluating the Arithmetic Intensity (AI) of the algorithm.

2.3 Arithmetic Intensity as Metric

Finding code that can benefit from GPGPUs in an already existing code base can be very complex. A factor that developers can take into consideration to identify algorithms to accelerate is their Arithmetic Intensity and in this thesis work we want to see if it can be a sufficient parameter and how it can be evaluated automatically.

Arithmetic Intensity (AI) of an algorithm, also known as Operational Intensity, is defined by [14]:

$$AI = \frac{FLOPs}{Memory\ accesses}\ . \tag{2.1}$$

where in (2.1):

- FLOPs are the amount of floating-point operations carried out by a segment of code;
- Memory accesses are the total of bytes required to execute it. In the context of GPU offloading they are the sum between the data transfer H→D and D→H for the algorithm.

An example, such as matrix multiplication, can help better understand this concept. Matrix multiplication is a widely used operation in different fields, like in Neural Networks. Given a matrix A of $n \cdot m$ values and a matrix B of $m \cdot p$ values, the operation produces as output the matrix C of $n \cdot p$ elements.

Each cell c_{ij} of the resulting matrix C is determined by:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in} + b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}. \tag{2.2}$$

where a_{ij} and b_{ij} are the elements of A and B, respectively.

A canonical implementation of this operation, written in C++ code, can be the following:

Listing 2.1 Matrix multiplication in C++.

```
void MatrixMultiplication(float **A, float **B, float **C, int rows_A,
    int cols_A, int cols_B){
    for (int row = 0; row < rows_A; row++) {
        for (int col = 0; col < cols_B; col++) {
                for (int k = 0; k < cols_A; k++) {
                    C[row][col] += A[row][k]*B[k][col];
                }
        }
    }
}
```

It is possible to calculate the FLOPs of the operation as:

$$FLOPs_{(matrix\ multiplication)} = n \cdot p \cdot (2m - 1). \tag{2.3}$$

since $n \cdot p$ operations are involved where each includes m multiplications and $(m-1)$ sums. The bytes related are:

$$Bytes_{(matrix\ multiplication)} = (n \cdot p + m \cdot p + n \cdot m) \cdot datatype\ size. \tag{2.4}$$

since the reads in memory are $(n \cdot m + m \cdot p)$ and the writes $(n \cdot p)$, each multiplied for the size of the data type considered.
For single-precision floating-points values of 4 bytes, like *float* data type in C++, the resulting AI of the operation is:

$$AI_{(matrix\ multiplication)} = \frac{n \cdot p \cdot (2m - 1)}{(n \cdot p + m \cdot p + n \cdot m) \cdot 4}. \tag{2.5}$$

Table 2.1 shows the values of AI for the multiplication between two matrices of different example sizes. It can be seen that as the size of rows and columns increases, the number of FLOPs grows much more than the memory accesses and so does the value of the AI, making it an advantageous operation to perform on the GPU.

Table 2.1: Arithmetic Intensity, obtained as the ratio between GFLOPs and memory accesses, for the multiplication of two matrices with different example sizes.

Matrix A size	Matrix B size	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/Bytes]
512 x 512	512 x 512	0.268	0.003	89.333
1024 x 512	512 x 1024	1.073	0.008	134.125
1024 x 1024	1024 x 1024	2.146	0.012	178.833
2048 x 1024	1024 x 2048	8.586	0.033	260.182
2048 x 2048	2048 x 2048	17.176	0.050	343.520
4096 x 2048	2048 x 4096	68.793	0.134	513.381
4096 x 4096	4096 x 4096	137.422	0.201	683.692

Arithmetic Intensity and hardware offloading are closely related: higher is the AI value, higher is the chance that the algorithm will benefit of the GPU features. Figure 2.4 shows different values for different algorithms on the axis represented by the AI. As we have seen, its value for an algorithm is not always fixed, but can depend on parameters defined at runtime.

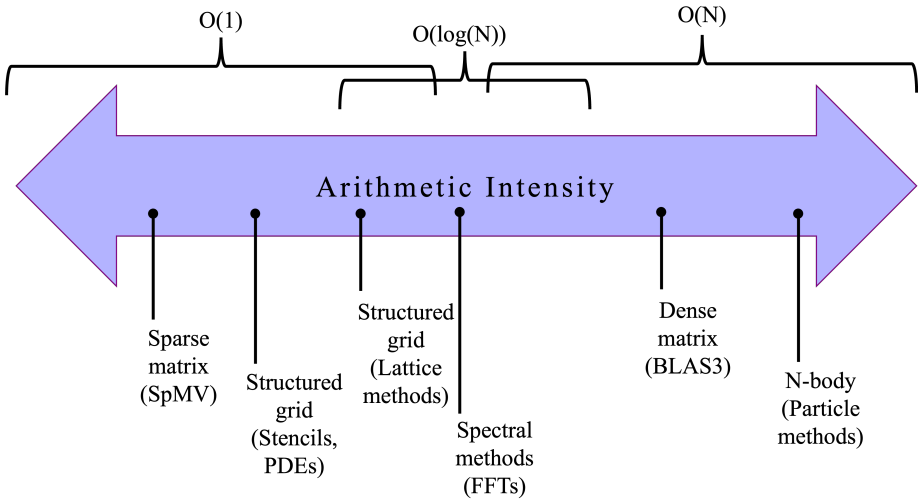


Figure 2.4: Axis representing Arithmetic Intensity (AI) with different example of algorithms. Higher is the value, higher is the change they will benefit from GPU offload. *Source: "Computer Architecture: a Quantitative Approach"* [14].

The challenge is being able to calculate Arithmetic Intensity of an algorithm automatically, without having to compute it manually as seen previously, and to do this we will use some tools introduced in the next chapter. Arithmetic Intensity has also a crucial purpose in allowing to understand if the execution of an algorithm is optimal or if it is

bounded by some factors, thanks to the use of a visual paradigm: the Roofline model.

2.3.1 Roofline Visual Model

The Roofline Model allows to understand what is limiting the performance of an algorithm, which optimization to follow and when the applied enhancements are sufficient. Basically, it can helps developers to visually quantify how their code performs.

This model uses Cartesian axis, where:

1. On the x-axis is placed the Arithmetic Intensity (AI);
2. On the y-axis the peak performances, expressed as floating-point operations per second (FLOP/s), or GFLOPS/s if expressed as multiple of 10^9.

The "roof" is built by drawing a line representing the maximum performance that a machine can achieve. It is formed by putting together the memory bandwidth limit of the machine under consideration, with the computational limit.

Afterwards the algorithm, in the form of a dot representing its characteristics, is placed on the graph. Its position towards the intersection T_H, as seen in the example Roofline Model in Figure 2.5, allows to understand the type of bound of the algorithm:

- Computation with $AI \leq T_H$ are memory bound;
- Computation with $AI \geq T_H$ are compute bound.

Since Arithmetic Intensity (AI) is a parameter independent from the hardware characteristics, the Roofline Model allows to understand the bound of an algorithm relating it to the actual machine under consideration.

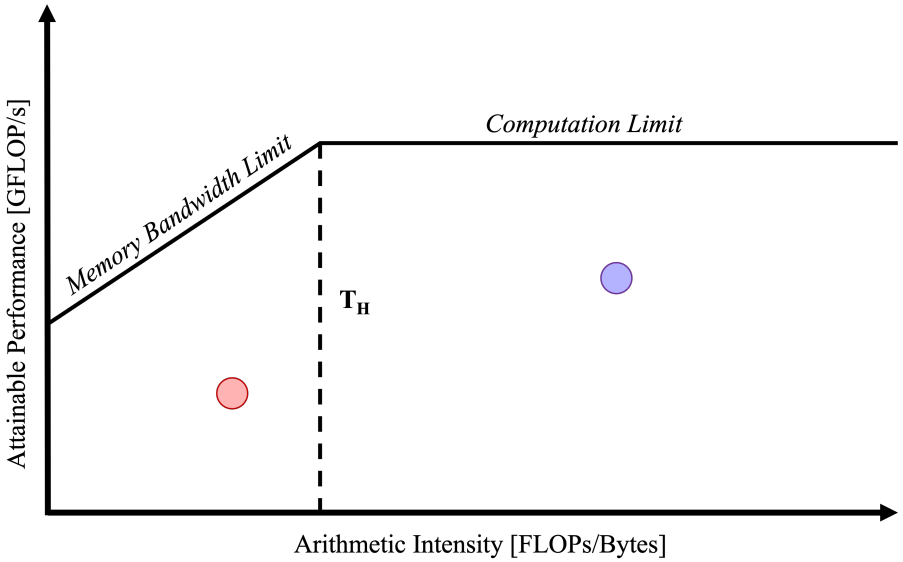


Figure 2.5: Example of a Roofline model graph with two points plotted, each representing an algorithm. The red one, to the left of the threshold, is memory bound and the purple one is compute bound. *Source: "Applying the Roofline Model"* [39].

Chapter 3

Tools and Methods for GPU Offload Assessment

In this chapter we will introduce and study several software tools to help developers evaluate algorithms that could potentially benefit from GPU execution.

The main parameter we will focus on is the Arithmetic Intensity (AI). As we have seen, the higher its value, the higher the chance that the algorithm can benefit from GPU acceleration. Compute the AI of an algorithm involves measuring the operations it performs and the memory accesses it makes. However, manually calculating AI multiple times for different sections of code can be a major obstacle and its value may also depend on parameters only known at runtime.

Even though evaluating AI is not the main purpose of these tools, we will analyze how they can be configured and utilized to calculate it automatically.

3.1 Intel VTune with SDE

The first proposed method to calculate the Arithmetic Intensity (AI) of an algorithm involves using two tools in conjunction, as suggested in the article written by Yang [51]. Intel VTune is used to evaluate the memory accesses and Intel SDE to measure the floating-point operations (FLOPs). The values collected from these analyses can then be divided to obtain the AI of a section of code, as evidenced in Equation 3.1.

$$AI_{(evaluated\ with\ VTune\ and\ SDE)} = \frac{FLOPs_{(computed\ with\ SDE)}}{Memory\ accesses_{(measured\ with\ VTune)}} . \tag{3.1}$$

In the following sections, we will introduce the two tools and explore how they can work together to achieve this outcome.

3.1.1 VTune - Introduction and Features Overview

VTune [21] is a software profiler developed by Intel that enables optimization of application performance and system configuration for HPC, cloud, IoT, storage, and more. It also allows to tune code running on GPUs.

VTune analyses support both Intel and AMD 64-bit architectures, with exception of Hardware Event-Based Sampling (EBS) analysis that requires an Intel processor for collection. VTune supports Linux, Windows and MacOS. It allows profiling of code written in multiple programming languages, such as C, C++, Python, Java, Fortran and more [23]. Data collection can be limited to specific sections or files by instrumenting the code using the Intel Instrumentation and Tracing Technology (ITT) APIs. These APIs are part of the "oneAPI" bundle which includes also VTune and other Intel profiling tools and will be detailed in Section 3.4.1.

To take full advantage of the VTune features, it is necessary to install Intel proprietary sampling drivers during its setup, which are also mandatory for exploiting EBS analysis. This operation requires root access to the system. A driverless sampling data collection is also available, but it has several analysis limitation for users without root privileges.

Intel VTune features a Graphical User Interface (GUI) that provides a visual representation of data and information, making it easier to study complex analysis. The setup phase is divided into three parts: *WHERE*, *WHAT* and *HOW*, simplifying the process of starting a new analysis.

A screenshot of the GUI can be found in Figure 3.1, in which numbered red circles have been inserted to help us to better explain its use. In particular:

1. *Project Navigator* can be used to move between the folders of previous analysis performed;
2. *WHERE* section allows to select the analysis target system by choosing between a local or a remote device;
3. *WHAT* section manages the path and the parameters of the application to be profiled;
4. *HOW* section allows to chose between different pre-configured analysis. Among the macro-categories, it is possible to find:
   - *Performance Snapshot*, which provides an overview of issues affecting application performance;
   - *Algorithm*, used to detect and tune algorithms anomalies;
   - *Microarchitecture*, helpful to identify hardware bottlenecks;
   - *Accelerators*, which analyses GPU kernel execution.

   Additionally, by clicking on the plus sign button circle in red, you can create customized analyses starting from the existing ones, modifying the execution parameters. This also makes it possible to enable or disable specific EBS to be read during the collection;

5. At the bottom right of the screenshot, you can finally start the analysis. It is also possible to begin a collection in a "paused" mode, where the application execution is started but the data collection deferred until activated by markers in the instrumented code, using the ITT APIs.

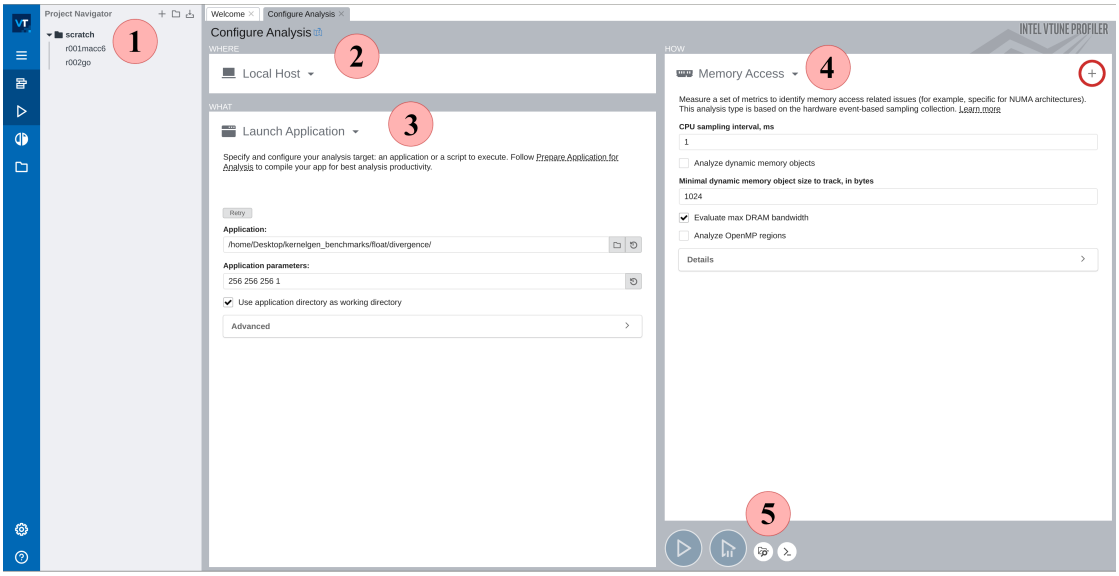


Figure 3.1: Screenshot of Intel VTune homepage to perform a new analysis.

Once completed, the results of the analysis will be saved in the folder of the Project Navigator section and opened in a new screen. More information can be found in the tool manual [22].

VTune also supports terminal usage via a Command Line Interface (CLI), which allows more experienced users to start the analysis quickly and in a personalized way. Results can be saved in different formats to be lately processed or opened in the GUI.

3.1.2 Evaluating Memory Accesses with VTune

To evaluate memory accesses with VTune, the first step is to create a customized analysis. This is mandatory if the evaluation will be carried out via the CLI and you want to limit the data collection to algorithms of interest only. Refer to the Section 3.4.1 of this chapter to learn how to instrument the code and use the ITT APIs.

To create a custom analysis, select the "Memory Access" option under the Microarchitecture section in the "HOW" tab from the GUI, and click on the plus symbol to customize the pre-configured analysis.

From the section that will appear, similar to the one showed in Figure 3.2, it is necessary to:

1. Give the analysis a custom name, to be used from the command line later;
2. Select the option "Analyze user tasks, events and counters";
3. Make sure that, in the EBS related section, the Uncore Events "UNC_IMC_DRAM_DATA_READS" and "UNC_IMC_DRAM_DATA_WRITES" are selected.

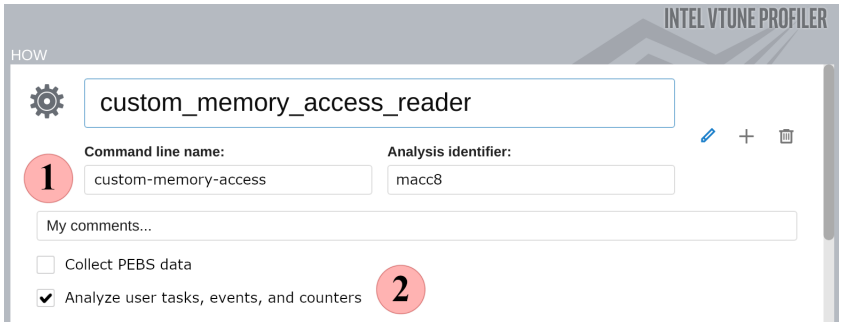


Figure 3.2: Section to create a custom analysis, starting from a predefined one, for Intel VTune.

"Uncore Events" is a term used by Intel to describe events that occurs in the portions of a microprocessor that are not part of the core, such as the ones related to the L3 cache and the thunderbolt or memory controller. As reported on *Monitoring Integrated Memory Controller Request [...] in Intel Core processors* [26], in order to monitor memory traffic metrics, the analysis must read the following values of the Integrated Memory Controller (IMC) counters, which are part of the Uncore Events:

- *UNC_IMC_DRAM_DATA_READS*: "counts every read (RdCAS) issued by the Memory Controller to DRAM (sum of all channels)"
- *UNC_IMC_DRAM_DATA_WRITES*: "counts every write (WrCAS) issued by the Memory Controller to DRAM (sum of all channels)".

Starting from the two values corresponding to these metrics, in order to obtain the number of bytes read and written from the memory, we need to add them and and multiply the result by 64, since all requests result in 64-byte data transfer from the DRAM. The resulting formula is shown in Equation 3.2.

$$\begin{aligned} Memory\ Access_{(Bytes\ evaluated\ with\ VTune)} = \\ (UNC\_IMC\_DRAM\_DATA\_READS + \\ UNC\_IMC\_DRAM\_DATA\_WRITES) \cdot 64. \end{aligned} \tag{3.2}$$

Once configured the analysis and figured out how to interpret the results, in order to carry out the assessment we moved to the CLI for pratical reasons. On an already compiled application, in the example *app.o*, execute on the terminal the command in Listing 3.1.

Listing 3.1 Intel VTune analysis to collect memory accesses via terminal.

```
$vtune -collect custom-memory-access -start-paused -finalization-mode=
    full -data-limit=0 -r ./results -- app.o
```

Where, in Listing 3.1:

- *-collect <string>*: run the specified analysis, where "custom_memory_access" is the name of the custom one we created before;
- *-start-paused*: starts the application but postpones data collection only when it will encounters the ITT APIs in the code, to limit the collection to algorithms of interest only;
- *-finalization-mode = <string>*: when "full" mode is specified, the finalization on the sampled data produces the most accurate results, but takes more time to complete;
- *-r <path>*: specifies the folder where to save the analysis results.

After data collection, the tool will print the analysis report on the terminal screen. The two memory traffic metrics, as introduced before, can be read and then manipulated from the "Uncore Event summary" section of the results.

3.1.3 Use SDE to Measure Floating-Point Operations

Intel Software Development Emulator [20], or SDE, is a command line tool developed by Intel with the main goal of executing applications that contain new instruction sets on a systems that don't support them. It is built upon "PIN", a dynamic binary instrumentation framework, to control the execution of an application, examining each instruction and evaluating if it should be emulated or not. For example, it can emulate the Intel Advanced Vector Extensions 512 (Intel AVX-512) instructions on system that don't support them, helping developers to gain familiarity with the vendor upcoming instruction set extensions.

SDE is available for Windows and Linux operating systems and supports only Intel processors.

SDE comes with several useful features, in addition to the emulator, like the "Mix Histogramming" tool. It allows, among other analysis, to evaluate the number of instructions executed, their length, category, the most frequent ones, and also the number of floating-point operations (FLOPs) computed.

To use the Mix tool for FLOPs collection on an application, run the following command from terminal:

Listing 3.2 Intel SDE Mix analysis to evaluate FLOPs.

```
$sde64 -skl -iform 1 -omix results -start_ssc_mark 111:repeat -
    stop_ssc_mark 222:repeat -- app.o
```

where *app.o* is the example name of an executable and:

- *-slk*: identifies the architecture to run the analysis on (e.g. -knl for Knights Landing processors, -hsw for Haswell and -ckx for Cascade Lake);
- *-iform*: used to specify the output format for the instruction trace generated, in this case the ISA format;
- *-omix <string>*: runs the Mix Histogramming tool and specifies the output file, for example "results";
- *-start_ssc_mark <string>* and *-stop_ssc_mark <string>*: allows to trace only certain sections of code, as explained in the section 3.4.1 about the ITT APIs. If not specified, it will trace data for the entire execution.

After the collection is completed, a report is generated containing the correspondence between the evaluated parameters and the detected values.

To compute the number of FLOPs executed by the application, open the results file and follow these steps [17]:

1. Locate the section of the report titled "EMIT_GLOBAL_DYNAMIC_STATS";
2. To calculate single-precision FLOPs, multiply each value corresponding to the parameters starting with "elements_fp_single_" by the number in the parameter name that appears after the underscore. It represents the number of elements that have been processed at once thanks to vectorization.
   If you are interested in double-precision FLOPs, consider "elements_fp_double_" instead;
3. Add together all of the values obtained to calculate the number of floating point operations. The equation to refer is therefore:

$$FLOPs_{(computed\ with\ SDE)} = \sum_{X=1}^{16} elements\_fp\_single\_X \cdot X. \tag{3.3}$$

To make the explanation more understandable, we can consider the following possible results, which are extrapolated from an example report:

Listing 3.3 Report section example that displays results useful for calculating FLOPs with Intel SDE.

```
# EMIT_GLOBAL_DYNAMIC_STATS
#
# $global-dynamic-counts
#
#       iform                                   count
                                  [...]
*elements_fp_single_1                        234234231
*elements_fp_single_2                          8798747
*elements_fp_single_4                            23453
```

To compute the overall single-precision floating-point operations (FLOPs), apply the formula in the equation 3.3:

$$FLOPs_{(for\ listing\ 3.3)} = (234234231 \cdot 1) + (8798747 \cdot 2) + (23453 \cdot 4) = \\ 287026713 \approx 0.287\ GFLOPs \tag{3.4}$$

A simple script can be used to parse the output file and compute the operation to evaluate FLOPs multiple times with little effort.

3.2 RRZE LIKWID

LIKWID [45], acronym of "Like I Knew What I'm Doing", is a toolsuite for performance analysis of HPC systems. It provides a set of command-line tools and libraries that can be used to collect and analyse various types of data, including monitoring of CPU, memory and energy consumption counters, to evaluate and optimize application efficiency.

LIKWID is developed by *Regionales Rechenzentrum Erlangen* (RRZE), the regional data center of the Friedrich-Alexander University (FAU) in Germany, under an open source license. It works on Intel, AMD, ARMv8 and POWER9 processors but is only available on Linux operating systems.

The suite contains eleven tools, each with its own terminal command that starts with "likwid-". Among them we can find the following [47]:

- *likwid-topology*: query multicore or multisocket architecture to get information about the thread and cache topology. Useful to optimize resource usage in parallel code;
- *likwid-perfctr*: allows to read the hardware performance counters of a processor while an application is running. It also elaborates some metrics to propose a derived one in the report, with standard or simplified name such as "Memory data volume [GBytes]", for less experienced users. It can be used to collect data about an entire execution or it allows the use of proprietary markers to delimit sections of code to be monitored;
- *likwid-pin*: out-of-the-box, it allows to pin processes to specific CPU cores for threaded applications without requiring any changes the source code;
- *likwid-powermeter*: reads the RAPL (*Running Average Power Limit*) counters and queries Turbo mode steps for measuring power consumption of processors and memory.

Monitoring and optimizing code for specific hardware requires in-depth knowledge of the machine. LIKWID, with its tools, allows a higher level of abstraction to simplify this process for the developer.

For example, before carrying out a data collection with "likwid-perfctr", LIKWID automatically identifies the processor architecture of the machine and detects which counters can be accessed. It also proposes sets of related metrics, called "Performance Groups",

put together by using the raw events available and that could be monitored during an application execution.

By executing "likwid-perfctr -a" on the command line, it will return the list of all Performance Groups available on the architecture of the machine used, as showed in Listing 3.4.

Listing 3.4 List of available Performance Groups, on an Intel Skylake architecture, using "likwid-perfctr"

```
$likwid-perfctr -a
    BRANCH  Branch prediction miss rate/ratio
    CACHES  Some data from the CBOXes
     CLOCK  Power and Energy consumption
      DATA  Load to store ratio
    ENERGY  Power and Energy consumption
    [...]
```

3.2.1 Measuring Arithmetic Intensity with LIKWID

To evaluate the Arithmetic Intensity (AI) of algorithms using LIKWID, we will rely on the "likwid-perfctr" tool. It allows limiting data collection to a determined section of code using the proprietary Marker APIs for supported programming languages, as explained in Section 3.4.2. If used as wrapper, it supports monitoring any application, regardless of the programming language used.

"likwid-perfctr" [12] is a tool that allows to read the hardware performance counters of a machine and to process the values obtained to calculate higher level metrics. Since the suite is open source, it is possible to study the functioning of the tool in more detail. We will focus on machines equipped with Intel processors, as we used one in our analyses.

Evaluate the hardware performance counters of an architecture means read its Model Specific Registers (MSR). They are processor-specific hardware registers also used to track "events": a situation that occurs and is intended by the processor designer to be measured, such as a cache miss. To interact with these registers is required to have root permissions or use a driver that interacts with them for you.

Also, the paranoid value of the Linux system should be set to "0" to allows measurement of the whole CPU and Uncore Events. Its value can be checked via terminal with the command:

Listing 3.5 Check the paranoid value on a Linux system.

```
$cat /proc/sys/kernel/perf_event_paranoid
```

"likwid-perfctr" uses the Linux "msr module" to read and write on those registers from the user space [47], avoiding the developer to have an in-depth knowledge of the architecture. LIKWID, for each processor architecture supported, holds a correspondence between the hardware counter name to be evaluated and the address of the register to read.

To calculate the Arithmetic Intensity, which is a derived metric, LIKWID must first obtain the FLOPs and memory accesses for a section of code. To study how those metrics are computed, we can take a look to the "Performance Groups" of a processor architecture. It is a set of files provided by LIKWID holding the correspondence between the counters to be evaluated and the events related to a specific analysis. It also contains their derived metrics, like AI.

Taking as an example a machine equipped with the Intel Skylake architecture, the Performance Group to evaluate single-point arithmetic and main memory performance is called "MEM_SP" [44]. Table 3.1 shows the counter useful for the AI analysis and their relative name of the event to be measured. Subsequently the events are combined together to obtain higher level metrics.

Table 3.1: Correspondence between the event name and the counter name for the "MEM_SP" Performance Group of the Intel Skylake architecture.

Counter Name	Event Name
PCM0	FP_ARITH_INST_RETIRED_128B_PACKED_SINGLE
PCM1	FP_ARITH_INST_RETIRED_SCALAR_SINGLE
PCM2	FP_ARITH_INST_RETIRED_256B_PACKED_SINGLE
MBOX0C1	DRAM_READS
MBOX0C2	DRAM_WRITES

To evaluate the FLOPs the following computation is performed on the sets of events:

$$\begin{aligned} FLOPs_{(using\ LIKWID\ on\ Skylake\ CPU)} = \\ (FP\_ARITH\_INST\_RETIRED\_128B\_PACKED\_SINGLE \cdot 4 + \\ FP\_ARITH\_INST\_RETIRED\_SCALAR\_SINGLE + \\ FP\_ARITH\_INST\_RETIRED\_256B\_PACKED\_SINGLE \cdot 8)\ . \end{aligned} \tag{3.5}$$

and for bytes of accesses in memory, evaluated considering the Uncore Events from the Integrated Memory Controller (IMC) counters:

$$\begin{aligned} Bytes\ of\ Memory\ Access_{(with\ LIKWID\ for\ Skylake\ CPU)} = \\ (DRAM\_READS + DRAM\_WRITES) \cdot 64\ . \end{aligned} \tag{3.6}$$

Finally, the tools divides the FLOPs computed by the cores and the memory data volume of the whole socket to propose the Arithmetic Intensity, as a ready-made metric.

Although the theory behind it may not seem trivial, calculating AI with LIKWID is very simple. From the command line, to evaluate the group "MEM_SP" for an Intel processor with Skylake architecture, you can run the following:

Listing 3.6 LIKWID analysis to collect Arithmetic Intensity for Intel Skylake architecture.

```
$likwid-perfctr -c 0 -g MEM_SP -m ./app.o
```

where:

- *-c <list>*: requires to list the processor ids to measure;
- *-g <string>*: specifies the Performance Group for which to collect data. It is unique to the processor architecture and the command "likwid-perfctr -a" can be used to see which ones are available for your architecture;
- *-m*: enable the Marker APIs inside the code to limit the data collection, as explained in Section 3.4.2.

This allows for the direct retrieval of the value of Arithmetic Intensity, referred to as Operational Intensity from the tool in the report. The counters and events used in the analysis along with all the other values evaluated from the Performance Group are also printed. The results of an example analysis are showed in Listing 3.7.

Listing 3.7 Example of the report obtained through a LIKWID analsys to obtain the Arithmetic Intensity of a section of code.

```
$likwid-perfctr -c 0 -g MEM_SP -m ./app.o
--------------------------------------------------------------------------------
Region read_stats, Group 1: MEM_SP
                                    [...]
+-----------------------------------------+---------+------------+
|                   Event                 | Counter | HWThread 0 |
+-----------------------------------------+---------+------------+
|                                    [...]                        |
| FP_ARITH_INST_RETIRED_128B_PACKED_SINGLE |   PMC0  |          0 |
|    FP_ARITH_INST_RETIRED_SCALAR_SINGLE   |   PMC1  |   53385470 |
| FP_ARITH_INST_RETIRED_256B_PACKED_SINGLE |   PMC2  |          0 |
|                DRAM_READS                | MBOX0C1 |    5049095 |
|                DRAM_WRITES               | MBOX0C2 |     334272 |
+-----------------------------------------+---------+------------+

+-----------------------------------+------------+
|               Metric              | HWThread 0 |
+-----------------------------------+------------+
|        Runtime (RDTSC) [s]        |     0.0174 |
|            SP [MFLOP/s]           |  3064.2843 |
|                                [...]          |
|    Memory data volume [GBytes]    |     0.3445 |
|       Operational intensity       |     0.1549 |
+-----------------------------------+------------+
```

3.3 Intel Advisor

Intel Advisor [18] is an analysis tool for improving applications performance by evaluating the efficiency of threading, vectorization, memory usage and GPU offloading of algorithms. It is available for Linux, Windows and MacOS operating systems and supports C, C++, FORTRAN, SYCL, OpenMP, OpenCL and Python code. Performing its analyses does not require privileged execution permissions or the installation of any special drivers.

Advisor consists of a set of tools that provide recommendations for optimizing code by identifying where to make improvements:

- *Vectorization and Code Insights*: identifies and recommends ways to improve loops that can be optimized due to their high impact on performance or low vectorization;
- *CPU and GPU Roofline Insights*: performs Roofline analysis of the application, relating the performance of the algorithms to that of the machine in use. It allows to identify and resolve execution bottlenecks;
- *Offload Modeling*: identifies GPU offloading opportunities in the source code, evaluating also the data transfer and speed up projection of the application if accelerated. It reports the limiting factors of a loop if is not advantageous to offload;
- *Threading*: check and tune threading design options, finding performance issues. It also creates a projection of the analysis on a system with more cores count.

All the evaluations can be performed through the use of the GUI or via the system command line, as in Listing 3.8.

Listing 3.8 Perform an Intel Advisor analysis via CLI.

```
1 $advisor --collect=<string> --project-dir ./results -- app.o
```

Where:

- *–collect=<string>*: runs the specified type of analysis. Each tool has a string that identifies it;
- *–project-dir <path>*: path in the system where to save the results of the data collection.

The application under examination should be compiled with the debug flag for more in-depth results. It is also possible to customize or modify how the analysis is performed by including other flags, which can be found in the Advisor manual [19].

Advisor stands out from the other tools we have considered due its "Offload Modeling" tool, which analyses code to identify potential candidates for GPU offloading. This mode is particularly relevant to our goals and we will explore it in the next section before conducting more extensive testing in Chapter 5 of this thesis. Additionally, the Offload Modeling serves as a starting point for evaluating the Arithmetic Intensity of an algorithm using Advisor.

3.3.1 Modelling GPU Offload Opportunities

The "Offload Modeling" mode of Advisor enables various analyses related to hardware acceleration, including performance estimation for running GPU-designed code on different GPU models, among those produced by Intel. Additionally, for CPU-designed code, it can estimate data transfer between the host and device, and identify offloading opportunities by estimating potential speed up of loops. We will focus on the latter, studying how the tool can be used to identify algorithms that can benefit from GPU offloading.

This analysis requires no instrumentation of the code or knowledge about the code base, since functions detection is totally automatic. However, it is possible to limit the files or sections to be evaluated using the ITT APIs, as explained in Section 3.4.1.

Carrying out the analysis does not require having a GPU installed in your system, since the tool does not relies on hardware counters but models the behaviour of a GPU. However, the only GPUs available in Advisor are the one produced by Intel and the default one when executing an analysis is the "Intel Arc Graphics" with 512 vector engines. Even if it is the only device brand present, the tool allows to tune the hardware parameter of the GPUs available, such as frequency and DRAM size, to perform an estimation on a customized device according to the developer's needs.

The Offload Modeling analysis can be performed via the GUI of Advisor or the system terminal, by using the parameter "–collect=offload" to start the data collection, as showed in Listing 3.8. Once the modelling is started, it performs the following steps on the application under examination [19]:

1. *Survey analysis*: first run of the application and collection of key parameters on the baseline CPU platform;

2. *Characterization analysis*: identification of the number of times functions in code are invoked, the number of floating-point and integer operations and estimation of the memory traffic between the host and the device;

3. *Performance Modeling*: estimation of the speed up on a modeled GPU, considering the improvement for loops that could be offloaded on the GPU.

By opening the results using the user interface, and selecting the "Accelerated Regions" tab, we will find ourselves in front of a screen similar to the one in Figure 3.3. Here we can find:

1. *Summary of the estimated offload characteristics*: here is reported the total speed up that the application would have if exploiting also the GPU, the number of loops and function suggested for offloading and the fraction of code accelerated;

2. *List of loops/functions*: functions suggested for offloading. The top not-offloaded function are shown in light gray, reporting also the reason why they have not been selected;

3. *Metrics report*: provides detailed metrics of the identified loops. By scrolling horizontally, you can find information such as the time required for the function to run

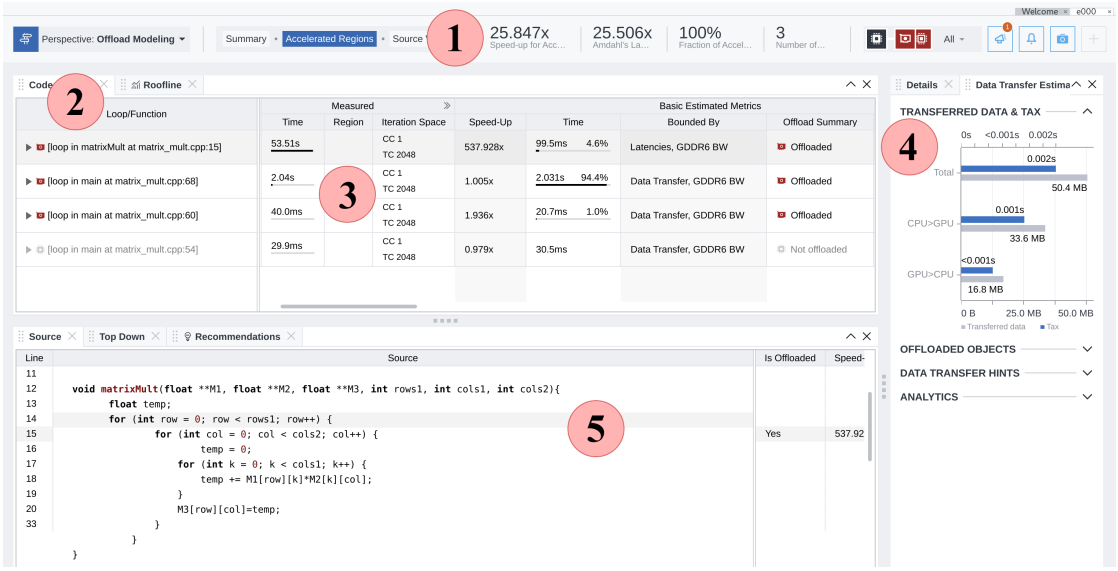


Figure 3.3: Screenshot of Intel Advisor results for the Performance Modeling analysis.

on the CPU of the machine, the estimated time for the function to execute on the modeled GPU, the computed speed up, the number of FLOPs, the estimated data transfer and factors that may limit execution, such as latencies, data exchange and DRAM bandwidth;

4. *Transferred data*: estimated data transfer H→D and D→H for the selected loop. It takes also into account data reuse: if the function is executed multiple times working on the same data, these are considered already held on the GPU memory, without the need for numerous transfers;

5. *Source code*: snippet from the file of the code identified for offloading.

To conduct this analysis and identify algorithms that can benefit from running on the GPU, Advisor measures CPU execution time on baseline platform. This value is compared to the time it would take on a modelled GPU, evaluated as follows[2]:

$$t_{region} = max(t_{compute}, t_{memory\ subsystem}) + t_{data\ transfer\ tax} + t_{kernel\ launch}. \tag{3.7}$$

where:

- $t_{compute}$: is the estimated time assuming bound exclusively by compute;
- $t_{memory\ subsystem}$: is the estimated time assuming bound exclusively by the memory subsystem;
- $t_{data\ transfer}$: is the time required for the data transfers H→D and D→H;
- $t_{kernel\ launch}$: is the requested kernel launch time.

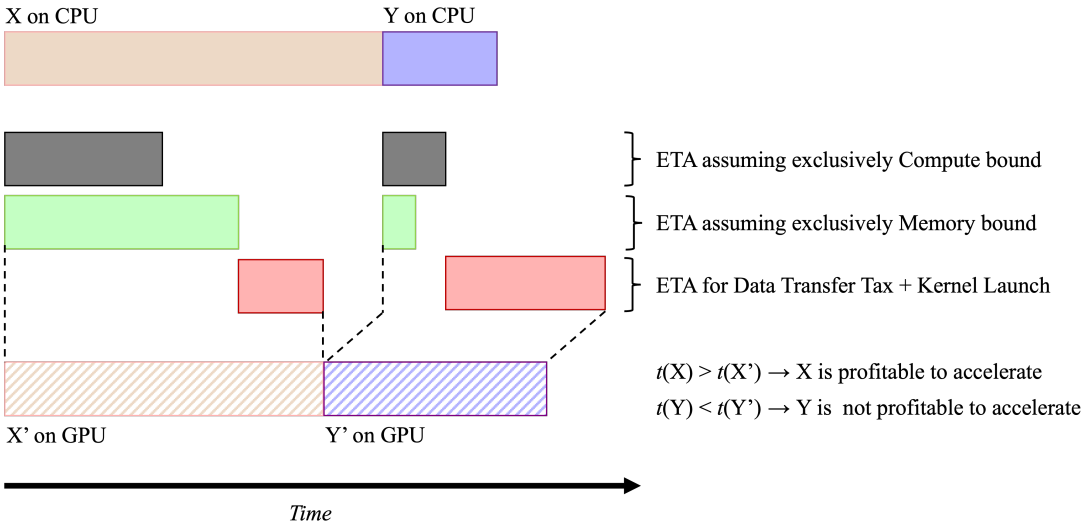


Figure 3.4: The mechanism behind Advisor suggestions for GPU offloading. *Source: "Modeling Heterogeneous Computing Performance with Offload Advisor"* [2].

A region is considered profitable and recommended for offloading if the ratio between the time required on the CPU and the time modelled for the GPU is greater than 1. Figure 3.4 provides a graphical representation of this concept.

The results obtained will therefore not be universal, since changing the underlying hardware of the machine or the GPUs to be modelled will also change the outcome of the analysis.

3.3.2 Measuring Arithmetic Intensity with Advisor

To measure Arithmetic Intensity (AI) of an algorithm using Advisor, we need to use the "Offload Modeling" analysis introduced in the previous section, to which is possible to add some flags that can improve the collection. The command to be run in the CLI is the following:

Listing 3.9 Arithmetic Intensity collection using Intel Advisor analysis via CLI.

```
1 $advisor --collect=offload --loop-filter-threshold=0 --data-transfer=
    full --project-dir ./results -- app.o
```

where:

- *–collect=offload*: performs the Offload Modeling analysis on the specified application;
- *–loop-filter-threshold=0*: evaluate all loops in the code, even those that have a very low running time;

- *–data-transfer=full*: model data transfer between host and device with high details, identifying also where data can be potentially reused. Adds significant overhead to the analysis;
- *–project-dir <path>*: path in the system where to save the results.

The identification of the loops, including the name of the functions and the line where they are located in the code, is automatic and does not require limiting the areas of interest using markers.

The Advisor screen containing the results is similar to the one in Figure 3.5. From the "Accelerated Regions" tab, after selecting a function of interest and scrolling horizontally in the report, the metrics we are interested in are:

1. *GFLOP*: representing the floating-point operations (FLOPs) computed by the algorithm, expressed as multiple of 10^9 (to not be confused with the "GFLOPS" section, that contains the GFLOPs per second);
2. *Estimated Data Transfer With Reuse*: containing the memory accesses performed, considering also data reuse, in the specified unit of measure. The "Read" value represents the H→D data movement, while the "Write" value the D→H one.

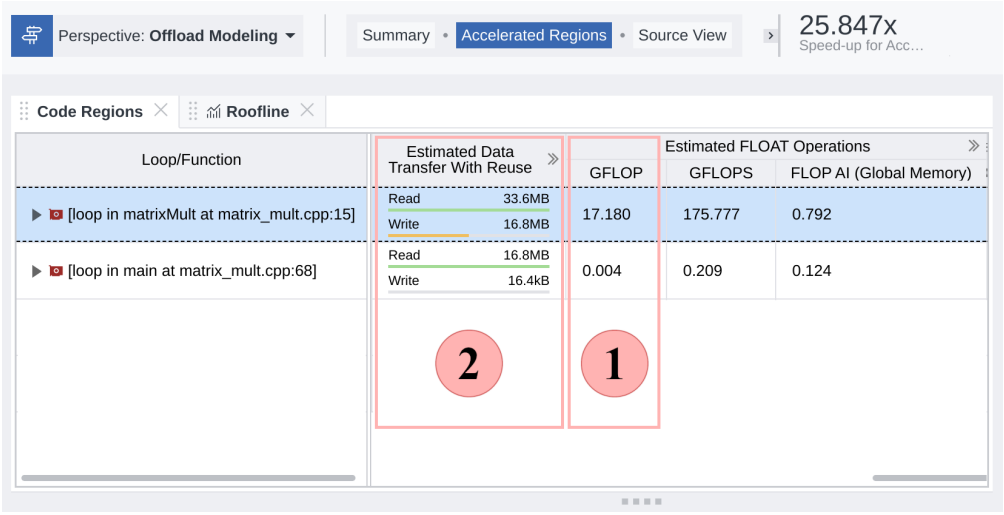


Figure 3.5: Screenshot of Intel Advisor results showing the metrics useful to obtain Arithmetic Intensity of a function.

The AI can be therefore obtained as:

$$AI_{(Advisor\ analysis)} = \frac{GFLOP}{Estimated\ Data\ Transfer\ With\ Reuse\ Read + Write\ [GB]}. \tag{3.8}$$

3.4 Control and Limit Data Collection

While it may be useful to evaluate certain metrics for the entire application execution, computing the Arithmetic Intensity (AI) of an algorithm to determine if it can benefit from GPU offloading is only profitable if it can be evaluated on delimited portions of code. This section explains how to instrument the code so that the collection of FLOPs and memory accesses by the the tools is limited only to the regions of interest.

3.4.1 Collection Control APIs for Intel Products

The Instrumentation and Tracing Technology (ITT) APIs are used to instrument the code and control the data collection for Intel products. They support C/C++ and FORTRAN applications on Windows, Linux and MacOS. The source code is open source and released on Github [25]. The installation of the ITT APIs occurs simultaneously with that of VTune and Advisor, as part of the same "oneAPI" bundle which includes them and other Intel profiling tools.

When compiling the application, it must be linked to the "libittnotify.a" static library ("-littnotify") and the build system needs to be configured to reach the API headers and libraries by adding [24]:

- *<install_dir>/sdk/include* to the include path;
- *<install_dir>/sdk/lib64* to the libraries path.

Finally, to instrument the code for VTune and Advisor, add the following to every C/C++ source file in which you want to control the data collection:

1. *#include <ittnotify.h>* to the file headers;
2. *___itt_resume()*: to start the collection and record data (note it uses two underscores at the beginning);
3. *___itt_pause()*: to stop recording performance data.

Here is a practical C++ example of how to limit the collection, and later use the "start-paused" analysis to execute the application but wait until the resume markers are found:

Listing 3.10 Example of instrumenting the code to perform an analysis with VTune or Advisor

```
#include <ittnotify.h>  //built with appropriate paths and libraries

__itt_resume(); //resume collection
for (int i = 0; i < M; i++){
    for (int j = 0; j < J; j++){
        someFunction();
    }
}
__itt_pause();  //stop collection
```

To instrument the code for the SDE-performed collection, specify in the analysis configuration with the commands "-start_ssc_mark 111:repeat" and "-stop_ssc_mark 222:repeat" that the data collection is limited. Then, add the following to every C/C++ source file:

1. *#include <ittnotify.h>* to the headers;
2. *___SSC_MARK(0x111)*: to start the collection and record data (note it uses two underscores at the beginning and the number in the brackets for the start and stop commands must be different);
3. *___SSC_MARK(0x222)*: to stop recording performance data.

A C++ example on how to use them, is shown in the sample code snippet below:

Listing 3.11 Example on how to instrument the code to perform an analysis with SDE

```
#include <ittnotify.h>  //built with appropriate paths and libraries

__SSC_MARK(0x111);  //start SDE collection
for (int i = 0; i < M; i++){
    for (int j = 0; j < J; j++){
        someFunction();
    }
}
__SSC_MARK(0x222);  //stop SDE collection
```

3.4.2 Marker APIs for LIKWID

LIKWID has developed the "Marker APIs" for controlling data collection. Through code instrumentation, the "likwid-perfctr" analysis can be conducted on a specific section of code. They are available for C, C++, Julia and FORTRAN.

For C/C++ applications, a set of defines allows to disable or enable the markers at build time. To enable them you need to set "-DLIKWID_PERFMON" as compiler flag along with linking the application to the LIKWID library ("-llikwid"). The build system needs also to be configured to reach the application include and libraries paths.

The instrumentation is done by modifying the source code, which you must have access to, adding in every file for which you want to control the collection the following:

1. *#include <likwid.h>*: to the headers;
2. The set of defines to enable or disable the markers using "-DLIKWID_PERFMON" as compiler flag;
3. *LIKWID_MARKER_INIT*: to initialize the Marker API;
4. *LIKWID_MARKER_REGISTER ("string")*: to register a region to be monitored, with the name contained in the brackets;

5. *LIKWID_MARKER_START("string")*: to start collecting data for the specified region;
6. *LIKWID_MARKER_STOP("string")*: to stop the collection for the named region;
7. *LIKWID_MARKER_CLOSE*: finalizes the Marker API for the "likwid-perfctr" evaluation.

A sample code snippet written in C++, demonstrating the usage of Marker APIs, could look like the following:

Listing 3.12 Marker APIs usage example to conduct an analysis with LIKWID.

```
#include <likwid.h>  //built with appropriate paths and libraries

#ifdef LIKWID_PERFMON
#include <likwid.h>
#else
#define LIKWID_MARKER_INIT
#define LIKWID_MARKER_THREADINIT
#define LIKWID_MARKER_SWITCH
#define LIKWID_MARKER_REGISTER(regionTag)
#define LIKWID_MARKER_START(regionTag)
#define LIKWID_MARKER_STOP(regionTag)
#define LIKWID_MARKER_CLOSE
#define LIKWID_MARKER_GET(regionTag, nevents, events, time, count)
#endif

int main(int argc, char **argv) {
    LIKWID_MARKER_INIT;

    LIKWID_MARKER_REGISTER("read_stats");
    LIKWID_MARKER_START("read_stats");
    for (int i = 0; i < M; i++){
        for (int j = 0; j < J; j++){
            someFunction();
        }
    }
    LIKWID_MARKER_STOP("read_stats");

    LIKWID_MARKER_CLOSE;
    return 0;
}
```

The Marker APIs allows also to define multiple regions to be monitored using different names, since "likwid-perfctr" will create a report for each of them.

Chapter 4

Experimental Analysis for Arithmetic Intensity Evaluation

To effectively identify code areas that can benefit from GPU execution, we will evaluate the Arithmetic Intensity (AI) of functions using the tools introduced in the previous chapter. We want to assess whether the tools can be successfully used for our purpose and for this reason we selected twelve benchmarks, part of the "KernelGen Test Suite" [29]. Each benchmark includes a function that performs floating-point operations and can benefit from parallel execution. The code can be compiled by selecting a compiler and setting custom flags in the makefile, including the choice whether to use single-precision or double-precision floating-point format for numbers. The KernelGen Test Suite is written in C language and has a counterpart in CUDA, a GPU programming language. The benchmarks were initially developed by Dr. Mikushin et al. to compare the performance of "KernelGen Compiler" [30], a C and FORTRAN compiler for NVIDIA GPUs, with other compilers such as CAPS Enterprise Compiler, NVIDIA CUDA Compiler, GCC compiler and Intel MIC Compiler. These comparisons were made by targeting host CPUs, NVIDIA GPUs and Intel Xeon Phi accelerators.

For each benchmark in the KernelGen Test Suite, we begun by studying the algorithmic implementation, trying to analyze the implementation complexity. Next, we computed expected FLOPs and memory accesses (needed to AI evaluation) using the algorithmic analysis. This evaluation is performed a priori to avoid biases on the analysis given the benchmarking results. Then we measured these values using Intel VTune with SDE, RRZE LIKWID and Intel Advisor to compare the results. Finally, we validated the obtained results by running each benchmark on the GPU using the CUDA implementation.

4.1 Environment and Tools Setup

To conduct the measurement, we are using a machine with exclusive access and not running on shared hardware. This is necessary since some tools require special permissions to be installed and do not support a shared environment. The system employed is equipped with 16 GB of DDR4 RAM, an Intel Core i7-7820HQ CPU, whose characteristics are listed in Table 4.1, and a NVIDIA Quadro M1200 GPU, with proprieties listed in Table 4.2. The operating system, based on Linux, is CentOS 7 running 3.10.0-1160.76.1.el7.x86_64 version of the kernel.

Table 4.1: Technical features of the machine CPU, used to carry out the measurements.

CPU Model	Intel Core i7-7820HQ
Architecture	x86 64-bit
CPUs	8
Threads per Core	2
Cores per Socket	4
Max Clock [GHz]	3.9
Min Clock [GHz]	0.8
L1d Cache [kB]	32
L1i Cache [kB]	32
L2 Cache [kB]	256
L3 Cache [kB]	8192

Table 4.2: Technical features of the GPU used in the validation phase.

GPU Model	NVIDIA Quadro M1200
Architecture	Maxwell
Boost Clock [MHz]	1148
Base Clock [MHz]	991
SM Count	5
L1 Cache [kB]	64 (per SM)
L2 Cache [MB]	2
Memory Type	GDDR5
Memory Size [GB]	4
Bandwidth [GB/s]	80.19

To evaluate the benchmarks with LIKWID, we compiled the source code using the version 11.2.1 of the GCC compiler. Instead, to perform the analyses with VTune, SDE, and Advisor, we used the Intel oneAPI ICC compiler version 2021.7.0, which is distributed in the same bundle of the tools and is specifically designed for HPC applications. Although VTune and Advisor can analyze any binary regardless of the compiler used, we chose this compiler to ensure maximum compatibility with the tools.

4.1.1 VTune and SDE Configuration

To perform the measurements using Intel VTune for the memory accesses and Intel SDE for FLOPs evaluation, we instrumented the code using the Instrumentation and Tracing Technology (ITT) APIs, wrapping the function that performs the computation of our interest. This allows to disable and then re-enable the tracing only for certain portions of code, to be sure of not collecting more data than necessary, as explained in Section 3.4.1.

Next, we compiled the benchmarks using Intel proprietary compiler as it follows:

Listing 4.1 Compile command for the Kernelgen Test Suite to be evaluated with VTune and SDE.

```
icc -g -O3 -std=c99 -L$(ITT_library) -I$(ITT_include) benchmark.c -o
    benchmark.o -littnotify
```

where:

- *-g*: enables debugging information;
- *-O3*: turns on an high grade of compile optimizations;
- *-std=c99*: lets the compiler know that the C99 standard version is used;
- *ITT_library* and *ITT_include*: are the paths to the ITT API libraries and include, used to limit the data collection to the region of interest;
- *-littnotify*: is the name of the ITT library;

Once we obtained the executable, the VTune analysis to collect the memory accesses for each benchmark can be initiated via the terminal command in Listing 4.2.

Listing 4.2 VTune analysis on the KernelGen Test Suite to collect memory accesses.

```
vtune -collect custom-memory-access -start-paused -finalization-mode=
    full -data-limit=0 -r ./result -- benchmark.o 512 512 512 1
```

Where "512 512 512 1" represent function arguments, which meaning depends on the benchmark.

Afterwards, on the same executable, we can measure the floating-point operations (FLOPs) using the SDE command from terminal showed in Listing 4.3.

Listing 4.3 Intel SDE terminal command to collect FLOPs from the KernelGen Test Suite.

```
sde64 -skl -iform 1 -omix result.sde -start_ssc_mark 111:repeat -
    stop_ssc_mark 222:repeat -- benchmark.o
```

The version of VTune used for our evaluations is 22.4.0 and the version of SDE is 9.14. In-depth information about the terminal commands to start the analyses and how the results can be collected can be found in Section 3.1.2 for VTune and Section 3.1.3 for SDE.

4.1.2 LIKWID Configuration

Before conducting the measurements using RRZE LIKWID, we instrumented each benchmark using its Marker API to limit the data collection only to the function of interest. This can be achieved as explained in Section 3.4.2.

We compiled the benchmarks using the GCC compiler, as it follows:

Listing 4.4 Compile command for the Kernelgen Test Suite to be evaluated with VTune and SDE.

```
gcc -g -O3 -std=c99 -DLIKWID_PERFMON -L$LIKWID_library -I$LIKWID_include
    benchmark.o -llikwid
```

where:

- *-g*: enables debugging information;
- *-O3*: turns on an high grade of compile optimizations;
- *-std=c99*: lets the compiler know that the C99 standard version is used;
- *-DLIKWID_PERFMON*: enables the Marker APIs;
- *LIKWID_library* and *LIKWID_include*: are the paths in the system to the Marker APIs libraries and include, used to limit the data collection to the region of interest;
- *-llikwid*: indicates the LIKWID library.

Obtained the executable, each analysis using LIKWID to collect the Arithmetic Intensity can be initiated with the terminal command in Listing 4.5, where "512 512 512 1" represent an example input dimensions: the first three number sequences are the dimensions of the data structures to be allocated and the last is the number of times the benchmark execution should be repeated, in this case always one. The version of likwid-perfctr we used is the 5.2.0.

Listing 4.5 LIKWID terminal command to collect AI of the benchmarks.

```
likwid-perfctr -c 0 -g MEM_SP -m benchmark.o 512 512 512 1
```

More information on how to use LIKWID can be found in Section 3.2.1.

4.1.3 Advisor Configuration

Performing the analyses with Intel Advisor does not require any instrumentation of the code since the tool is able to automatically detect functions to evaluate their AI. The Kernelgen Test Suite was compiled using the following command with the Intel C compiler:

Listing 4.6 Compile command for the Kernelgen Test Suite to be evaluated with Advisor.

```
icc -g -O3 -std=c99 benchmark.c -o benchmark.o
```

where:

- *-g*: enables debugging information;
- *-O3*: turns on an high grade of compile optimizations;
- *-std=c99*: lets the compiler know that the C99 standard version is used.

Once compiled, the analysis for each benchmark can be launched by running from the command line:

Listing 4.7 Advisor terminal command to collect AI of the benchmarks.

```
advisor --collect=offload --loop-filter-threshold=0 --data-transfer=full
    --project-dir ./benchmark_results -- benchmark.o 512 512 512 1
```

The version of Advisor we used is the 2022.3. More information about Advisor analysis to evaluate AI and how the results can be collected can be found in Section 3.3.2.

4.2 Evaluation of Arithmetic Intensity

This section presents the results of the different analyses. Firstly, we analyzed each benchmark in the KernelGen Test Suite in terms of algorithmic features. Next, we have calculated the Arithmetic Intensity (AI) of each operation manually, starting from the source code, that we call "expected" in the tables containing the results. Finally, the AI was measured with Intel VTune with SDE, RRZE LIKWID and Intel Advisor.

Each benchmark, in order to be executed, requires three or four input parameters: the first two or three are the input dimensions of the data structures to be allocated, depending from the benchmark itself, and the last one always represents the number of times that the main function must be executed, in case multiple runs are needed. Each measurement is repeated ten times, to evaluate also the variance and the standard deviation of the results, and using three different dimensions for each benchmark. To equalize the execution times, we used square data structures with input dimensions of "1024", "2048" and "4096" for benchmarks that require three dimensions, and "128", "256" and "512" for those that require two dimensions. For the purpose of the experiment all the benchmarks have been compiled enabling the single-precision floating point format for numbers (4 bytes).

4.2.1 *divergence* Benchmark: Divergence Operator

The first benchmark we evaluated is called "divergence", and is based on the mathematical divergence operation. In vector differential calculus, the divergence measures the tendency of a vectorial field $\mathbf{F} = F_1\mathbf{i} + F_2\mathbf{j} + F_3\mathbf{k}$ to diverge or converge towards a point in space, and it is indicated with the symbol $\nabla \cdot \mathbf{F}$.

In Cartesian coordinates the operation is defined as the sum of the partial derivatives equations (PDEs) of the components of $\mathbf{F}$ along the axes:

$$div\mathbf{F} = \nabla \cdot \mathbf{F} = \frac{\partial F_1}{\partial x} + \frac{\partial F_2}{\partial y} + \frac{\partial F_3}{\partial z} \ . \tag{4.1}$$

To implement this operation in the discrete field, the algorithm makes use of the finite differences theorem. According to this theorem, the value of the derivative at a certain point of the field is approximated by a weighted sum of the values at the neighboring items, while discarding the first and the last ones, as shown in Equation 4.2. Usually the coefficients like a, b and c, in the finite differences theorem, are generated from the Taylor series expansions.

$$\begin{aligned} \frac{\partial f(x,y,z)}{\partial x} &= a(f_{x+1,y,z} - f_{x-1,y,z}) \\ \frac{\partial f(x,y,z)}{\partial y} &= b(f_{x,y+1,z} - f_{x,y-1,z}) \\ \frac{\partial f(x,y,z)}{\partial z} &= c(f_{x,y,z+1} - f_{x,y,z-1}). \end{aligned} \tag{4.2}$$

Since the algorithm performs a large number of independent operations to compute the differences between neighboring grid points, it is well-suited to be parallelized on a GPU.

The number of FLOPs and memory accesses, expressed as bytes, expected by the operation are the following:

- $FLOPS_{divergence} : 8 \cdot (x-2) \cdot (y-2) \cdot (z-2)$
- $Bytes_{divergence} : 4 \cdot (x \cdot y \cdot z) \cdot sizeof(type)$

The Arithmetic Intensity of the algorithm that implements the divergence operation can be computed as:

$$AI_{divergence} = \frac{8 \cdot (x-2) \cdot (y-2) \cdot (z-2)}{4 \cdot (x \cdot y \cdot z) \cdot sizeof(type)} \ . \tag{4.3}$$

We evaluated the core function of the divergence benchmark using VTune with SDE, LIKWID, and Advisor to measure the performed values of FLOPs, memory accesses, and AI. The benchmark was executed with three different input dimensions: "128 128 128", "256 256 256" and "512 512 512". Each measurement was repeated 10 times to also report the values of variance and standard deviation.

Expected results and tools results are reported in Tables 4.3, 4.4 and 4.5, respectively for each input size.

It is evident that all the tools correctly measured the FLOPs performed by the operation. However, when evaluating memory accesses, VTune and LIKWID overestimated the values since their analysis is based on reads and writes from the main memory of the host. This indicates that their evaluation is not optimized to data offloading, as is it

Table 4.3: Comparison between the results obtained with the tools for the *divergence* benchmark using "128 128 128" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.016	0.034	0.471		
Intel VTune+SDE	0.016	n/a	n/a	n/a	n/a
RRZE LIKWID	0.016	0.073	0.220	0.060	0.004
Intel Advisor	0.016	0.034	0.471	0.000	0.000

Table 4.4: Comparison between the results obtained with the tools for the *divergence* benchmark using "256 256 256" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.131	0.268	0.489		
Intel VTune+SDE	0.131	0.444	0.295	0.137	0.019
RRZE LIKWID	0.131	0.529	0.248	0.055	0.003
Intel Advisor	0.131	0.268	0.489	0.000	0.000

Table 4.5: Comparison between the results obtained with the tools for the *divergence* benchmark using "512 512 512" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	1.061	2.147	0.494		
Intel VTune+SDE	1.061	5.289	0.201	0.058	0.003
RRZE LIKWID	1.061	4.275	0.248	0.049	0.002
Intel Advisor	1.061	2.147	0.494	0.000	0.000

for Advisor. Additionally, during the execution with the smallest input, VTune failed to evaluate the memory accesses due to the function execution speed, which was faster than its sampling interval of 0.1 ms. Finally, unlike the other tools, Advisor shows a variance of zero between the various measurements made, correctly estimating the value of the AI.

4.2.2 *gameoflife* Benchmark: Conway's Game Of Life

The Conway's game of life represents an algorithm that can be applied to grid of elements in which each item represent a cell that may be "dead" or "alive". In this case it consists of a matrix of $x \cdot y$ elements, which can be above or below a threshold, or in other versions just zeros and ones, to represent the state. The algorithm purpose is to generate the next generation of cells following this rules [9]:

1. Any living cell that has less than two living neighbors has no chance of surviving, so dies of under-population;
2. Any living cell that has two or three living neighbors can survive;
3. Any living cell that is surrounded by more than three living neighbors has no chance of surviving, so dies of over-population;
4. Any dead cell that is surrounded by exactly three live cells becomes alive. thanks to cellular reproduction.

By the "neighbors of the cell", where the cell represents an item $a = M[row][column]$ in the matrix, are meant the eight elements starting from $a_{-1-1} = M[row-1][column-1]$ up to $a_{+1+1} = M[row+1][column+1]$ from the location of the element a:

$$Neighbors = \begin{bmatrix} a_{-1-1} & a_{0-1} & a_{-11} \\ a_{-10} & a_{00} & a_{10} \\ a_{11} & a_{01} & a_{11} \end{bmatrix} \tag{4.4}$$

The number of FLOPs and memory accesses, expressed as bytes, expected by the algorithm are the following:

- $FLOPs_{gameoflife} : 14 \cdot (x-2) \cdot (y-1)$
- $Bytes_{gameoflife} : (x \cdot y + x \cdot y) \cdot sizeof(type)$

Therefore, the AI of the Conway's game of life can be defined as:

$$AI_{gameoflife} = \frac{14 \cdot (x-2) \cdot (y-1)}{(x \cdot y + x \cdot y) \cdot sizeof(type)} \ . \tag{4.5}$$

We evaluated the core function of the gameoflife benchmark using VTune with SDE, LIKWID, and Advisor to measure the performed values of FLOPs, memory accesses, and AI. The benchmark was executed with three different input dimensions: "1024 1024", "2048 2048" and "4096 4096". Each measurement was repeated 10 times to also report the values of variance and standard deviation.

Expected results and tools results are reported in Tables 4.6, 4.7 and 4.8, respectively for each input size.

After inspecting the results, we noticed a slight discrepancy in the measured FLOPs between the Intel tools and LIKWID, which is likely due to the different algorithm optimization performed by the different compilers. As for memory accesses, VTune failed to

Table 4.6: Comparison between the results obtained with the tools for the *gameoflife* benchmark using "1024 1024" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.015	0.008	1.875		
Intel VTune+SDE	0.016	n/a	n/a	n/a	n/a
RRZE LIKWID	0.008	0.028	0.299	0.246	0.061
Intel Advisor	0.021	0.012	1.750	0.000	0.000

Table 4.7: Comparison between the results obtained with the tools for the *gameoflife* benchmark using "2048 2048" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.059	0.034	1.735		
Intel VTune+SDE	0.063	0.752	0.084	0.076	0.006
RRZE LIKWID	0.033	0.157	0.212	0.189	0.036
Intel Advisor	0.087	0.050	1.740	0.000	0.000

Table 4.8: Comparison between the results obtained with the tools for the *gameoflife* benchmark using "4096 4096" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.235	0.134	1.754		
Intel VTune+SDE	0.251	0.740	0.339	0.164	0.027
RRZE LIKWID	0.134	0.361	0.371	0.128	0.016
Intel Advisor	0.352	0.201	1.751	0.000	0.000

evaluate them for the "1024 1024" input due to the function execution speed, which was faster than its sampling interval. Advisor demonstrates more accurate modelling of data transfer between the host and device compared to VTune and LIKWID, getting closer to the expected AI value.

4.2.3 *gaussblur* Benchmark: Gaussian Blur

The gaussblur benchmark is based on the operation known as Gaussian Blur, named after the mathematician Carl Friedrich Gauss, primarily used in image processing to reduce details. The final visual output resembles an out-of-focus object, as shown in the example in Figure 4.1.

Figure 4.1: Pictures of the famous Temple Bar in Dublin, the city where I did my internship. The picture on the left shows the original photo, while the picture on the right shows the same image after applying a Gaussian blur filter.

To apply the algorithm, the image can be considered as a matrix of pixels on which a convolution is computed using a second matrix called "Kernel", which represents the filter to apply. The matrix convolution is an high parallelizable operation in which we superimpose the center of the kernel on the n-Th element of the matrix considered. This value is recalculated as the weighted sum of the products of each element of the kernel matrix with the value of the underlying matrix, representing a pixel.

The mathematical formulation of the operation is:

$$g(x,y) = K * f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} K(dx,dy) f(x-dx, y-dy). \tag{4.6}$$

where $f(x,y)$ is the original image, K is the Kernel and $g(x,y)$ is the resulting image. The Gaussian Blur differs from similar operations from the values that make up the Kernel: the pixels nearest the centre of it are given more weight than those far away, as shown in the example 4.7.

$$Kernel = \begin{bmatrix} 0.01 & 0.05 & 0.05 & 0.05 & 0.01 \\ 0.05 & 0.11 & 0.11 & 0.11 & 0.05 \\ 0.05 & 0.11 & 0.25 & 0.11 & 0.05 \\ 0.05 & 0.11 & 0.11 & 0.11 & 0.05 \\ 0.01 & 0.05 & 0.05 & 0.05 & 0.01 \end{bmatrix} \tag{4.7}$$

The resulting matrix will contain $(k-1)$ less elements for each column and row compared to the original one, since the kernel is not applied to the border elements. Therefore considering a matrix M of $x \cdot y$ elements in input with a Kernel of $k \cdot k$ items, the size of the matrix containing the result of the convolution will be $n \cdot m$ where $n = (x - k + 1)$ and $m = (y - k + 1)$. In the considered benchmark, a 5x5 items Kernel is used.

The number of floating point operations and memory accesses, expressed as bytes, expected by the Gaussian blur operation, is:

- $FLOPS_{gaussblur} : 31 \cdot (m \cdot n)$
- $Bytes_{gaussblur} : (x \cdot y + k \cdot k + n \cdot m) \cdot sizeof(type)$

The Arithmetic Intensity of the operation can therefore be defined as:

$$AI_{gaussblur} = \frac{31 \cdot (m \cdot n)}{(x \cdot y + k \cdot k + n \cdot m) \cdot sizeof(type)} \ . \tag{4.8}$$

We evaluated the core function of the gaussblur benchmark using VTune with SDE, LIKWID, and Advisor to measure the performed values of FLOPs, memory accesses, and AI. The benchmark was executed with three different input dimensions: "1024 1024", "2048 2048" and "4096 4096". Each measurement was repeated 10 times to also report the values of variance and standard deviation.

Expected results and tools results are reported in Tables 4.9, 4.10 and 4.11, respectively for each input size.

Table 4.9: Comparison between the results obtained with the tools for the *gaussblur* benchmark using "1024 1024" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.032	0.008	4.000		
Intel VTune+SDE	0.032	0.101	0.316	0.218	0.048
RRZE LIKWID	0.032	0.117	0.273	2.276	5.179
Intel Advisor	0.032	0.008	4.000	0.000	0.000

For this benchmarks all the tools correctly evaluated the number of FLOPs, while only Advisor estimated the data transfer correctly. Compared to VTune, LIKWID have an

Table 4.10: Comparison between the results obtained with the tools for the *gaussblur* benchmark using "2048 2048" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.130	0.034	3.824		
Intel VTune+SDE	0.130	0.327	0.398	0.180	0.032
RRZE LIKWID	0.130	0.439	0.296	0.888	0.789
Intel Advisor	0.130	0.034	3.824	0.000	0.000

Table 4.11: Comparison between the results obtained with the tools for the *gameoflife* benchmark using "4096 4096" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.519	0.134	3.873		
Intel VTune+SDE	0.519	0.554	0.937	0.249	0.062
RRZE LIKWID	0.519	1.120	0.464	0.777	0.604
Intel Advisor	0.519	0.134	3.873	0.000	0.000

higher standard deviation, obtained from ten execution of the benchmark, appearing to be less reliable. The value of AI estimated by Advisor corresponds to the expected one computed manually.

4.2.4 *gradient* Benchmark: Gradient Operator

The gradient of a scalar function $f(x_1, x_2, x_3, ...x_n)$, represented with the symbol ∇f called *Nabla*, is a mathematical operation that allows to understand the rate of fastest increase of the function and its direction. In a three-dimensional Cartesian coordinate system the operation is given by:

$$\nabla f = \frac{\partial f}{\partial x}\boldsymbol{i} + \frac{\partial f}{\partial y}\boldsymbol{j} + \frac{\partial f}{\partial z}\boldsymbol{k} \ . \tag{4.9}$$

that is, the partial derivatives of the function in reference to the standard unit vectors in the direction of the Cartesian axes.

In the considered algorithm an approximation of the derivatives will be performed using the theorem of the finite differences, introduced in the Paragraph 4.2.1 regarding the divergence benchmark. The theorem approximates the value of the derivative at a certain

point by a weighted sum of the values at the neighboring items. While the divergence operator maps a vector to a scalar quantity by taking the sum of its three partial derivatives, the gradient operator expresses the variation of a physical quantity per unit of length in a given direction.

The number of floating-point operations and memory accesses, expressed as bytes, expected by this benchmark, where each component of the function is represented by a vector called respectively x, y and z, is:

- $FLOPS_{gradient} : 6 \cdot (x-2) \cdot (y-2) \cdot (z-2)$
- $Bytes_{gradient} : 4 \cdot (x \cdot y \cdot z) \cdot sizeof(type)$

The Arithmetic Intensity of the gradient benchmark can therefore be defined as:

$$AI_{gradient} = \frac{6 \cdot (x-2) \cdot (y-2) \cdot (z-2)}{4 \cdot (x \cdot y \cdot z) \cdot sizeof(type)} \ . \tag{4.10}$$

We evaluated the core function of the gradient benchmark using VTune with SDE, LIKWID, and Advisor to measure the performed values of FLOPs, memory accesses, and AI. The benchmark was executed with three different input dimensions: "128 128 128", "256 256 256" and "512 512 512". Each measurement was repeated 10 times to also report the values of variance and standard deviation.

Expected results and tools results are reported in Tables 4.12, 4.13 and 4.14, respectively for each input size.

Table 4.12: Comparison between the results obtained with the tools for the *gradient* benchmark using "128 128 128" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.012	0.034	0.353		
Intel VTune+SDE	0.012	n/a	n/a	n/a	n/a
RRZE LIKWID	0.012	0.124	0.097	0.042	0.002
Intel Advisor	0.012	0.033	0.364	0.000	0.000

After inspecting the results, we noticed that all the tools correctly evaluated the number of FLOPs. In terms of memory access evaluation, Advisor shows results correctly oriented towards data offload. While VTune was unable to collect data for the smallest input size, it shows values more in line with our analysis than LIKWID.

4.2.5 *laplacian* and *lapgsrb* Benchmarks: Laplace Operator

The Laplace operator is a second-order differential mathematical operation defined as the divergence of a function gradient in an Euclidean space. In Cartesian coordinates it is

Table 4.13: Comparison between the results obtained with the tools for the *gradient* benchmark using "256 256 256" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.098	0.268	0.366		
Intel VTune+SDE	0.098	0.377	0.260	0.105	0.011
RRZE LIKWID	0.098	0.899	0.109	0.039	0.002
Intel Advisor	0.098	0.268	0.366	0.000	0.000

Table 4.14: Comparison between the results obtained with the tools for the *gradient* benchmark using "512 512 512" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.796	2.147	0.371		
Intel VTune+SDE	0.796	4.682	0.170	0.022	0.001
RRZE LIKWID	0.796	7.391	0.108	0.040	0.002
Intel Advisor	0.795	2.147	0.370	0.000	0.000

defined as the sum of the second partial derivatives and can operate from two up to n dimensions. Assumes particular importance in the fields of electromagnetism and fluid dynamics.

The Laplace operator is defined as Nabla squared. Considering a function in a space of three dimensions, the related equation is:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2}\boldsymbol{i} + \frac{\partial^2 f}{\partial y^2}\boldsymbol{j} + \frac{\partial^2 f}{\partial z^2}\boldsymbol{k} \ . \tag{4.11}$$

On the same theory of the divergence benchmark, it is possible to approximate the computation of the Laplacian operator by the finite differences method, thus obtaining the "discrete Laplacian". This operator could be used for edge detection in image processing, since the image is considered as a pixel matrix and the discrete Laplacian operator is applied on each element. The operator is defined as the sum of differences over the nearest neighbors of the matrix item being considered.

The two benchmark being considered differ in the number of neighboring elements being evaluated. For the *laplacian* benchmark 6 closest points are considered (therefore 7 points including the central one) and for the *lapgsrb* one the Laplacian operator has been applied for 24 neighboring points (25 points considering the one in question).

The number of expected floating point operations and memory accesses, expressed as bytes, of the *laplacian* benchmark, where the matrix contains the three dimensions defined as x, y and z, is:

- $FLOPS_{laplacian} : 8 \cdot (x-2) \cdot (y-2) \cdot (z-2)$
- $Bytes_{laplacian} : (x \cdot y \cdot z + x \cdot y \cdot z) \cdot sizeof(type)$

The Arithmetic Intensity of the 7-point approximation of the Laplace operator can therefore be defined as:

$$AI_{laplacian} = \frac{8 \cdot (x-2) \cdot (y-2) \cdot (z-2)}{(x \cdot y \cdot z + x \cdot y \cdot z) \cdot sizeof(type)} \ . \tag{4.12}$$

We evaluated the core function of the laplacian benchmark using VTune with SDE, LIKWID, and Advisor to measure the performed values of FLOPs, memory accesses, and AI. The benchmark was executed with three different input dimensions: "128 128 128", "256 256 256" and "512 512 512". Each measurement was repeated 10 times to also report the values of variance and standard deviation.

Expected results and tools results are reported in Tables 4.15, 4.16 and 4.17, respectively for each input size.

Table 4.15: Comparison between the results obtained with the tools for the *laplacian* benchmark using "128 128 128" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.016	0.017	0.941		
Intel VTune+SDE	0.016	n/a	n/a	n/a	n/a
RRZE LIKWID	0.016	0.056	0.287	0.108	0.012
Intel Advisor	0.016	0.025	0.640	0.000	0.000

For the *lapgsrb* benchmark the number of expected floating point operations and memory accesses, expressed as bytes, where the matrix contains the three dimensions defined as x, y and z, are:

- $FLOPS_{lapgsrb} : 28 \cdot (x-4) \cdot (y-4) \cdot (z-4)$
- $Bytes_{lapgsrb} : (x \cdot y \cdot z + x \cdot y \cdot z) \cdot sizeof(type)$

The Arithmetic Intensity of the 25-point approximation of the Laplace operator can therefore be defined as:

$$AI_{lapgsrb} = \frac{28 \cdot (x-4) \cdot (y-4) \cdot (z-4)}{(x \cdot y \cdot z + x \cdot y \cdot z) \cdot sizeof(type)} \ . \tag{4.13}$$

Table 4.16: Comparison between the results obtained with the tools for the *laplacian* benchmark using "256 256 256" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.131	0.134	0.978		
Intel VTune+SDE	0.131	0.279	0.469	0.253	0.064
RRZE LIKWID	0.131	0.392	0.335	0.110	0.012
Intel Advisor	0.131	0.201	0.652	0.000	0.000

Table 4.17: Comparison between the results obtained with the tools for the *laplacian* benchmark using "512 512 512" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	1.061	1.074	0.988		
Intel VTune+SDE	1.061	2.679	0.396	0.102	0.010
RRZE LIKWID	1.061	3.215	0.330	0.122	0.015
Intel Advisor	1.061	1.607	0.660	0.000	0.000

Also for the lapgsrb benchmark, we evaluated its core function using VTune with SDE, LIKWID, and Advisor to measure the performed values of FLOPs, memory accesses, and AI. The benchmark was executed with three different input dimensions: "128 128 128", "256 256 256" and "512 512 512". Each measurement was repeated 10 times to also report the values of variance and standard deviation.

Expected results and tools results are reported in Tables 4.18, 4.19 and 4.20, respectively for each input size.

Table 4.18: Comparison between the results obtained with the tools for the *lapgsrb* benchmark using "128 128 128" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.053	0.017	3.118		
Intel VTune+SDE	0.053	0.077	0.691	0.422	0.178
RRZE LIKWID	0.053	0.200	0.266	0.510	0.260
Intel Advisor	0.053	0.017	3.118	0.000	0.000

Table 4.19: Comparison between the results obtained with the tools for the *lapgsrb* benchmark using "256 256 256" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.448	0.134	3.343		
Intel VTune+SDE	0.448	0.798	0.561	0.291	0.085
RRZE LIKWID	0.448	1.640	0.273	0.613	0.376
Intel Advisor	0.448	0.134	3.343	0.000	0.000

Table 4.20: Comparison between the results obtained with the tools for the *lapgsrb* benchmark using "512 512 512" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	3.671	1.074	3.418		
Intel VTune+SDE	3.671	5.631	0.652	0.400	0.160
RRZE LIKWID	3.671	11.779	0.312	0.537	0.288
Intel Advisor	3.671	1.074	3.418	0.000	0.000

For both the benchmarks and for all the evaluations, the number of FLOPs measured by the tools is correct. For the laplacian benchmark the number of memory accesses is slightly overestimated by Advisor, while for VTune and LIKWID they are out-of-scale, showing their inadequacy for the purpose of evaluating the data exchange between host and device for a GPU execution. This happens also for the lapgsrb benchmark, while Advisor correctly estimated the value of AI. The standard deviation of the LIKWID measurement is higher than the VTune one, which proves to me more reliable.

4.2.6 *matvec* Benchmark: Matrix-vector Multiplication

The matrix-vector multiplication is a mathematical operation that takes as input a matrix M of $x \cdot y$ elements and a vector V of y, to produce as output a vector of y elements. Is it possible to define this operation only if the number of columns of M is the same of the rows of V. The matrix-vector multiplication (mvm) can be defined by the following equation:

$$mvm = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \end{bmatrix} \tag{4.14}$$

The operation defines the n-th element of the result matrix as the dot product of the n-th row of the matrix M with the vector V.

The number of floating point operations and memory accesses, expressed as bytes, expected by the matrix-vector multiplication, are:

- $FLOPS_{matvec} : x \cdot (2y - 1)$
- $Bytes_{matvec} : (x + y + x \cdot y) \cdot sizeof(type)$

The Arithmetic Intensity of the benchmark can therefore be defined as:

$$AI_{matvec} = \frac{x \cdot (2y - 1)}{(x + y + x \cdot y) \cdot sizeof(type)} \ . \tag{4.15}$$

We evaluated the core function of the matvec benchmark using VTune with SDE, LIKWID, and Advisor to measure the performed values of FLOPs, memory accesses, and AI. The benchmark was executed with three different input dimensions: "1024 1024", "2048 2048" and "4096 4096". Each measurement was repeated 10 times to also report the values of variance and standard deviation.

Expected results and tools results are reported in Tables 4.21, 4.22 and 4.23, respectively for each input size.

Table 4.21: Comparison between the results obtained with the tools for the *matvec* benchmark using "1024 1024" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.002	0.004	0.500		
Intel VTune+SDE	0.002	n/a	n/a	n/a	n/a
RRZE LIKWID	0.002	0.016	0.135	0.065	0.004
Intel Advisor	0.002	0.004	0.500	0.000	0.000

For the matvec benchmark, all the tools correctly evaluated the number of FLOPs. Unfortunately, VTune for the three dimensions and for all the repetitions was not able to estimate the number of memory accesses, since the function execution time was faster than its sampling interval. LIKWID measurements, even if are not as expected, are not completely out of range probably because the vector may fit better in the cache than a matrix, resulting in fewer reads of the same range of values. Advisor instead, was able to estimate correctly the AI for all the measurements.

Table 4.22: Comparison between the results obtained with the tools for the *matvec* benchmark using "2048 2048" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.008	0.017	0.471		
Intel VTune+SDE	0.008	n/a	n/a	n/a	n/a
RRZE LIKWID	0.009	0.037	0.237	0.159	0.025
Intel Advisor	0.008	0.017	0.471	0.000	0.000

Table 4.23: Comparison between the results obtained with the tools for the *matvec* benchmark using "4096 4096" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.034	0.067	0.507		
Intel VTune+SDE	0.034	n/a	n/a	n/a	n/a
RRZE LIKWID	0.034	0.127	0.266	0.111	0.012
Intel Advisor	0.034	0.067	0.507	0.000	0.000

4.2.7 *tricubic* Benchmark: Tricubic Interpolation

In numerical analysis by "interpolation" is intended an evaluation of the values of a function in different points. It is referred to as "tricubic" when the arbitrary points are in 3D space and is mainly used in ocean dynamics and meteorology.

To briefly introduce the theory behind the operation, assuming that a function f is given at a corner of a regular mesh (that is a collection of quadrilaterals that represents a surface), inside each cube of three dimensions the function is approximated by an expression such as:

$$f(x,y,z) = \sum_{i=0}^{N}\sum_{j=0}^{N}\sum_{k=0}^{N} a_{ijk}x^i y^j z^k. \tag{4.16}$$

To guarantee the C^1 continuity, f and its three first derivatives should be continuous on each of the six faces of the cubes. To ensure that, four constraints at the eight corners of the element are needed and therefore are necessary 32 constraints. Reconsidering the equation 4.16, N must be equal to 3 to keep the size of the coefficients required. The resulting tricubic interpolation of the function can be expressed as:

$$f(x,y,z) = \sum_{i=0}^{3}\sum_{j=0}^{3}\sum_{k=0}^{3} a_{ijk}x^i y^j z^k. \tag{4.17}$$

and is demonstrated the first 32 constrains of the 64 obtained from the formula are enough to ensure the continuity. An in-depth study by Lekien et al. can be found at [27].

The number of floating point operations and memory accesses, expressed as bytes, expected by the tricubic algorithm is:

- $FLOPS_{tricubic} : 242 \cdot (x \cdot y \cdot (z-2))$
- $Bytes_{tricubic} : 5 \cdot (x \cdot y \cdot z) \cdot sizeof(type)$

The Arithmetic Intensity of the benchmark can therefore be defined as:

$$AI_{tricubic} = \frac{242 \cdot (x \cdot y \cdot (z-2))}{5 \cdot (x \cdot y \cdot z) \cdot sizeof(type)} \ . \tag{4.18}$$

We evaluated the core function of the tricubic benchmark using VTune with SDE, LIKWID, and Advisor to measure the performed values of FLOPs, memory accesses, and AI. The benchmark was executed with three different input dimensions: "128 128 128", "256 256 256" and "512 512 512". Each measurement was repeated 10 times to also report the values of variance and standard deviation.

Expected results and tools results are reported in Tables 4.24, 4.25 and 4.26, respectively for each input size.

Table 4.24: Comparison between the results obtained with the tools for the *tricubic* benchmark using "128 128 128" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.499	0.042	11.881		
Intel VTune+SDE	0.369	0.488	0.756	0.469	0.220
RRZE LIKWID	0.360	0.301	1.195	1.606	2.580
Intel Advisor	0.369	0.042	8.786	0.000	0.000

After inspecting the results, we notice that SDE, LIKWID, and Advisor measured a number of FLOPs that were really similar to each other but slightly different from the expected values. Since the tricubic function is quite computationally intensive, it is likely that the compilers performed optimizations on the operations, reducing the number of operations required. This is also supported by the fact that the tools using an Intel compiler measured the same value. Advisor estimates the AI more faithfully, compared to the other tools, to what we expected, thanks to its ability to model the data transfer between host and GPU. Also, both VTune and LIKWID reported an high value of standard deviation and variance for the different measurement.

Table 4.25: Comparison between the results obtained with the tools for the *tricubic* benchmark using "256 256 256" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	4.028	0.336	11.988		
Intel VTune+SDE	3.061	1.957	1.564	0.972	0.946
RRZE LIKWID	2.980	2.046	1.456	1.774	3.147
Intel Advisor	3.061	0.335	9.137	0.000	0.000

Table 4.26: Comparison between the results obtained with the tools for the *laplacian* benchmark using "512 512 512" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	32.353	2.684	12.054		
Intel VTune+SDE	24.924	14.986	1.663	1.354	1.835
RRZE LIKWID	24.264	17.377	1.396	1.598	2.554
Intel Advisor	24.924	2.687	9.276	0.000	0.000

4.2.8 *uxx1* Benchmark: Approximation of Second Derivative

The algorithm implemented in the benchmark in question approximates the second derivative of a function $f_{(x,y,x)}$, which has been discretized and its values are contained in a three-dimensional matrix.

The computation is based on the finite difference method, introduced for the divergence benchmark in Paragraph 4.2.1, that allows to calculate an approximation of the derivative of a function even for orders higher then the first one. Since there is no data dependency between the various operations, the algorithm can be efficiently implemented on GPU.

To obtain the second derivative of a function, the Equation 4.19 is applied to the function, where h represents the interval between two adjacent points in the approximation.

$$f''(x) = \frac{\frac{f(x+h)-f(x)}{h} - \frac{f(x)-f(x-h)}{h}}{h} = \frac{f(x+h)-2f(x)+f(x-h)}{h^2} \,. \quad (4.19)$$

For the sake of brevity, the equation considers only a single-variable function. However, the benchmark applies the same concept to calculate the approximate derivative in three dimensions instead of one.

The number of floating point operations and memory accesses, expressed as bytes, expected by this implementation of the second derivative approximation is:

- $FLOPS_{uxx1} : 25 \cdot (x-3) \cdot (y-3) \cdot (z-3)$
- $Bytes_{uxx1} : 6 \cdot (x \cdot y \cdot z) \cdot sizeof(type)$

The Arithmetic Intensity of the uxx1 benchmark can therefore be defined as:

$$AI_{uxx1} = \frac{25 \cdot (x-3) \cdot (y-3) \cdot (z-3)}{6 \cdot (x \cdot y \cdot z) \cdot sizeof(type)} \ . \tag{4.20}$$

We evaluated the core function of the uxx1 benchmark using VTune with SDE, LIKWID, and Advisor to measure the performed values of FLOPs, memory accesses, and AI. The benchmark was executed with three different input dimensions: "128 128 128", "256 256 256" and "512 512 512". Each measurement was repeated 10 times to also report the values of variance and standard deviation.

Expected results and tools results are reported in Tables 4.27, 4.28 and 4.29, respectively for each input size.

Table 4.27: Comparison between the results obtained with the tools for the *uxx1* benchmark using "128 128 128" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.049	0.05	0.980		
Intel VTune+SDE	0.054	n/a	n/a	n/a	n/a
RRZE LIKWID	0.037	0.194	0.191	0.151	0.023
Intel Advisor	0.054	0.050	1.080	0.000	0.000

Table 4.28: Comparison between the results obtained with the tools for the *uxx1* benchmark using "256 256 256" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.405	0.403	1.005		
Intel VTune+SDE	0.451	0.700	0.644	0.189	0.036
RRZE LIKWID	0.308	1.720	0.179	0.181	0.033
Intel Advisor	0.451	0.403	1.119	0.000	0.000

Table 4.29: Comparison between the results obtained with the tools for the *uxx1* benchmark using "512 512 512" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	3.297	3.221	1.024		
Intel VTune+SDE	3.682	6.702	0.549	0.125	0.016
RRZE LIKWID	2.506	13.654	0.184	0.164	0.027
Intel Advisor	3.682	3.217	1.145	0.000	0.000

We can observe from the result tables that the number of FLOPs varies slightly between the expected values and those measured by the tools. This variation is caused by the compiler, which intervenes in the optimization. Advisor evaluation of AI is correct. VTune for the first input size can't measure the number of memory accesses performed by the algorithm, since its sampling interval is slower than the function execution but for the other sizes the estimation is good, even if not oriented to data offloading. The AI evaluated with LIKWID is not correct, especially for the largest size.

4.2.9 *vecadd* Benchmark: Sum between Matrices

Although the name suggests that the benchmark performs an addition between vectors, the algorithm also allows the addition of two matrices up to three dimensions.

Starting from the matrices A and B, with dimensions x, y and z, the operation returns another matrix, called sum matrix, with the same size as the two in input. The sum is in fact possible only if the starting matrices A and B have the same dimensions. The result is obtained by adding the elements that occupy the same position. Assuming that $A + B = C$:

$$c_{xyz} = a_{xyz} + b_{xyz}. \tag{4.21}$$

By arranging the dimensions of the matrices, it is possible to obtain obtain the sum of two vectors, as well as the sum of objects in two dimensions.

The number of floating point operations and memory accesses, expressed as bytes, expected by the operation is:

- $FLOPS_{vecadd} : (x \cdot y \cdot z)$
- $Bytes_{vecadd} : (x \cdot y \cdot z + x \cdot y \cdot z + x \cdot y \cdot z) \cdot sizeof(type)$

The Arithmetic Intensity of the sum between two matrices of three components can therefore be defined as:

$$AI_{vecadd} = \frac{(x \cdot y \cdot z)}{(x \cdot y \cdot z + x \cdot y \cdot z + x \cdot y \cdot z) \cdot sizeof(type)} \ . \tag{4.22}$$

We evaluated the core function of the vecadd benchmark using VTune with SDE, LIKWID, and Advisor to measure the performed values of FLOPs, memory accesses, and AI. The benchmark was executed with three different input dimensions: "128 128 128", "256 256 256" and "512 512 512". Each measurement was repeated 10 times to also report the values of variance and standard deviation.

Expected results and tools results are reported in Tables 4.30, 4.31 and 4.32, respectively for each input size.

Table 4.30: Comparison between the results obtained with the tools for the *vecadd* benchmark using "128 128 128" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.002	0.025	0.080		
Intel VTune+SDE	0.002	n/a	n/a	n/a	n/a
RRZE LIKWID	0.002	0.052	0.039	0.016	0.000
Intel Advisor	0.002	0.025	0.080	0.000	0.000

Table 4.31: Comparison between the results obtained with the tools for the *vecadd* benchmark using "256 256 256" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.017	0.201	0.085		
Intel VTune+SDE	0.017	0.212	0.080	0.040	0.002
RRZE LIKWID	0.017	0.383	0.044	0.011	0.000
Intel Advisor	0.017	0.201	0.085	0.000	0.000

For the vecadd benchmark, all the tools correctly estimate the number of FLOPs. However, VTune cannot estimate the number of memory accesses for the first input dimension due to its sampling interval being slower than the function execution. VTune and LIKIWID also show limitations in evaluating data transfer oriented to offloading. In contrast, Advisor accurately estimates the Arithmetic Intensity of the operation for all input sizes.

Table 4.32: Comparison between the results obtained with the tools for the *vecadd* benchmark using "512 512 512" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.134	1.611	0.083		
Intel VTune+SDE	0.134	2.347	0.057	0.005	0.000
RRZE LIKWID	0.134	2.937	0.046	0.011	0.000
Intel Advisor	0.134	1.607	0.083	0.000	0.000

4.2.10 *wave13pt* Benchmark: 3D Wave Equation Solver

The *wave*13*pt* benchmark of the KernelGen Test Suite is derived from a benchmark originally developed for the PATUS Framework [5]. The benchmark implements an approximation of a 3-dimensional wave equation, which is commonly used to model sound or electromagnetic waves.

Starting from the classical wave equation and imposing some initial conditions as it follows:

$$\begin{aligned} \frac{\partial^2 u}{\partial t^2} - c^2 \Delta u &= 0 \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \\ u(x,y,z)|_{t=0} &= \sin(2\pi x)\sin(2\pi y)\sin(2\pi z). \end{aligned} \tag{4.23}$$

By using an explicit finite difference method to discretize the equation in both space and time, we can employ a fourth-order discretization of the Laplacian Delta over an equidistant spatial grid with step size h and a second-order scheme with a time step *delta t*. The resulting equation is:

$$\frac{u^{(t+\delta t)} - 2u^{(t)} + u^{(t-\delta t)}}{\delta t} - c^2 \Delta_h u^{(t)} = 0. \tag{4.24}$$

As explained in [6], solving the equation by $u^{t+\delta t}$ and interpreting u as a grid in space and time with mesh size h and time step δt, we obtain the Equation 4.25 that is an approximation of a 3-dimensional wave equation.

$$
\begin{aligned}
u[x,y,z;t+1] = 2u[x,y,z;t] - u[x,y,z;t-1] + c^2\frac{\delta t}{h^2}(-\frac{15}{2}u[x,y,z;t]+ \\
-\frac{1}{12}(u[x-2,y,z;t]+u[x,y-2,z;t]+u[x,y,z-2;t])+ \\
\frac{4}{3}(u[x-1,y,z;t]+u[x,y-1,z;t]+u[x,y,z-1;t])+ \\
\frac{4}{3}(u[x+1,y,z;t]+u[x,y+1,z;t]+u[x,y,z+1;t])+ \\
-\frac{1}{12}(u[x+2,y,z;t]+u[x,y+2,z;t]+u[x,y,z-+2;t])).
\end{aligned}
\tag{4.25}
$$

Given the three matrices defined in the C-code representing the spacial grid, the expected number of floating point operations and memory accesses expressed in bytes, for the wave13pt benchmark are:

- $FLOPS_{wave13pt} : 16 \cdot (x \cdot y \cdot (z-4))$
- $Bytes_{wave13pt} : 3 \cdot (x \cdot y \cdot z) \cdot sizeof(type)$

The Arithmetic Intensity (AI) can therefore be obtained as:

$$
AI_{wave13pt} = \frac{16 \cdot (x \cdot y \cdot (z-4))}{3 \cdot (x \cdot y \cdot z) \cdot sizeof(type)} \ . \tag{4.26}
$$

We evaluated the core function of the wave13pt benchmark using VTune with SDE, LIKWID, and Advisor to measure the performed values of FLOPs, memory accesses, and AI. The benchmark was executed with three different input dimensions: "128 128 128", "256 256 256" and "512 512 512". Each measurement was repeated 10 times to also report the values of variance and standard deviation.

Expected results and tools results are reported in Tables 4.33, 4.34 and 4.35, respectively for each input size.

Table 4.33: Comparison between the results obtained with the tools for the *wave13pt* benchmark using "128 128 128" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.033	0.025	1.320		
Intel VTune+SDE	0.031	n/a	n/a	n/a	n/a
RRZE LIKWID	0.031	0.077	0.400	0.199	0.040
Intel Advisor	0.031	0.025	1.240	0.000	0.000

All the tools correctly estimated the FLOPs computed by the operation, with small tolerable differences due to compiler optimization. The memory accesses estimated by

Table 4.34: Comparison between the results obtained with the tools for the *wave13pt* benchmark using "256 256 256" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.264	0.201	1.313		
Intel VTune+SDE	0.256	0.570	0.449	0.130	0.017
RRZE LIKWID	0.257	0.594	0.433	0.252	0.064
Intel Advisor	0.256	0.201	1.274	0.000	0.000

Table 4.35: Comparison between the results obtained with the tools for the *wave13pt* benchmark using "512 512 512" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	2.131	1.611	1.323		
Intel VTune+SDE	2.098	5.410	0.388	0.156	0.024
RRZE LIKWID	2.102	6.318	0.333	0.245	0.060
Intel Advisor	2.098	1.607	1.306	0.000	0.000

Advisor are correct and so the estimated AI. VTune and LIKWID evaluation of memory is proved to be not oriented to data offloading, with VTune sampling interval that does not allow the tool to evaluate the first input dimension. LIKWID shows a higher standard deviation between the different repetitions of the same measurement.

4.2.11 *whispering* Benchmark: 2D Nanophotonics Simulation

The algorithm underlying this benchmark implements a 2D nanophotonics simulation. Nanophotonics is the study of how light behaves on the nanometer scale and its interaction with objects at this scale. This type of simulation is commonly used in electrical engineering fields such as for solar cells, as well in biomedical fields for controlled release of anti-cancer therapeutics.

The algorithm uses the user-entered input (x, y) to randomly initialize 2D matrices. These matrices are then used as input for calculating Maxwell's equation using the FDTD (finite difference time-domain) method: it is a version of the finite differences introduced for the divergence benchmark in Section 4.2.1, that discretizes the equations using central-difference approximations to the space and time partial derivatives. The results obtained are then integrated with the energy density to complete the computation.

However, the mathematical theory behind the algorithm will not be explained in detail and can be found in [28].

By studying the algorithm, we can determine the number of floating point operations and bytes of memory accesses involved in this operation, which are:

- $FLOPS_{whispering} : 31 \cdot (x-2) \cdot (y-2)$
- $Bytes_{whispering} : (10 \cdot (x \cdot y)) \cdot sizeof(type)$

The resulting Arithmetic Intensity of the whispering benchmark can be obtained as:

$$AI_{whispering} = \frac{31 \cdot (x-2) \cdot (y-2)}{(10 \cdot (x \cdot y)) \cdot sizeof(type)} \ . \tag{4.27}$$

We evaluated the core function of the benchmark using VTune with SDE, LIKWID, and Advisor to measure the performed values of FLOPs, memory accesses, and AI. The benchmark was executed with three different input dimensions: "1024 1024", "2048 2048" and "4096 4096". Each measurement was repeated 10 times to also report the values of variance and standard deviation.

Expected results and tools results are reported in Tables 4.36, 4.37 and 4.38, respectively for each input size.

Table 4.36: Comparison between the results obtained with the tools for the *whispering* benchmark using "1024 1024" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.032	0.042	0.762		
Intel VTune+SDE	0.030	0.086	0.347	0.205	0.042
RRZE LIKWID	0.030	0.626	0.048	0.121	0.015
Intel Advisor	0.039	0.050	0.780	0.000	0.000

As shown in the result tables, all the tools correctly estimated the number of FLOPs with negligible differences. However, Advisor slightly overestimated the number of memory accesses, although it still evaluated the correct value of AI. On the other hand, VTune and LIKWID once again demonstrate their inadequacy in modeling data transfer from host to device since their measurements are based on the transfer from main memory to the core. Moreover, LIKWID showed a higher standard deviation between measurements with the same input than VTune.

Table 4.37: Comparison between the results obtained with the tools for the *whispering* benchmark using "2048 2048" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.130	0.168	0.774		
Intel VTune+SDE	0.121	0.478	0.253	0.076	0.006
RRZE LIKWID	0.121	2.108	0.057	0.153	0.023
Intel Advisor	0.155	0.201	0.771	0.000	0.000

Table 4.38: Comparison between the results obtained with the tools for the *whispering* benchmark using "2048 2048" as input.

Tool	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/ Bytes]	Arithmetic Intensity Standard deviation	Arithmetic Intensity Variance
Expected	0.520	0.671	0.775		
Intel VTune+SDE	0.486	1.974	0.246	0.066	0.004
RRZE LIKWID	0.486	8.420	0.058	0.162	0.026
Intel Advisor	0.620	0.805	0.770	0.000	0.000

4.3 Validation through GPU Execution

We have completed data collection using the tools and now aim to verify our results by running the KernelGen Test Suite on the machine GPU. Through the use of the implementation of the benchmarks in CUDA, a GPU programming language developed by NVIDIA, we will verify if the number of floating point operations performed and the amount of data exchanged between the host and the device match the values we calculated manually and those measured with the three tools.

Before proceeding, let us examine how we can evaluate the Arithmetic Intensity of a function executed on a GPU. The methods proposed are specifically designed for NVIDIA GPUs.

4.3.1 How to Measure Arithmetic Intensity of a GPU Kernel

To obtain the Arithmetic Intensity (AI) of a function executed on a GPU, called kernel, first we need to profile the number of floating-point operations (FLOPs) performed and then the number of memory accesses. Finally, we can divide them to get the value of AI.

We propose two methods for both the evaluations: one that supports older architectures and another that works only with the newest ones. All the tools described are

developed by NVIDIA.

The first method to measure FLOPs of a kernel, supported up to the NVIDIA Volta architecture, is to use NVIDIA Visual Profiler (nvprof) CLI [38], that allows to collect and view profiling data of CUDA activities. The analysis can be performed by executing:

Listing 4.8 NVIDIA Visual Profiler analysis on a GPU kernel to evaluate its FLOPs.

```
nvprof --kernels kernelName --metrics flop_count_sp ./app.o
```

where:

- *--kernels <list>*: limits the collection to the specified kernels list;
- *--metrics <list>*: allows to list the metrics to evaluate during the analysis. "flop_count_sp" counts the number of single-precision floating-point operations.

A table containing the number of FLOPs performed by the kernel will be printed on screen.

The second proposed method uses NVIDIA Nsight Compute (ncu) CLI [36], a performance analysis tool for GPU applications. It is supported starting from the Volta architecture and can be started using the following command:

Listing 4.9 NVIDIA Nsight Compute analysis on a GPU kernel to evaluate its FLOPs.

```
ncu -k kernelName --metrics
    smsp__sass_thread_inst_executed_op_fadd_pred_on.sum,
    smsp__sass_thread_inst_executed_op_fmul_pred_on.sum,
    smsp__sass_thread_inst_executed_op_ffma_pred_on.sum ./app.o
```

where:

- *-k <string>*: collects the data only for the specified kernel;
- *--metrics <list*: requires the list of metrics to evaluate during the analysis.

In this case, the results are also printed on screen. However, find the number of FLOPs computed requires an extra step, since it is broken down into more basic metrics, as listed in Listing 4.9. One metric reports the number of additions, another reports the number of multiplications, and a third reports the number of fused multiply-add operations, which performs both addition and multiplication in one operation, so it must be counted twice. The number of FLOPs can therefore be derived using the following formula:

$$FLOPs_{ncu\ analysis} = smsp\_\_sass\_thread\_inst\_executed\_op\_fadd\_pred\_on.sum + smsp\_\_sass\_thread\_inst\_executed\_op\_fmul\_pred\_on.sum + smsp\_\_sass\_thread\_inst\_executed\_op\_ffma\_pred\_on.sum \cdot 2. \tag{4.28}$$

We now propose two methods, based on the architecture available, to evaluate the memory accesses required by an application that also runs on GPU. Unfortunately, unlike

measuring FLOPs, it is not possible to distinguish the memory usage for a single kernel as the measurement occurs for the entire execution. If there are multiple kernels that have to run on the GPU, only one kernel can be enabled at a time to trace its memory accesses with the following methods. For GPUs up to Volta, we can still use the "nvprof" tool, and the analysis can be performed by executing:

Listing 4.10 NVIDIA Visual Profiler analysis to collect memory metrics via terminal.

```
nvprof --track-memory-allocation on -o output_memory.sql ./app
```

where:

- *--track-memory-allocation on*: turns on the tracking of the memory operations performed;
- *-o <string>*: exports the report of the analysis the specified file. Note that the file format must be "sql".

We can use the NVIDIA Nsight System (nsys) CLI [37] for GPUs with architecture starting from Pascal. Similar to the previous ones, it is an analysis tool used to profile and analyze the performance of GPU applications. The analysis to track memory accesses can be initiated with the following command:

Listing 4.11 NVIDIA Nsight System analysis to collect memory metrics via terminal.

```
nsys export output_memory.nsys-rep --type=sqlite
```

where:

- *export <string>*: launches the application and generates the export file;
- *--type=sqlite*: specifies that the report containing the results must be in sqlite format.

After the analysis is completed, we can apply the same procedure to evaluate the memory metrics for both the proposed methods. Since CPU and GPU have separate memory spaces, first the memory needs to be allocated on the device, next the data is copied from and to the host, and then it is released. The operation of allocation on the device in C/C++, takes place through the *cudaMalloc(void** pointer, size_t nbytes)* function, that allocates spaces for the device copies of the data. By opening the SQL database and summing the values from the "byte" column of the table "CUPTI_ACTIVITY_KIND_MEMORY", we can obtain the quantity of memory allocated in bytes. Instead, the data moving operations are carried out through the *cudaMemCpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction)* function. Its value can be obtained by opening the database and summing the values from the "byte" column of the table "CUPTI_ACTIVITY_KIND_MEMCPY". All the cudaMemCpy operations for a specific kernel represents its data exchange between the host and the device and therefore the value is considered to evaluate the AI of the function.

4.3.2 Comparing Tool Results using GPU-based Analysis

Each benchmark in the KernelGen Test Suite includes an implementation of the core function written in CUDA that we want to run on the GPU of the machine to evaluate the Arithmetic Intensity achieved. We aim to compare the results with the ones previously estimated manually and collected with the tools, to validate if they match in real-world usage scenarios.

To compile the GPU version of the benchmarks we used the NVIDIA CUDA Compiler (nvcc), a proprietary compiler developed by NVIDIA that enables GPU acceleration of kernels. It is part of "CUDA Toolkit" [33], a suite of tools for developing GPU-accelerated applications. The command used to compile each benchmark is the following:

Listing 4.12 Command to compile the CUDA implementation of the KernelGen Test Suite.

```
1 nvcc -L$(CUDA_libraries) -I$(CUDA_include) -O3 -arch=sm_50 benchmark.o
```

where:

- *CUDA_libraries* and *CUDA_include*: are paths in the system to libraries and include of the CUDA Toolkit;
- *-O3*: turns on an high grade of compile optimizations, the same used to compile the code for the tools analysis;
- *-arch=<string>*: specify the architecture on which the code will be executed. "sm_50" matches a Maxwell card in this case.

To collect the FLOPs and memory accesses of the GPU execution, we can apply the methods explained in Section 4.3.1. The results of the data collection for all the benchmarks, together with the expected values evaluated with pen and paper for comparison, are available in Table 4.39.

Upon studying the results, a discrepancy between the memory access values for GPU execution and the expected values can be observed, and this happens for every benchmark in the suite. By inspecting the source code of the benchmarks, we can see that each one works with matrix data structures, to contain the input and output of the operations performed. The GPU implementation mandates allocation of all the matrices on the device, followed by moving input matrices to the device and later copying back the resulting matrices to the host. However, it was observed that during the H→D copies, the matrix that will contain the results is initialized with random values also moved to the device, which is unnecessary. This results in an increased number of bytes moved, which does not align with the expected values. The benchmarks were not optimized for GPU execution as their primary goal was to perform compiler comparisons. To optimize the process, the input and output matrices must be allocated on the device, and only the input ones should be moved to the device. Once the GPU has filled the output matrix, it can then be moved to the host.

To overcome the issue of increased data movement performed by the GPU implementation of the KernelGen Test Suite, is possible to consider in our analysis the bytes of

memory allocation for each benchmark, instead of the memory accesses, that is the value crucial to evaluate the Arithmetic Intensity. Notably, the memory allocation value for the benchmarks always corresponds to the sum of data that needs to be copied to the device and moved back to the host. This value, as shown in Table 4.39, also coincides with the expected value, which was manually evaluated from the source code. These findings validate previous observations across various tools and underscore the importance of accurately evaluating the amount of data to be transferred during each phase, as it can significantly impact the execution time once the algorithms that can be offloaded to the GPU are identified.

Regarding the evaluation of floating-point operations (FLOPs), we notice that sometimes there are a small variation that may be due to the compiler optimization, resuting in a reduction in the number of required operations. This occurs in the gameoflife, tricubic and wave13pt benchmarks. However, in the gaussblur, uxx1 and whispering benchmarks, the number of FLOPs is slight higher than expected. NVIDIA reports that a floating point instruction written in CUDA can result in multiple assembly instructions depending on the compiler flags and the actual operation, causing a difference from the expected value [32]. In general, the results of the FLOPs evaluation are in line with our expectations, with a negligible variation.

Overall, the results of the GPU evaluation of the benchmarks match our expectations and validate our analysis. Figure 4.2 through Figure 4.13 provide a graphical comparison between the AI measured for the CUDA implementations and the tools measurements for all the benchmarks. The FLOPs evaluated by SDE, LIKWID and Advisor are correct, with only slight variations, which supports their reliability in the analysis. However, in terms of memory accesses, LIKWID and VTune measure a value that is not suitable for the analysis we want to perform. This is because they read from counters located between the main memory and the processor core, which also take into account the multiple readings that occur during the operation, thereby increasing the number of memory accesses counted. Advisor is confirmed as the tool that best estimates the AI, as the memory movement it estimates specifically focuses on data offloading. Moreover, the number of memory accesses it models could also help developer to fix performance issues by identifying opportunities to optimize data movement between the host and device, even after the GPU implementation has already been written, as was the case in our study.

Table 4.39: Comparison of the GFLOPs and memory metrics between the values calculated manually and those obtained by the GPU execution, for the KernelGen Test Suite.

Benchmark	Input Size	**Expected**		**GPU Measurements**		
		GFLOPs	Memory Accesses [GB]	GFLOPs	Memory Allocation [GB]	Memory Accesses [GB]
divergence	128	0.016	0.034	0.016	0.034	0.042
	256	0.131	0.268	0.131	0.268	0.336
	512	1.061	2.147	1.061	2.147	2.684
gameoflife	1024	0.015	0.008	0.008	0.008	0.013
	2048	0.059	0.034	0.033	0.034	0.050
	4096	0.235	0.134	0.134	0.134	0.201
gaussblur	1024	0.032	0.008	0.045	0.008	0.013
	2048	0.130	0.034	0.180	0.034	0.050
	4096	0.519	0.134	0.720	0.134	0.201
gradient	128	0.012	0.034	0.012	0.034	0.059
	256	0.098	0.268	0.098	0.268	0.470
	512	0.796	2.147	0.796	2.147	3.758
lapgsrb	128	0.053	0.017	0.053	0.017	0.025
	256	0.448	0.134	0.448	0.134	0.201
	512	3.671	1.074	3.671	1.074	1.611
laplacian	128	0.016	0.017	0.016	0.017	0.025
	256	0.131	0.134	0.131	0.134	0.201
	512	1.061	1.074	1.061	1.074	1.611
matvec	1024	0.002	0.004	0.002	0.004	0.004
	2048	0.008	0.017	0.008	0.017	0.017
	4096	0.034	0.067	0.034	0.067	0.067
tricubic	128	0.499	0.042	0.492	0.042	0.050
	256	4.028	0.336	4.081	0.336	0.403
	512	32.353	2.684	33.232	2.684	3.221
uxx1	128	0.049	0.050	0.074	0.050	0.059
	256	0.405	0.403	0.615	0.403	0.470
	512	3.297	3.221	5.011	3.221	3.758
vecadd	128	0.002	0.025	0.002	0.025	0.042
	256	0.017	0.201	0.017	0.201	0.336
	512	0.134	1.611	0.134	1.611	2.684
wave13pt	128	0.033	0.025	0.031	0.025	0.034
	256	0.264	0.201	0.256	0.201	0.268
	512	2.131	1.611	2.098	1.611	2.147
whispering	1024	0.032	0.042	0.045	0.042	0.042
	2048	0.130	0.168	0.180	0.168	0.168
	4096	0.520	0.671	0.721	0.671	0.671

Figure 4.2: *divergence* benchmark, comparison of CUDA implementation with tools results.

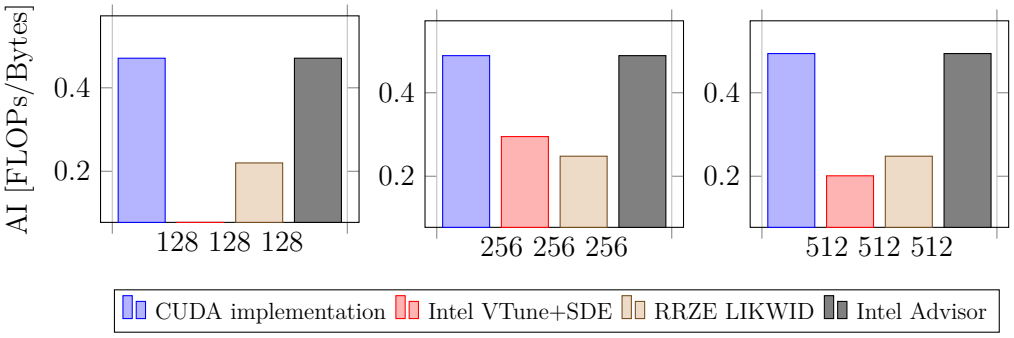


Figure 4.3: *gameoflife* benchmark, comparison of CUDA implementation with tools results.

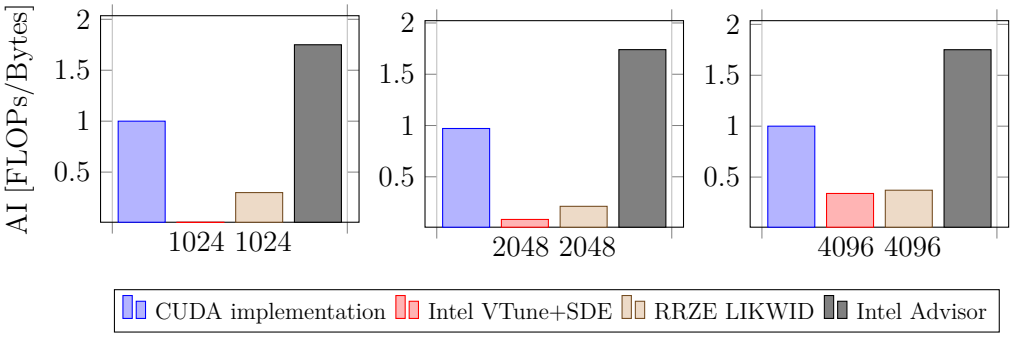


Figure 4.4: *gaussblur* benchmark, comparison of CUDA implementation with tools results.

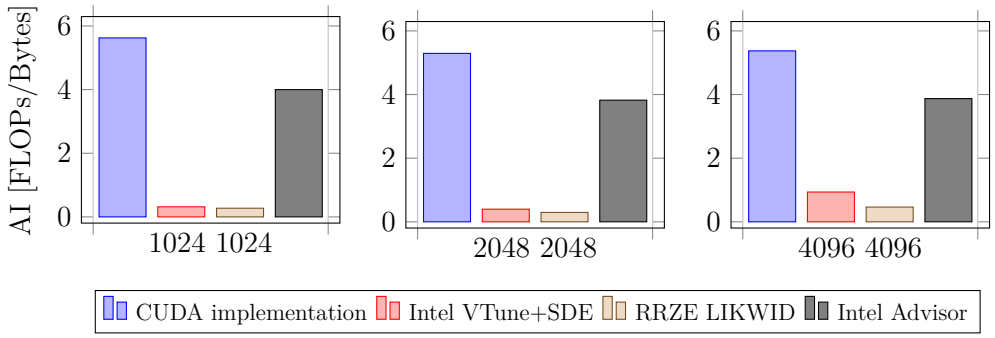


Figure 4.5: *gradient* benchmark, comparison of CUDA implementation with tools results.

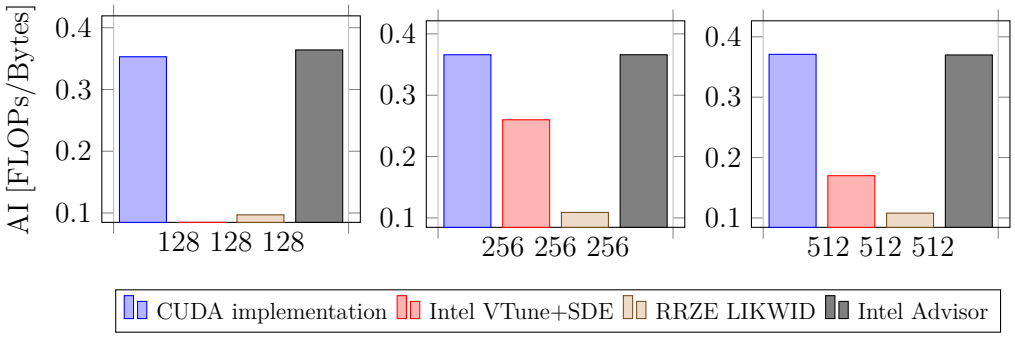


Figure 4.6: *laplacian* benchmark, comparison of CUDA implementation with tools results.

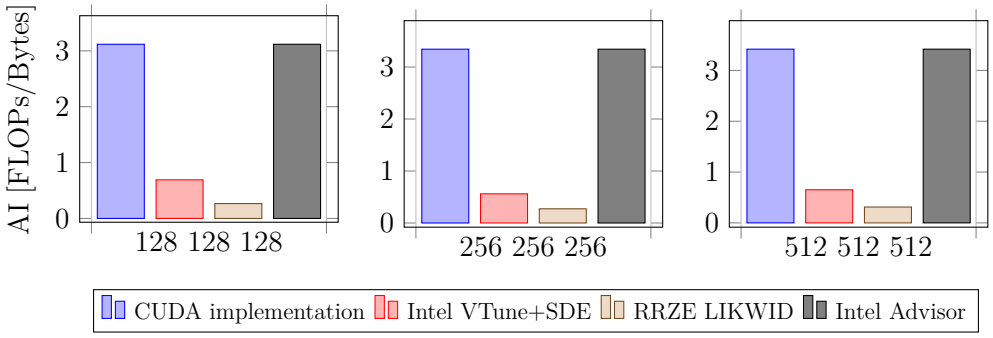


Figure 4.7: *lapgsrb* benchmark, comparison of CUDA implementation with tools results.

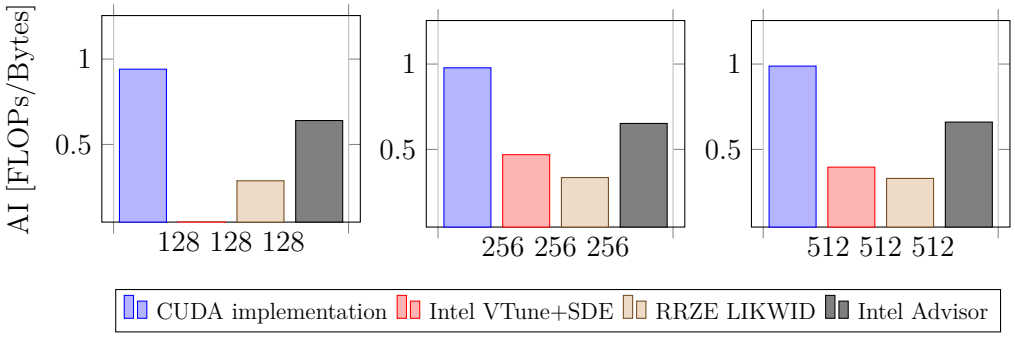


Figure 4.8: *matvec* benchmark, comparison of CUDA implementation with tools results.

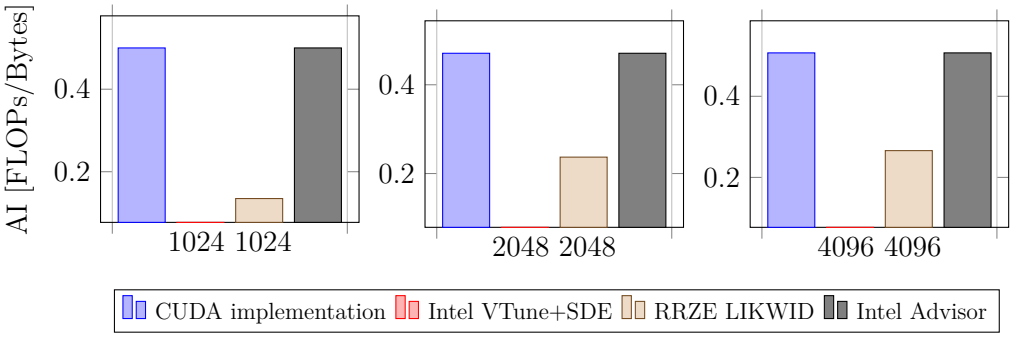


Figure 4.9: *tricubic* benchmark, comparison of CUDA implementation with tools results.

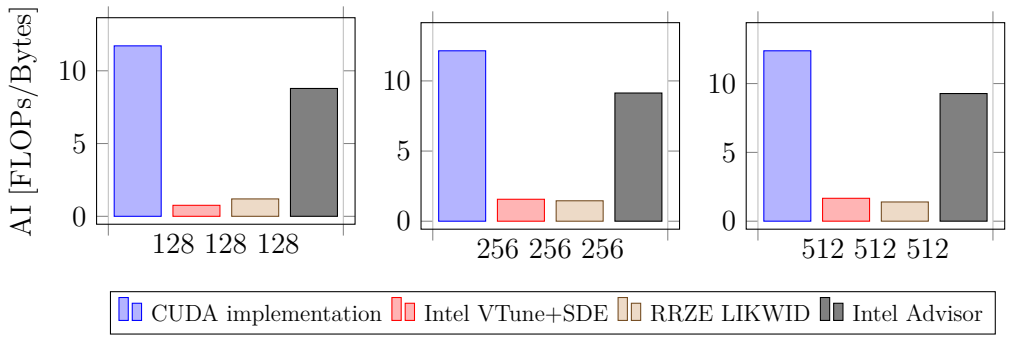


Figure 4.10: *uxx1* benchmark, comparison of CUDA implementation with tools results.

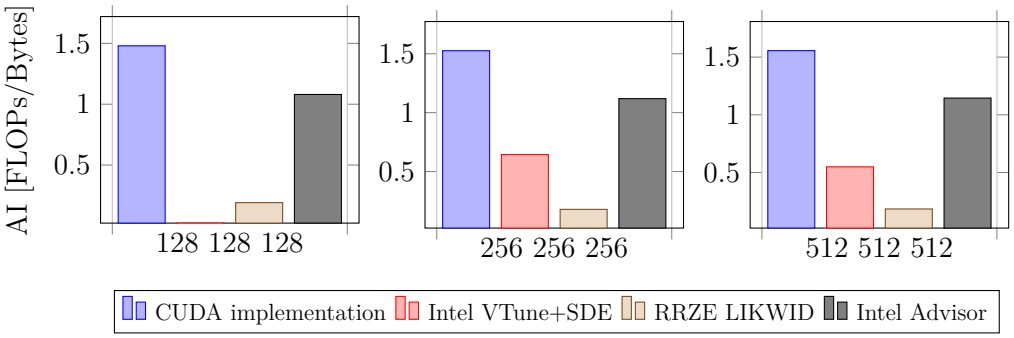


Figure 4.11: *vecadd* benchmark, comparison of CUDA implementation with tools results.

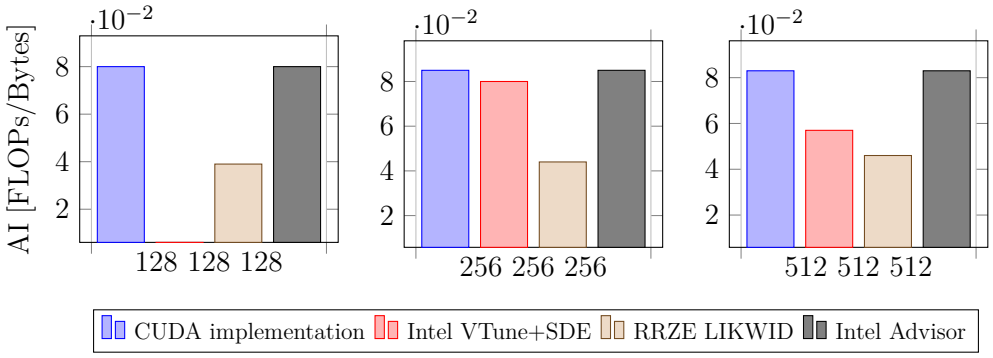


Figure 4.12: *wave13pt* benchmark, comparison of CUDA implementation with tools results.

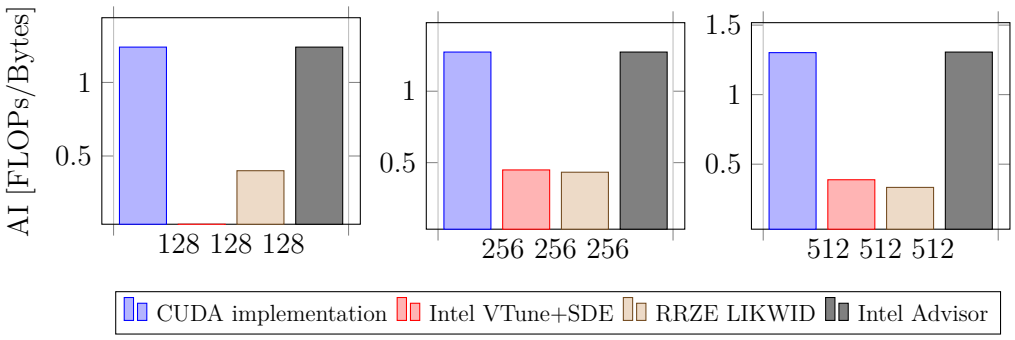
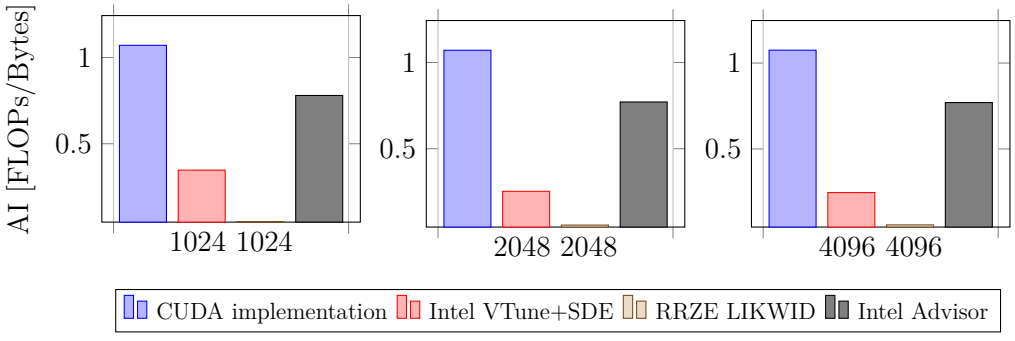


Figure 4.13: *whispering* benchmark, comparison of CUDA implementation with tools results.



Chapter 5

Exploring GPU Offload Opportunities in OpenTimer: A Case Study

This chapter builds upon the concepts and methods introduced in the previous ones by applying them to a real-world scenario. The challenge is to identify algorithms and loops that could benefit from GPU execution in projects where the code base is large and not all known beforehand. As a case study, we chosen the Static Time Analysis (STA) tool OpenTimer [43]. It has been selected because it is prominent in the open-source Electronic Design Automation (EDA) world, with multiple awards won in CAD contests, and it operates in a similar area as the Synopsys team where my internship was carried out. Additionally the source code is completely available on GitHub.

To identify and evaluate functions that could be offloaded to the GPU, we used the Offload Modeling analysis of Intel Advisor and the Arithmetic Intensity evaluation method introduced for the tool.

We expanded the analysis by implementing one of its core function to run on the GPU, using the OpenACC directives, and explored developing techniques as batching.

Finally, we illustrate two different methods to compile the project enabling GPU acceleration. The first method uses a GCC version with native NVPTX extension for OpenACC support, and the second method involves mixing GCC with NVC compilation. A final version of the code can be found in the GitHub repository in [10].

5.1 Introduction to Static Time Analysis (STA) and OpenTimer

OpenTimer is a tool for Static Time Analysis (STA) in electronic design, helping designers validate the timing of integrated circuits. Static Time Analysis (STA) is a crucial step in the Electronic Design Automation (EDA) process as it ensures that the circuit meets timing requirements and operates correctly under all possible conditions [4]. By checking

all paths for timing violation under worst-case conditions, OpenTimer ensures the data is presented at the input of each synchronous device when the clock edge arrives. Compared to dynamic simulation, static timing analysis can quickly evaluate timings without simulating logical operations, resulting in faster evaluation times [16].

OpenTimer is written in modern C++17 and uses CMake to manage the compiling process. Internally it holds the circuit structure as a Direct Acyclic Graph (DAG) $G = \{P, E\}$ where P is the pin set and E is the edge set. Each pin p in the graph can be levelized by a level index "level[p]" in order to maintain a topological order between different pins. Timing is propagated level by level keeping the circuit topology intact [15].

The tool offers a shell for interaction through command-line interface. Every timing operation executed through the shell can be interpreted in three distinct forms:

- *Builder*: OpenTimer creates a graph of operations to be performed and adds tasks every time a builder operation is called. The function to insert a gate in the circuit or read a cells library, for example, are builder operations;
- *Action*: An action triggers the timing update for each task in the graph upon the one that produces the result of the call. The command to report the timing of the circuit is an action;
- *Accessors*: queries the timer without altering the internal data structure. For example, the command to dump the timing graph is considered an accessor.

The OpenTimer shell can be launched using the *ot-shell* command from a system terminal, and the following builder commands can be used to load data into the graph from industry-standard input files:

- *read_celllib <string>.lib*: reads a liberty format library file. It contains an ASCII representation of timing and power parameters associated with any cell in the semiconductor, obtained by simulating the component in different conditions;
- *read_verilog <string>.v*: imports a gate level Verilog netlist and initializes the circuit graph from the given libraries;
- *read_spef <string>.spef*: applies the design parasitics of a set of nets as a resistive-capacitive (RC) network;
- *read_sdc <string>.sdc*: reads a Synopsys design constraint file, containing the initial timing conditions to impose on the design. It is a TCL-based format to specify design intent and constraints for synthesis, clocking, timer, power and area.

After importing the design characteristics, a range of actions can be performed. For example, by utilizing the *report_timing* command, as demonstrated in Listing 5.1, on the "simple" design - a benchmark integrated into the OpenTimer project with the layout depicted in Figure 5.1 - the analysis displays the most critical path on the screen. This path has the maximum delay between the input and output in the circuit.

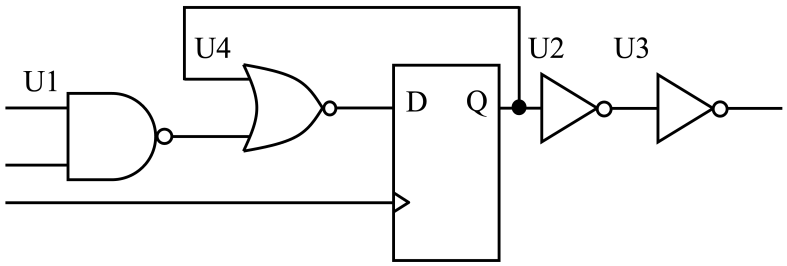


Figure 5.1: Layout of the OpenTimer "simple" benchmark.

Listing 5.1 OpenTimer results for the report_timing command on the "simple" design. *Source: OpenTimer GitHub* [43].

```
ot> read_celllib osu018_stdcells.lib
ot> read_verilog simple.v
ot> read_sdc simple.sdc
ot> report_timing

Startpoint    : inp1
Endpoint      : f1:D
Analysis type : min
-------------------------------------------------------
       Type        Delay        Time   Dir  Description
-------------------------------------------------------
       port        0.000       0.000  fall  inp1
        pin        0.000       0.000  fall  u1:A (NAND2X1)
        pin        2.786       2.786  rise  u1:Y (NAND2X1)
        pin        0.000       2.786  rise  u4:A (NOR2X1)
        pin        0.181       2.967  fall  u4:Y (NOR2X1)
        pin        0.000       2.967  fall  f1:D (DFFNEGX1)
    arrival                    2.967        data arrival time

related pin      25.000      25.000  fall  f1:CLK (DFFNEGX1)
 constraint       1.518      26.518        library hold_falling
   required                   26.518        data required time
-------------------------------------------------------
      slack                 -23.551        VIOLATED
```

In the example analysis, the critical path is originated from the *inp1* port and feed into the data pin *f1:D* of the *DFFNEGX1* flip-flop. The slack, which is the difference between required time and the arrival one, has been violated. Therefore, the circuit designer must take corrective measures.

A range of other actions is available, including:

- *report_at*: reports the arrival time at a pin;
- *report_slew*: reports the transition time at a pin;
- *report_slack*: reports the slack of a pin;

- *report_leakage_power*: reports the aggregate cell leakage power of the design.

5.2 Identification of Offload Opportunities with Intel Advisor

Our current objective is to pinpoint potential GPU offload opportunities within the code of OpenTimer. Due to the large size of the code base and our lack of prior knowledge of the code, we will rely on the automatic evaluation conducted by Intel Advisor "Offload Modeling" analysis. Section 3.3.1 contains additional information on the tool and provides guidance on how to perform the evaluation.

Advisor analysis deems a loop profitable and recommended for offloading if the ratio between the time required on the baseline CPU and the time estimated for the modelled GPU is greater than 1. Table 4.1 provides information on the CPU characteristics of the machine used for the analysis, whereas Table 5.1 outlines the specifications of the top-of-the-line Intel XeHPG 512 GPU modeled from the tool.

Table 5.1: Technical features of the GPU modeled by Intel Advisor.

GPU Model	Intel XeHPG 512
Architecture	Xe-HPG
Frequency [GHz]	1.80
L1 Cache Size [MB]	6
L1 Cache Bandwidth [TB/s]	14.75
L3 Cache Size [MB]	16
L3 Cache Bandwidth [TB/s]	3.69
Memory Type	GDDR6
Memory Bandwidth [GB/s]	460.8

To perform the analysis, we selected the "aes_core" benchmark and used the report_timing action in OpenTimer. This design consists of 20795 components that work together to provide encryption of digital data using the AES symmetric encryption algorithm. This algorithm uses a block cipher to encrypt and decrypt data, which is performed by the circuit on a block of plaintext. To accomplish this, a secret key is shared between the sender and receiver of the data.

In addition to the Verilog design, we loaded both early and late libraries for the circuit. The early library represents a best-case scenario where signals propagate faster than typical values, while the late library represents the opposite. By loading both libraries, the developer can ensure that the design meets timing requirements under all conditions. Before conducting the analysis, we also loaded the file containing the parasitics and the one with the constraints that specify the timing conditions to impose on the design.

After completing the analysis and opening the results with Advisor, as shown in Picture 5.2, the tool estimated that the application would run 10.274 times faster compared to using the CPU alone and identified 19 offload opportunities.

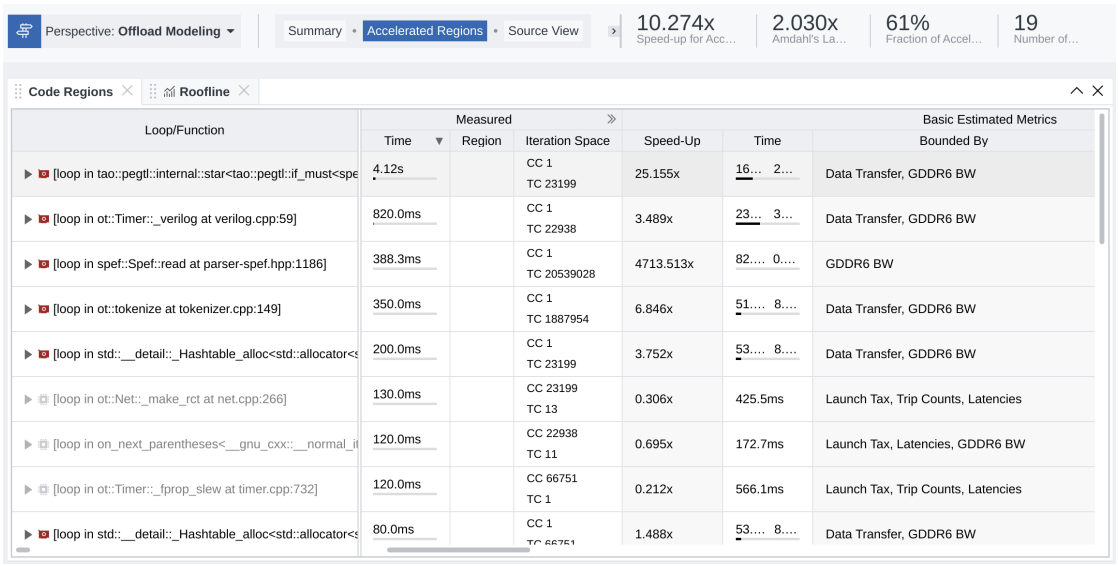


Figure 5.2: Screenshot of the Intel Advisor results page for the "report_timing" action in OpenTimer on the "aes_core" design.

We have selected the top five loops/functions based on the Intel Advisor estimated benefit from GPU offloading. We aim to study each of them to determine whether they can be actually accelerated or not. The functions are presented in Table 5.2, which is ordered by $(CPU\ time) - (GPU\ estimated\ time)$. This order represents the potential gain that the function can achieve following Amdahl's law.

Table 5.2: First five loops suggested to offload by Intel Advisor with OpenTimer analyzing the "aes_core" benchmark.

Loop/function	CPU time [ms]	GPU estimated time [ms]	Saved time [ms]	Speed Up
Loop at star.hpp:39	4120.0	163.8	3956.2	25.155x
Loop at verilog.cpp:59	820.0	235.0	585.0	3.489x
Loop at parser-spef.hpp:1186	388.3	0.824	387.476	4713.513x
Loop at tokenizer.cpp:149	350.0	51.1	298.9	6.846x
Loop at hashtable_policy.h:2120	200.0	53.3	146.7	3.752x

5.2.1 Loop at *star.hpp:39*

The first loop suggested for GPU offloading, the most profitable for Intel Advisor, is contained in the file "ot/parser-spef/pegtl/pegtl/internal/star.hpp" at the line 39.

To obtain additional information about the history calls that occur before the loop, we can open the "Top Down" tab, which is part of the accelerated regions screen in Intel

Advisor, as shown in Picture 5.3. Examining the call stack in reverse can helps us to gain a better understanding of the loop execution and its characteristics. By studying the stack we can determine components and data on which the loop operates, and we found that it belongs to the handling stage of the parasitics information, contained in the ".spef" file that was loaded into the OpenTimer shell at the beginning.

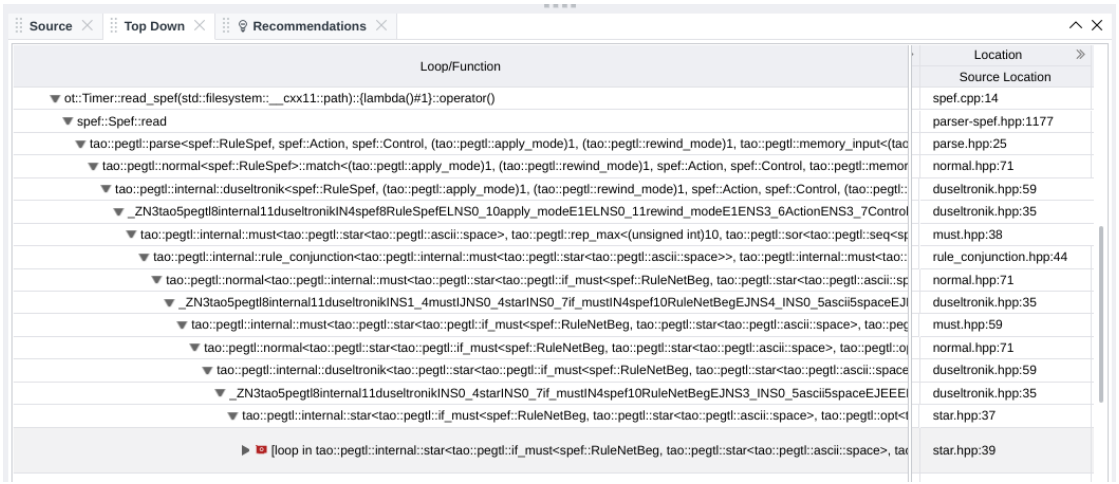


Figure 5.3: Intel Advisor "Top down" tab showing the functions call stack before the loop at star.hpp:39.

Prior to the loop, the file has been loaded into a string buffer in memory which is then used to perform parsing. A parsing operation analyzes the incoming data flow to determine its correctness and to make it machine-readable. This phase is handled by an external library called PEGTL, a C++ parser combinator library. Although included in the project, the library is not written by the OpenTimer developers themselves but by a third-party.

Although Intel Advisor recommends offloading the code region, it would not be appropriate to do so since the suggested loop belongs to an external library. As the developers are not the owners of this code, it would be difficult for them to optimize it for GPU execution.

Intel Advisor was unable to recognize that the function was part of an external library and not of the OpenTimer project. It is possible to exclude external libraries from the Offload Modeling by adding the parameter "–exclude-files=<path>" before starting the analysis, where *path* is a comma-separated list of file directories to exclude.

5.2.2 Loop at *verilog.cpp:59*

The loop presented is a crucial component of the functions that handle the gate level Verilog net list, and its primary purpose is to store the collected data into an internal class. It is located within the file "ot/timer/verilog.cpp", beginning line 59.

To give some context, when a Verilog file path is inserted into the shell, a called "module" data structure is allocated within OpenTimer, and the file path is passed to a function that parses it and populates the module. In essence, a module represents a block that defines a design, such as "aes_core," and its data structure consists of the following components:

- The module name;
- A vector of strings called "wires", which represents the internal nets used for physical connections;
- A vector of strings called "inputs", which contains the input pins;
- A vector of strings called "outputs", containing the output pins;
- A vector of strings called "ports", for all the cell pins in the module;
- A vector of objects called "gates", tracking a cell with its information. It contains two C++ unordered map to keep track of the connections between a net and and its input and between its output and another net.

Subsequently, OpenTimer proceeds to update the circuit by reading all inputs, outputs, and wires from the module data structure and storing the obtained data within an object of the class "Timer," which is defined in the file "timer.hpp." This stored data will be used for future applications. The loop that Intel Advisor suggests offloading is responsible for reading the "gate" vector from the "module" object and storing its details in the "Timer" object. The code snippet of the loop identified is contained in Listing 5.2.

Listing 5.2 Code snippet containing the loop at *verilog.cpp:59* suggested for offloading by Advisor.

```
for(auto& gate: module.gates){
    _insert_gate(gate.name, gate.cell);
    for([c, n]: gate.cellpin2net){
        auto& pin = _insert_pin(gate.name + ":" + c);
        auto& net = _insert_net(n);
        _connect_pin(pin, net);
    }
}
```

Let's take a detailed look at the loops and functions contained in the code snippet to determine whether there is a potential benefit in running them on a GPU:

1. The outer for loop iterates on all the gates contained in the module object:
   (a) The function *"insert_gate"* defined in "timer.cpp:135" creates a new Gate object that is not yet connected to other gates or wires. The function takes two parameters: the name of the Gate and a Cell. The newly created Gate object is then added to the unordered map "_gates" declared in "timer.hpp," which contains a tuple of the Gate name and Gate object. The Cell object is filled

using the data structure containing the cell pins called "_celllib." Pins of the gate are added using the "_insert_pin" function, and a reference of them is included inside the Gate to keep track. Finally, a function initializes the arc by creating a correspondence between each cell and its pins and the pin with the cells connected.

2. The inner foor loop iterates on each Pin-Net mapping specified in the gate:

   (a) The *"_insert_pin"* function, defined in timer.cpp:421, checks if the passed string, composed of the gate name plus the cell pin, already exists in the unordered map "_pin". The "_pin" map contains objects of type Pin and strings that hold the pin identifier. If the string exists in the map, the function returns a reference to the corresponding object. However, if the string does not exist, a new Pin object is created and added to the list of pins called "_frontiers". Finally, the function returns a reference to the newly created Pin object.

   (b) *"_insert_net"*, defined in "timer.cpp:386," creates and returns a reference to an object of type "Net" with the name passed as input. This function also adds this information to the unordered map "_nets". The newly created Net object is not yet connected to any pin or loaded with parasitic information.

   (c) *"_connect_pin"*, located at "timer.cpp:291", creates a connection between a pin and the corresponding net, creating an arc. First, the reference to the pin is inserted into the Net object. Then, the function "_insert_arc" is called with the pin and the net as input parameters to store this correspondence in the list of "Arc" objects called "_arcs". Finally, this function inserts the fanin and fanout references into their corresponding data structures in the Pin class.

The two loops perform complex operations that involve filling multiple unordered_map and lists, rather than arithmetic computations. Although Advisor reported a very high trip count of 22938, it is not beneficial to offload the loops to the GPU due to the overhead of transferring all the data to perform non-trivial operations on C++ data structures. Therefore, despite Advisor's suggestion, the loops may not be suitable for offloading to the GPU.

5.2.3 Loop at *parser-spef.hpp:1186*

The third loop identified by Advisor is part of the manipulation process of the spef file, which contains the parasitics information related to the module. This loop is executed in the same flow as the first one evaluated, which is located at *star.hpp:39*. As previously mentioned, the spef file is read and stored in a string buffer variable in memory before it is parsed by the PEGTL library. However, if any comment is present in the file, indicated by the characters "//" at the start of the line, the subsequent characters until the new line operator are replaced with a space to prevent any disruption during the parsing phase. A code snippet containing the relevant loops is provided in Listing 5.3.

Listing 5.3 Code snippet containing the loop at *parser-spef.hpp:1186* suggested for offloading by Advisor.

```
1  for(size_t i=0; i<buffer.size(); i++){
2    if(buffer[i] == '/' && i+1 < buffer.size() && buffer[i+1] == '/') {
3      buffer[i] = buffer[i+1] = ' ';
4      for(i=i+2; i<buffer.size(); ++i) {
5        if(buffer[i] == '\n' || buffer[i] == '\r') {
6          break;
7        }
8        else buffer[i] = ' ';
9      }
10   }
11 }
```

Although the string contained in the buffer is accessed character by character as a vector in this operation, it is not possible to offload it to a GPU to take advantage of its SIMD architecture because of the break statement in line 6, which is used to exit the loop once the newline "$\backslash n$" or carriage return "$\backslash r$" character is encountered. This ensures that the internal loop stops as soon as all comments have been removed and can start looking for "//" characters in the vector again. Unfortunately, threads running in parallel would not be aware of this interruption, and they might already be removing data from a later element of the vector since the comment's length is not known in advance. This could lead to the removal of critical information about the parasitics, making the loop unsuitable to offload on a GPU.

5.2.4 Loop at *tokenizer.cpp:149*

This function recommended by Advisor, is located at line 149 of "ot/utility/tokenizer.cpp" file in the OpenTimer installation folder. The function is part of the execution flow that reads the Verilog file passed as input from the shell, and creates a "module" object containing the Verilog information about the design of "aes_core". This module is the same as the one mentioned in the previous function identified by Advisor for the analysis of the loop at "verilog.cpp:59", but this operation occurs at an earlier stage.

The file is first opened and its contents are stored in a buffer. Then, a tokenization operation is performed on the characters in the buffer, which involves splitting the text into tokens based on pre-defined delimiters like "input", "output", or "wire" and removing comments. These tokens are subsequently used to populate the data structures. The loop suggested by Advisor is part of a function that takes three arguments: the file path, a string of delimiters that signal the end of a token, and a string of exceptions that specifies a delimiter that should also be included in the token before it is closed.

After the file is opened and saved into a buffer, an operation of tokenization takes place on the characters contained in the buffer, splitting it into tokens based on predefined delimiters, such as "input", "output", or "wire", which are later used to fill the data structures, and removing comments. The function which the loop is part of, with a simplified code snippet presented in Listing 5.4, takes three arguments: the path to the file, the string of delimiters, that indicate when a token ends, and the string of exceptions, which contains a delimiter that must also be saved in the token before closing it.

Listing 5.4 Code snippet containing the loop at *tokenizer.cpp:149* suggested for offloading by Advisor.

```
1  std::string token;
2  std::vector<std::string> tokens;
3
4  for(size_t i=0; i<file_size; ++i) {
5    bool is_delimiter = (delimiter.find(buffer[i]));
6    if(is_delimiter || std::isspace(buffer[i])) {
7      if(!token.empty()) {
8        tokens.push_back(std::move(token));
9        token.clear();
10     }
11     if(is_delimiter && exceptions.find(buffer[i]) != -1) {
12       token.push_back(buffer[i]);
13       tokens.push_back(std::move(token));
14     }
15   } else {
16     token.push_back(buffer[i]);
17   }
18 }
```

To provide a more detailed explanation, the loop takes each character from the buffer individually and performs the following operations:

- If the character is a delimiter or a whitespace, and the current token is not empty, the token is stored to a vector of tokens and then it is cleared, ready to temporarily hold new characters;
- If the character is both a delimiter and an exception, it is saved in the current token, and the token is then pushed into the vector of tokens;
- If the character is not a delimiter, it is added to the current token;

After executing the tokenization loop, the buffer containing the file will be transformed into a vector of tokens. However, parallelizing this loop on a GPU is not possible due to the data dependency between the current character being processed and the previous ones. Parallel execution would disregard this dependency, potentially leading to incorrect tokenization results. Therefore, preserving the order of character processing is crucial to properly construct the graph representing the circuit for the "aes_core" module.

5.2.5 Loop at *hashtable_policy.h:2120*

The last of the five functions suggested for GPU offloading by Advisor, is the loop at the line 2120 of "hashtable_policy.h".

This function is a component of the GNU ISO C++ library, commonly referred to as *libstdc++*. It is utilized in the phase detailed in Subsection 5.2.2 about the loop at verilog.cpp:59, when the gates are added to an "unordered_map", that is a data structure implemented as a hash table, and the hashtable_policy.h file provides the classes that control its behavior.

Unfortunately, also in this instance, the tool did not provide accurate or constructive suggestions, thus making it arduous for developers to implement the proposed advice on GPU.

5.3 Sorting Analysis Results by Arithmetic Intensity

The Offload Modeling analysis conducted by Advisor was not successful as the five best suggested loops for offloading, despite having a high trip count, could not be accelerated on the GPU due to the reason previously explained, which were discovered after a detailed analysis of their behavior.

As a result, we decided to order the 19 identified loops by their Arithmetic Intensity (AI), as detected by the tool, instead of by speed up. This change in approach was made to determine if the utilization of AI as evaluation parameter, which is it is hardware-independent, can aid in identifying functions that are suitable for offloading on the GPU and evaluate whether execution times can improve.

Table 5.3 contains the first five functions for AI. Although Advisor identified them, they are marked as top not-offloaded since the speed up evaluated is deemed not advantageous. Specifically, the ratio between the CPU execution time and the GPU execution time is estimated to be less than 1.

Table 5.3: Top five results ordered by Arithmetic Intensity from the "report_timing" analysis of Intel Advisor on the "aes_core" benchmark.

Loop/function	GFLOPs	Memory Accesses [GB]	Arithmetic Intensity [FLOPs/Bytes]
Loop in Rct::_update_ldelay at net.cpp:160	0.012	0.083	0.145
Loop in Rct::_update_response at net.cpp:178	0.006	0.083	0.072
Loop in Rct::_update_load at net.cpp:126	0.006	0.083	0.072
Loop in Timer::_fprop_delay at timer.cpp:746	0.003	0.240	0.013
Loop in Rct::_update_delay at net.cpp:143	0.001	0.083	0.012

Despite the tool modeled speedup not being deemed beneficial and the estimated AI value not being particularly high, it is still worthwhile to study the functions in order to identify any potential offload opportunities. This will allow us to verify the accuracy of the Advisor analysis and determine whether GPU acceleration can still provide any benefits in the application execution.

We have observed that four out of the five functions for Arithmetic Intensity belong to the same C++ class "Rct", which is responsible for the RC propagation phase of the circuit. To gain a better understanding of this class and its role, we need to take a step back in the OpenTimer execution flow.

In the presentation paper authored by Huang et al. [15], it is explained that OpenTimer represents a circuit as a DAG $G = \{P, E\}$ where P is the sets of pins and E is the set of edges. To enable temporal propagation, OpenTimer constructs a data structure known as a "bucket list". Each bucket is assigned to a level index "l" and contains a list of pins that belong to the same level as "l". This allows to maintain a levelized structure while preserving a topological order to propagate the timing level-by-level.

When a pin is levelized, starting from the lowest level, its fanout (the number of input gates connected to the output of a single gate) is inserted in the bucket list.

The levelized bucket list facilitates the execution of the forward time propagation procedure, which performs six tasks for each pin in the bucket list for every level:

1. RC propagation;
2. Slew propagation;
3. Delay propagation;
4. Arrival time propagation;
5. Jump point propagation (contraction of the graph to reduce the search space);
6. CPPR credit propagation (removal of some pessimism for the timing test).

The loops within the "Rct" class identified by Advisor are all part of the first phase enumerated, responsible for updating the RC parameters necessary for the propagation of the slew and delay through a net. This process begins with reading the ".spef" file containing the parasitic design of the module, modeled as a resistive-capacitive network that includes the capacitance of internal gates and resistance of the wires between them.

OpenTimer estimates the output slew and delay of the RC network by computing the the first and second moment of the impulse response and taking their symmetric value.

The objective of this propagation is to calculate the RC parameters for each RC net, and to evaluate the delay and impulse between the root and every output pin.

Now, we will examine the operations performed by the first function based on the AI value, "Rct::_update_ldelay". Our goal is to assess the feasibility of offloading the function to the GPU and determine the potential benefits of executing it on the device.

5.3.1 Loop at *net.cpp:160*

The first loop listed for Arithmetic Intensity pertains to the "Rct::_update_ldelay" function, which can be found in the file "OpenTimer/ot/timer/net.cpp:160".

To estimate the timing of the circuit, OpenTimer performs the following steps when implementing the RC delay computation on the graph:

1. For each node in the circuit, the load is computed;

2. The delay between the port and the node is calculated;

3. The sum of the product of capacitance and delay in the subtrees of the node is computed, which is referred to as load delay (ldelay);

4. Using the "ldelay" obtained in step 3, the delay and impulse between the port and the node are calculated.

The loop identified by Intel Advisor computes the load delay in the subtrees of every node, as explained in point three. The source code of the function is presented in Listing 5.5.

Listing 5.5 Function to compute the load delay by OpenTimer, located in *net.cpp:160*.

```
void Rct::_update_ldelay(RctNode* parent, RctNode* from) {

  for(auto e : from->_fanout) {
    if(auto& to = e->_to; &to != parent) {
      _update_ldelay(from, &to);
      FOR_EACH_EL_RF(el, rf) {
        from->_ldelay[el][rf] += to._ldelay[el][rf];
      }
    }
  }

  FOR_EACH_EL_RF(el, rf) {
    from->_ldelay[el][rf] += from->cap(el, rf) * from->_delay[el][rf];
  }
}
```

The function performs a Depth-First Search (DFS) traversal of the RC tree of each node in the circuit. DFS is an algorithm that starts from the root of the tree and explores as far as possible along each branch before backtracking. Figure 5.4 shows the traversal order performed by the algorithm on an example graph.

During backtracking, the function adds up the load delay from the most recent visited node, which has accumulated the values of the previously visited nodes, to the current one. Then to obtain the load delay, the function sums the value with the product of capacitance and delay.

Every time a node is visited, these operations are performed for the four combinations of early/late and rise/fall values. The "FOR_EACH_EL_RF(el, rf)" function is a user-defined loop that iterates on the four values without having to write them out each time, as they are just a combination of zeroes and ones.

As the operation does not present any significant impediments to run on a GPU, apart from the required execution time which we will evaluate later, we can implement a version that can be offloaded to accelerate the code. By introducing techniques that are accessible to developers new to the world of GPU programming, we can offload the code without requiring any particular previous knowledge of the subject.

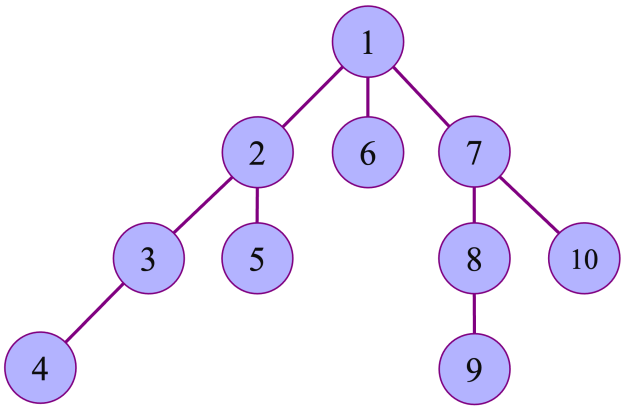


Figure 5.4: Example of a traversal sequence of nodes using the DFS algorithm.

5.4 Porting OpenTimer RC Delay Function to GPU

Our goal is to offload the "Rct::_update_ldelay" function, which computes the load delay in the subtrees of each node in a circuit, to the GPU. However, the function performs a DFS traversal and recursive operations like this are typically not well-suited for GPU acceleration due to the high frequency of memory access during the tree traversal, which can result in expensive data transfers.

While we don't want to completely rewrite the OpenTimer structure, we do want to explore GPU execution methodologies. To achieve this, we will use a batching technique, which involves accumulating data before offloading them to the GPU. By building a data structure to process a set of elements at the same time, we can take advantage of GPU parallelism and manipulate the batched workloads all together, which can reduce data transfer and improve performance [7]. This will enable us to implement the function on GPU without the need to completely rewrite the OpenTimer structure.

To accomplish this, rather than performing the operation at line 7 of Listing 5.5 directly on GPU, which would be not convenient, we can accumulate the values on a support data structure while traversing the tree and then compute them in batches when the node is reached during backtracking.

An updated version of the function with these changes implemented can be found in Listing 5.6

Listing 5.6 Rct::_update_ldelay function rewritten implementing batching technique on data.

```
1 void Rct::_update_ldelay(RctNode * parent, RctNode * from) {
2   int i, j;
3   std::vector<std::vector<float>> to_ldelay_acc;
4
5   for (auto e: from->_fanout) {
6     if (auto & to = e->_to; & to != parent) {
```

```
7         _update_ldelay(from, & to);
8         std::vector <float> to_ldelay_RctNode;
9         FOR_EACH_EL_RF(el, rf) {
10          //Save current "to._ldelay" values into a temporary vector
11          to_ldelay_RctNode.push_back(to._ldelay[el][rf]);
12        }
13        //Push the vector inside an accumulator data structure
14        to_ldelay_acc.push_back(to_ldelay_RctNode);
15      }
16    }
17
18    //Save the number of nodes visited and allocate temporary vectors
19    int num_nodes = to_ldelay_acc.size();
20    float *to_float_star = new float[num_nodes * 4];
21    float from_ldelay[4], from_cap[4], from_delay[4];
22
23    //Move the accumulator to a simpler vector of float
24    for (i = 0; i < num_nodes; i++) {
25      for (j = 0; j < 4; j++) {
26        to_float_star[i * 4 + j] = to_ldelay_acc[i][j];
27      }
28    }
29
30    //Save delay and capacitance to temporary vectors
31    FOR_EACH_EL_RF(el, rf) {
32      from_cap[el*2+rf] = from->cap(el, rf);
33      from_delay[el*2+rf] = from->_delay[el][rf];
34    }
35
36    //Loop to be accelerated using OpenACC directives
37    for (i = 0; i < 4; i++) {
38      float acc = 0.0;
39      for (j = 0; j < num_nodes; j++) {
40        acc += to_float_star[i + j * 4];
41      }
42      from_ldelay[i] = acc + from_cap[i] * from_delay[i];
43    }
44
45    delete[] to_float_star;
46
47    //Copy the result back to the OpenTimer data structure
48    FOR_EACH_EL_RF(el, rf) {
49      from->_ldelay[el][rf] = from_ldelay[el * 2 + rf];
50    }
51
52  }
```

The following operations were applied to obtain the code that will support GPU acceleration:

1. Instead of computing the operation directly for each visited node, we saved the four float values representing the combinations early/late and rise/fall contained inside the "to._ldelay" vector into a temporary C++ std::vector;

2. Before continuing the tree traversal, we pushed the vector into an accumulator vector<vector<float» that keeps track of all visited nodes. We used this dynamic data structure because it does not require advance knowledge of the tree depth and the number of nodes that will be added;

3. When backtracking starts, we move the accumulator std::vector to a dynamically allocated vector of float to be sent to the GPU. We translate the 2-dimensional data structure into a single vector, where we perform a column by moving 4 values at a time. While C++ data structures can be moved to the GPU, they require compiling the entire project with a NVIDIA Compiler and CUDA Unified Memory. To minimize the impact on large projects, in Section 5.6 we propose alternative compilation methods;

4. We also add the sum of the product of capacitance and delay for the node to the operations to be performed on the device. Since they are contained in complex data structures, we temporarily save them in vectors of float;

5. From line 31 to 37 of Listing 2, two nested loops perform all the necessary calculations, ready to be offloaded to the GPU using the OpenACC directives, introduced in Section 5.5;

6. The results of the computation are contained in a vector of four elements, which is the only one copied back from the device to the host. The values can be moved to the OpenTimer data structure to continue the computation correctly.

The loop to be offloaded can be parallelized across different GPU threads because the four combinations of early/late and rise/fall are independent, allowing for vectorization. Vectorization is another technique to further increase GPU performance by reorganizing the data in memory so that it can be accessed efficiently by the parallel processing units, rewriting the loop logic to take advantage of the parallelism.

In the following section, we introduce the use of OpenACC directives, that will enable us to instrument the code allowing the compiler identify which data structures to move to the GPU and which computations can be accelerated.

5.5 Exploring OpenACC for Heterogeneous Computing

OpenACC [40] is an open standard platform-independent programming model that enables programmers to accelerate High Performance Computing (HPC) applications written in C/C++ and FORTRAN. It is supported by a wide range of compilers that can automatically generate parallel code, handle data movement, and enable various features like detecting data dependencies between loops. Compared to CUDA, which is specifically designed for NVIDIA devices and offers fine-grained control over the hardware, OpenACC is a more accessible option.

OpenACC high-level compiler directives are not device-dependent and can be used on a variety of hardware architectures provided by different vendors, such as NVIDIA, AMD,

and Intel. This make it easier to program GPU code, as it can be added to existing code base with minimal modifications, simplifying the process of adapting existing application to exploit GPU parallelism.

In order to enable acceleration using OpenACC, the source code needs to be instrumented to specify to the compiler which portions should be offloaded and which data structures copied to the device. In C++ this is done using pragmas. The keywords to highlight the code to be executed on the GPU begin with "#pragma acc" and to handle data movement with "#pragma acc data".

In particular, some of the pragmas that manage data movement between the host and the device are the following [41]:

- *copy*: allocates memory on the GPU and moves the specified data from the host to the device for computation. When completed copies the data back to the host;
- *copyin*: allocates memory on the device and moves the specified data from the host to the device for computation;
- *copyout*: allocates memory on the device and when the computation is completed moves the data to the host;
- *create*: allocates memory on the device without performing any data movement;
- *present*: informs the compiler that the specified data structure is already present on the device, and therefore it is not necessary to move it again.

Within the brackets of the OpenACC data movement pragmas it is necessary to specify the name of the variable to be transferred, along with its size if it is a vector. For example, to copy a five element vector V to the device, before the loop that deals with it the programmer would add the following pragma to the code: "#pragma acc data copyin(V[:5])".

Once data transfer is managed, programmers can specify the code section to be accelerated using various available pragmas. Some of these are as follows:

- *kernels*: identifies a region for parallel execution, but leaves it to the compiler to analyze and identify which loops are safe to accelerate;
- *parallel*: identifies a region to be parallelized, and the developer asserts that is safe to be accelerated. When paired with the *loop* directive, the compiler will generate a parallel version of the loop.

The main difference between the two directives is the level of control left to the compiler. The *kernels* keyword is a hint for the compiler about where it should look for parallelism, while the *parallel* directive is an assertion [41].

5.5.1 Accelerating OpenTimer with OpenACC Directives

We can now enable acceleration using OpenACC directives for the "Rct::_update_ldelay" function we rewrote and shown in line 37 of Listing 5.6, which takes care of summing the batched values and computing the product of capacitance and delay to obtain the load delay for each node in the tree visit. The loops become as follows:

Listing 5.7 Accelerated loops with OpenACC directives for the Rct::_update_ldelay function, ready to run on a GPU.

```
1  #pragma acc data copyin(to_float_star[:num_nodes*4], from_cap[:4],
       from_delay[:4])
2  #pragma acc data copyout(from_ldelay[:4])
3  #pragma acc parallel loop
4  for (i = 0; i < 4; i++) {
5      float acc = 0.0;
6      for (j = 0; j < num_nodes; j++) {
7          acc += to_float_star[i + j * 4];
8      }
9      from_ldelay[i] = acc + from_cap[i] * from_delay[i];
10 }
```

Specifically, we instructed the compiler to offload the loops by using the following OpenACC pragmas:

- *data copyin(to_float_star[:num_nodes*4], from_cap[:4], from_delay[:4])*: performs the allocation and transfer of vectors containing temporary data structures to the device. These structures hold the batched load delay values of the child nodes, as well as the capacitance and delay values for the four possible combinations of early/late and rise/fall transitions;

- *data copyout(from_ldelay[:4])*: allocates the data to contain the result load delay and transfers the value to the host once the computation is completed;

- *parallel loop*: asserts that the two nested loops can be parallelized on the GPU.

The number of bytes to be transferred from the device to the host is always 16, that is 4 values multiplied by 4 bytes, the size of a float number. However, the bytes copied from the host to the device cannot be predetermined, as it varies based on the size of the *float* to_float_star[num_nodes*4]* vector. For each node, the vector is dynamically allocated with a size depending on the number of the sub-nodes visited from the considered node. As this function is part of a tree traversal, multiple calls will occur, each leading in data movement on the GPU. Therefore, it is essential to optimize the performance as much as possible, as executions with low *num_nodes* value could be inefficient on the GPU due to the cost of data transfer. The number of sub-nodes for each node varies based on the complexity of the design being analyzed with OpenTimer.

For instance, in the analysis of "aes_core", the design that we worked with, the frequency distribution of the number of sub-nodes for each tree node is contained in Table 5.4.

Table 5.4: Frequency distribution of the number of sub-nodes for each tree node in the "aes_core" design.

Number of Sub-nodes	Occurrences
0	41297
1	287536
2	17912
3	90
4	2

An excessive amount of data movement can occur when performing operations with too few batched sub-nodes, which can significantly slow down the execution. To optimize the loop, we can study its Arithmetic Intensity and determine the threshold value of *num_nodes* below which offloading the operation would be too burdensome, making it more efficient to run it on the CPU.

To achieve this, we compiled Table 5.5 which includes values of FLOPs and memory accesses to calculate the AI based on the number of sub-nodes traversed. By examining the results, we can determine the optimal number of number of values the vector should contain to find a balance between minimizing data movement and maximizing computational efficiency.

Table 5.5: Arithmetic Intensity of the loop for different values of Sub-nodes.

Number of Sub-nodes	FLOPs performed	Memory Accesses [Bytes]	Arithmetic Intensity [FLOPs/ Bytes]
0	4	48	0.083
1	8	64	0.125
2	16	80	0.200
3	20	96	0.208
4	24	112	0.214
5	28	128	0.219

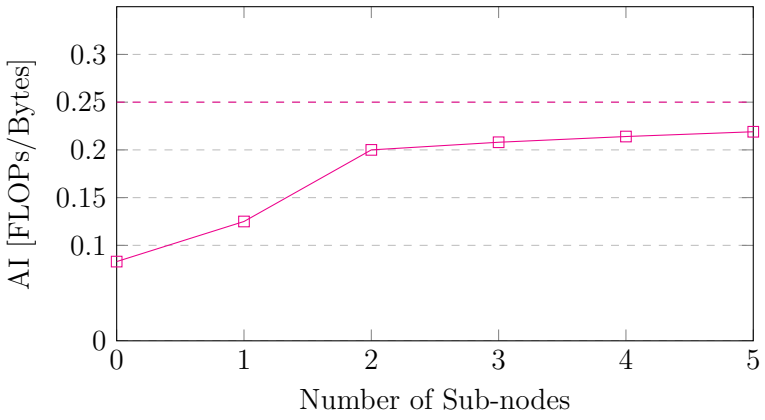
The analysis results reveal that AI value almost doubles, passing from 0.125 to 0.200, when the *num_nodes* values increases from 1 to 2. Then it it slowly reaches a saturation point at 0.250, as showed in Figure 5.5.

Based on these findings, we have decided to divide the function execution flow based on the number of sub-nodes batched in the vector. With *num_nodes* value up to 1, the function will continue to execute on the CPU, since the data transfer to the GPU would be not efficient. However, when the value is 2 or higher, the function will be offloaded to the GPU, to take advantage of its computational power.

Listing 5.8 contains the final version of the "Rct::_update_ldelay" OpenTimer function, optimized to run on the GPU with the OpenACC directives. This version is the

result of a thorough analysis and several iterations aimed at maximizing its performance on the GPU.

Figure 5.5: Values of Arithmetic Intensity for the function related to the number of sub-nodes.



Listing 5.8 Final version of the "Rct::_update_ldelay" function optimized to run on the GPU with the OpenACC directives.

```
1  void Rct::_update_ldelay(RctNode * parent, RctNode * from) {
2    int i, j;
3    std::vector<std::vector<float>> to_ldelay_acc;
4
5    for (auto e: from->_fanout) {
6      if (auto & to = e->_to; & to != parent) {
7        _update_ldelay(from, & to);
8        std::vector <float> to_ldelay_RctNode;
9        FOR_EACH_EL_RF(el, rf) {
10         to_ldelay_RctNode.push_back(to._ldelay[el][rf]);
11       }
12       to_ldelay_acc.push_back(to_ldelay_RctNode);
13     }
14   }
15
16   int num_nodes = to_ldelay_acc.size();
17   if (num_nodes > 1) {
18     /********* GPU EXECUTION - num_nodes>1 *********/
19     float *to_float_star = new float[num_nodes * 4];
20     float from_ldelay[4], from_cap[4], from_delay[4];
21
22     for (i = 0; i < num_nodes; i++) {
23       for (j = 0; j < 4; j++) {
24         to_float_star[i * 4 + j] = to_ldelay_acc[i][j];
25       }
26     }
27
28     FOR_EACH_EL_RF(el, rf) {
```

```
29        from_cap[el*2+rf] = from->cap(el, rf);
30        from_delay[el*2+rf] = from->_delay[el][rf];
31      }
32
33      #pragma acc data copyin(to_float_star[:num_nodes*4], from_cap[:4],
        from_delay[:4])
34      #pragma acc data copyout(from_ldelay[:4])
35      #pragma acc parallel loop
36      for (i = 0; i < 4; i++) {
37        float acc = 0.0;
38        for (j = 0; j < num_nodes; j++) {
39          acc += to_float_star[i + j * 4];
40        }
41        from_ldelay[i] = acc + from_cap[i] * from_delay[i];
42      }
43
44      delete[] to_float_star;
45
46      FOR_EACH_EL_RF(el, rf) {
47        from->_ldelay[el][rf] = from_ldelay[el * 2 + rf];
48      }
49
50    } else {
51      /******** CPU EXECUTION - num_nodes<=1 ********/
52      for (i = 0; i < num_nodes; i++) {
53        for (j = 0; j < 4; j++) {
54          from->_ldelay[i][j] += to_ldelay_acc[i][j];
55        }
56      }
57      FOR_EACH_EL_RF(el, rf) {
58        from->_ldelay[el][rf] += from->cap(el, rf) * from->_delay[el][rf];
59      }
60
61    }
62  }
```

5.6 Compile GPU Code with Minimal Project Impact

Adding GPU code to an existing project, particularly a large one, can present several challenges for developers. GPU programming requires specific libraries to communicate with the hardware and not all compilers support GPU acceleration. As a result, it may be necessary to make some changes to the build system in order to integrate GPU code.

OpenTimer compilation is performed via the CMake build system, which uses the CMakeLists text file to describe the build process. In this section, we present two methods that we believe offer significant advantages for integrating C/C++ code with OpenACC directives to be executed on NVIDIA GPUs. These methods are as follows:

1. Using GCC-11 compiler with NVPTX (Nvidia Parallel Thread Execution) extension,

we were able to compile the entire project without making any changes to the CMake build system already in place;

2. Compiling the GPU specific code using the NVIDIA NVC++ compiler, creating a static library, and then linking it with the rest of the project which is compiled using a traditional CPU compiler.

We have successfully compiled the GPU-accelerated version of OpenTimer using both methods, and provide a detailed explanation of each one in the corresponding subsections. Finally, we compare the performance of the two methods to determine which one provides the best results.

5.6.1 Using GCC Compiler with NVPTX Extension

To compile the accelerated version of OpenTimer, the first method we utilized relies on the GCC compiler. GCC, or the GNU Compiler Collection, is the default compiler for most Linux distributions and can be configured to target NVIDIA GPUs by using the NVPTX (Nvidia Parallel Thread Execution) extension [49], enabling the offloading of code with OpenACC support.

Using this method, it will be sufficient to use the compiler together with the appropriate compile flag to enable code acceleration. This eliminates the need to edit the OpenTimer CMakeLists file.

Since the GCC compiler does not come with built-in offload support, to simplify the configuration process we utilized Spack [50], an open source package management system. Once Spack is installed on the system, a version of the GCC compiler with the NVPTX extension can be setup by running the command:

Listing 5.9 Install GCC compiler including the NVPTX extension using Spack.

```
$spack install gcc+nvptx
```

Spack automatically creates a module in its environment that includes the specified version of the compiler. To load the environment in a terminal shell, simply run the following command:

Listing 5.10 Load the Spack environment in a terminal shell.

```
$. /spack/share/spack/setup/setup-env.sh
```

This will also automatically add the path to the executable files in the module to the *PATH* Linux variable. However, including the libraries used by the compiler to the *LD_LIBRARY_PATH* variable of the system is also necessary and must be done manually.

The compiler setup process is complete. To compile the entire OpenTimer project and accelerate all the loops that contain OpenACC directives, simply use the following flags while compiling the code:

Listing 5.11 Compile OpenTimer accelerated using OpenACC directives using GCC with NVPTX extension and flags.

```
$cmake ../ -DCMAKE_CXX_FLAGS="-fopenacc -offload=nvptx-none -fopt-info-optimized-omp"
```

where:

- *-fopenacc*: enables OpenACC directives;
- *-offload=nvptx-none*: specifies the target to enable code acceleration. In this case "nvptx-none" is a GCC configuration that targets NVIDIA PTX;
- *-fopt-info-optimized-omp*: enables OpenACC diagnostics also showing the loop parallelism assigned.

5.6.2 Creating an Accelerated Static Library with NVIDIA Compiler

In OpenTimer, we utilized a second method to enable GPU acceleration, which allows for the use of any compiler for the code that will run on the CPU, while utilizing the NVIDIA NVC compiler exclusively for the files containing the OpenACC directives. This can be accomplished by creating an accelerated static library that can be linked to the rest of the code. However, there is a trade off, as small modifications will be required in the CMake build system of the project.

To create the static library, the code file to be offloaded must be compiled with NVC++, a compiler developed by NVIDIA specifically for their hardware, which is part of the NVIDIA HPC SDK [48]. To achieve this, we created a "NetAccelerated" folder inside the OpenTimer project with a specific CMakeLists file that creates the library based on the "OpenTimer/ot/timer/net.cpp" file. Here are some of the steps involved:

- In the *CMAKE_CXX_FLAGS* the flag "-fPIC" is added to enable position independent code to create the library;
- We used *set(CMAKE_INSTALL_PREFIX $CMAKE_CURRENT_LIST_DIR/library)* to set the "./NetAccelerated/library" path as the directory for the install procedure;
- We used *add_library(NetAccelerated STATIC $OT_NET)* to build the static library called NetAccelerated that can be linked to other targets;
- The *target_include_directories* directive specifies the destination folder to use when compiling the library;
- Finally, *install(TARGETS NetAccelerated DESTINATION $CMAKE_INSTALL_PREFIX)* compiles the static library when the "make" command is run.

Since the entire configuration is non trivial, the complete CMakeLists file can be found on GitHub at [11].

Additionally, compared to the previous compilation method, NVC provides more detailed information about the loop parallelism during the compilation phase. These information are printed on the terminal screen and can be used to identify and improve the code during development. For instance, Listing 5.12 displays the diagnostic output for the accelerated loops in the "Rct::_update_ldelay" function.

Listing 5.12 Diagnostic output for the "Rct::_update_ldelay" loops showing the parallelism assigned by the compiler.

```
ot::Rct::_update_ldelay(ot::RctNode *, ot::RctNode *):
     193, Generating copyin(from_delay[:]) [if not already present]
          Generating copyout(from_ldelay[:]) [if not already present]
          Generating copyin(to_float_star[:num_nodes*4],from_cap[:],
    from_float_star[:num_nodes*4]) [if not already present]
          Generating NVIDIA GPU code
          198, #pragma acc loop gang /* blockIdx.x */
          200, #pragma acc loop vector(128) /* threadIdx.x */
               Generating implicit reduction(+:acc)
     200, Loop is parallelizable
```

To build the "NetAccelerated" static library on any machine, is necessary to change in the file the paths for the variables "OPENACC_PATH", "OPENACC_INC", and "OPENACC_LIB" with the ones for the configured HPC SDK, along with the path of the NVC compiler with the one in use. Then, in the NetAccelerated folder, execute the commands "make" and then "cmake ../" to build the static library.

To link the library containing the code to be offloaded to OpenTimer during compilation, we needed to slight modify the project CMakeLists file. Firstly, we excluded from the list of files in the project to be compiled the one we accelerated, "net.cpp". Next, we added the CMake variables contained in Listing 5.13 to the file. This CMake code allows us to save the path of the static library in a variable that will be added to the project using the "add_library" function. Finally, we linked the library to the OpenTimer executable using the "target_link_libraries" function.

Listing 5.13 Environment variables required in the CMakeLists file of OpenTimer to use the accelerated static library.

```
set(ACCELERATED_NET ${PROJECT_SOURCE_DIR}/NetAccelerated/lib/
    libNetAccelerated.a)
add_library(OpenTimer ${OT_CPP})
target_link_libraries(OpenTimer PUBLIC ${ACCELERATED_NET})
```

The complete version of OpenTimer GPU-accelerated can be found in [10].

Before building the project, simply update the paths for the "CUDA_PATH" and "NVC_PATH" in the CMakeLists file to match the locations of the CUDA and compiler libraries on the used machine. These libraries are required during compilation.

This compilation methods allows the flexibility to use any compiler for the section that will be run on the CPU. However, we have chosen the standard version of GCC compiler.

The required flags for the GCC compiler have already been added to the CMakeLists file, but they can be customized to use a different one. Then, when everything is ready, navigate to the project directory in a terminal shell and run "make" followed by "cmake ../" to build the entire project.

5.7 Execution Times Evaluation for the Various Approaches

We now aim to evaluate the resulting execution times for the various OpenTimer implementations. We compared the standard CPU-based version with our accelerated version that computes the load delay of the RC tree on the GPU. Then, we evaluated the the two compilation approaches we introduced before, which enable GPU offloading for the function.

To automate the analysis process, we relied on OpenTimer C++ APIs since the standard shell requires users to manually enter commands into the shell. We developed three C++ applications using different OpenTimer versions as baseline to perform the measurements:

1. A standard version that only runs on the CPU;
2. A version where the code to be offload has been compiled with NVC++ and the remaining CPU code using the standard implementation of GCC;
3. A version where we compiled the entire project using GCC with NVPTX extension to enable acceleration.

Each application takes the name of a benchmark, which represents a digital circuit, as input. Then it performs the "report_timing" action after loading the early and late libraries, the Verilog design, the parasitics, and constraints files for the module. We embedded a timer in the application using the C++ "std::chrono::steady_clock" function to collect the execution time for each run.

To measure the execution times, we selected 26 benchmarks included in OpenTimer, each representing a different circuit design with varying component numbers. The measurements has been repeated ten times for each benchmark using the three versions. We then calculated the average values for these repetitions, which we present in Table 5.6.

The collected execution times demonstrate that performing a Depth-First Search (DFS) traversal on a graph is not an efficient operation to execute on the GPU, despite the precautions taken during the development phase. This is due to the high the number of nodes in the graph, each of which requires a data movement to the device, resulting in a bottleneck. On average, the CPU execution time takes 36.28% of the time required by the accelerated code using the NVC compiler in conjunction with the standard GCC compiler and the 21.80% of the code compiled using GCC with NVPTX extension.

Furthermore, we evaluated also the execution times for a CPU-version of OpenTimer that implements batching and compared it to the standard version. We found that it takes

4.98% more time to execute, even if our case is disadvantageous since the high number of nodes. This demonstrates that batching, which typically requires the allocation of new data structures and filling them, is not an onerous operation and that the time required can easily be recovered by executing the function on the GPU, if beneficial.

Finally, to compare the effectiveness of the two methods of compiling the code to be accelerated, we utilized version 11.3.0 of GCC, 11.8 of NVIDIA CUDA, and 22.9 of NVIDIA NVC compiler for both of the implementations. Our analysis, presented in Table 5.6, revealed that out of 26 benchmarks, 23 favored compiling the code with OpenACC directives using NVC to create a static library, followed by compiling the remaining CPU code using a standard version of GCC. This approach yielded an average execution time that was 63% less compared to using the GCC compiler with NVPTX support. However, this method requires a more invasive compilation process, including making changes to the CMakeLists file, compared to using GCC with NVPTX support, which enables OpenACC directive to work on GPU almost out-of-the-box.

Table 5.6: OpenTimer execution times for CPU standard version and for GPU implementations, using the two different compilation methods to accelerate the code.

Benchmark	CPU Execution [ms]	NVC + GCC [ms]	GCC with NVPTX ext. [ms]	Faster Accelerated Solution
c3_slack	44.604	46.534	45.821	GCC with NVPTX ext.
c17	45.511	81.342	111.156	NVC + GCC
s27	46.644	99.724	208.550	NVC + GCC
s400	49.293	293.165	437.650	NVC + GCC
s526	51.779	382.150	478.524	NVC + GCC
simple	52.055	104.909	142.085	NVC + GCC
s386	59.115	328.495	484.028	NVC + GCC
s349	64.661	270.970	466.666	NVC + GCC
s344	66.150	273.805	439.586	NVC + GCC
s1494	78.094	947.867	849.674	GCC with NVPTX ext.
s510	78.290	451.695	602.622	NVC + GCC
s1196	86.533	815.462	810.046	GCC with NVPTX ext.
c432	93.353	137.320	472.217	NVC + GCC
c1908	109.050	252.002	467.406	NVC + GCC
c499	123.803	230.869	522.193	NVC + GCC
c1355	129.847	221.181	466.621	NVC + GCC
c3540	136.548	394.631	867.011	NVC + GCC
c6288	148.883	530.673	1152.849	NVC + GCC
c880	160.005	268.552	601.946	NVC + GCC
c2670	320.963	501.975	854.164	NVC + GCC
c5315	429.512	763.909	1350.638	NVC + GCC
c7552	450.275	803.307	1520.935	NVC + GCC
ac97_ctrl	459.577	6072.552	10779.579	NVC + GCC
aes_core	1030.752	7542.972	10597.931	NVC + GCC
des_perf	3412.909	24867.922	48808.327	NVC + GCC
vga_lcd	4415.907	79179.312	115491.217	NVC + GCC

Chapter 6

Results

The study of Arithmetic Intensity is a first crucial parameter that can help identify functions that are suitable for offloading on a GPU, even without prior knowledge of the code base. To automate the data collection process and avoid manual computations, we explored three different methods that do not require a GPU to be present on the machine. Each of these methods showed its own set of benefits and drawbacks, which we evaluated by comparing their results to a manual evaluation of each benchmark in the KernelGen Test suite.

The first method we tested involved using SDE to compute the floating-point operations (FLOPs), together with Intel VTune to evaluate the memory accesses of a function. Both tools required the code of interest to be marked manually to not measure the entire execution, and they were unable to differentiate between multiple sections for evaluation. Moreover, utilizing VTune requires an Intel CPU and the installation of Intel proprietary drivers, which require root access to the environment where the analysis is performed.

We found that SDE was both fast and precise in evaluating the FLOPs performed by every function. However VTune, in our testing of 12 benchmarks in the KernelGen Test Suite with small input dimensions, was unable to measure the memory accesses for 8 of them because the function execution speed was faster than the tool sampling interval of 0.1 ms. For the other input dimensions we tested it was able to evaluate the memory accesses without issue.

Unfortunately, the memory accesses evaluated by VTune are not always indicative of the data movement required for offloading computations to the GPU. To better explain why, we can consider the example of performing computation on a matrix data structure, which is often used in the KernelGen Test Suite. If we want to offload this computation to the GPU, we expect the data movement from the host to the device to be exactly the size of the matrix in bytes. However, VTune memory counters used to evaluate memory movements are based on the Integrated Memory Controller (IMC) counters, positioned as shown in Figure 6.1 between the main memory and the L3 cache. As a result, when the values are moved to the CPU core for computation, the total number of bytes accessed can be higher than the size of the matrix itself.

This is because of the cache miss phenomenon. When the CPU performs the computation, it stores some chunks of data in the cache, which is faster than reading directly from

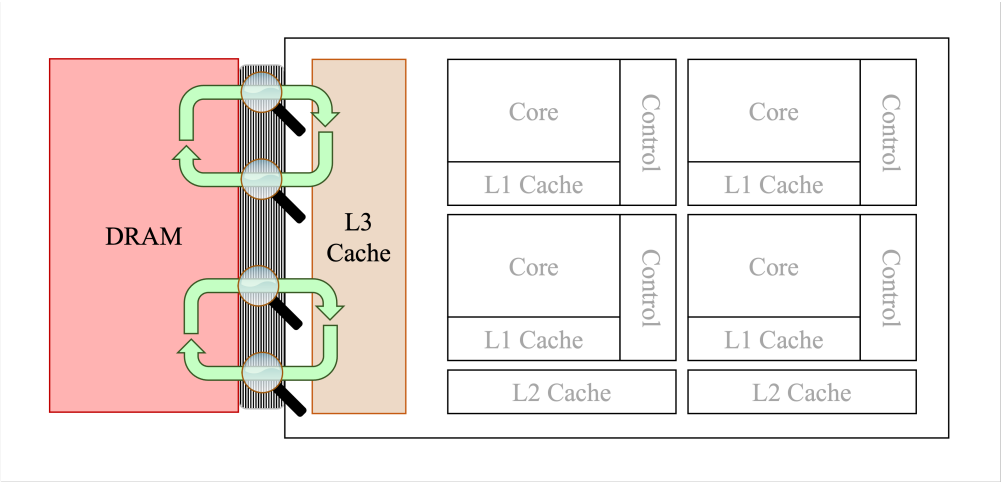


Figure 6.1: The location of the integrated memory controller counters, indicated by a magnifying glass, used for the memory analysis with VTune and LIKWID.

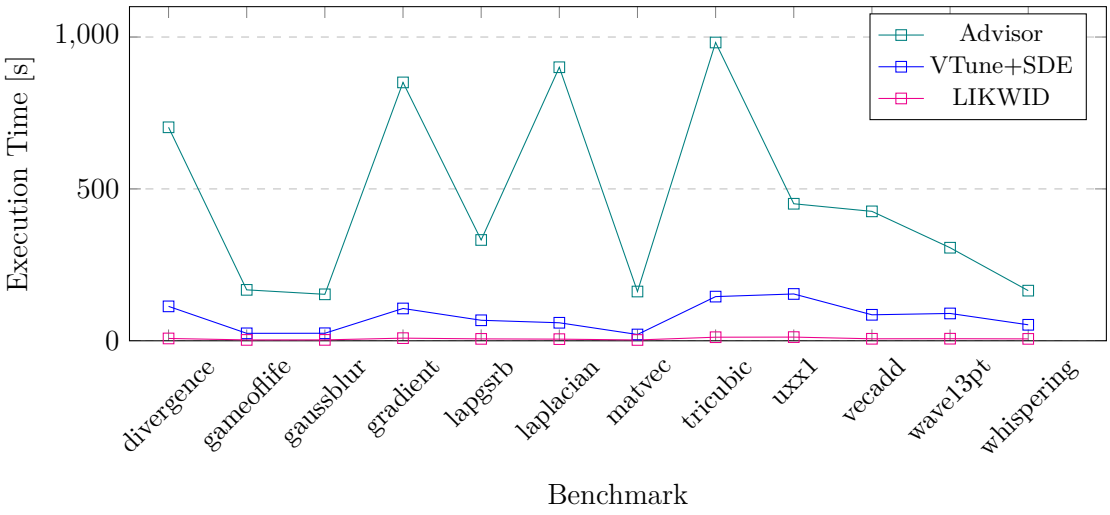
the DRAM. However, if the matrix is larger than the size of the cache or if the CPU can't predict correctly which data it will need next, a cache miss occurs, and the CPU must read another chunk of data from the DRAM. Furthermore, since the matrix elements are not always accessed in a contiguous manner, it can lead to frequent cache misses, as the cache is designed to work efficiently with contiguous blocks of memory. All these factors can cause the number of memory reads from the DRAM to be higher than the size of the matrix itself. This evaluation method is therefore poorly suited for accurately evaluating Arithmetic Intensity in the context of GPU offloading.

Our study included a second method based on RRZE LIKWID. This tool also requires the code to be instrumented to limit the amount of data collected, however it is possible to use multiple markers in the code, each with its identifier, to perform different measurements simultaneously. LIKWID is lightweight and open source tool, and its analysis adds almost no overhead to the execution time. Even it has been shown to be highly accurate in estimating the number of FLOPs performed by a section of code, it evaluates memory accesses in the same way as Intel VTune, unfortunately not oriented to GPU offloading. Furthermore, results obtained using this tool indicate that multiple repetitions of the memory accesses evaluation on the same function have a higher variance compared to other methods used.

The third and last method we tested to evaluate the Arithmetic Intensity of a function uses Intel Advisor. Unlike the other tools we introduced before that profile applications, it performs a modeling analysis which has been shown to be accurate in measuring FLOPs and evaluating memory accesses to obtain AI. In particular, Advisor is designed to model memory accesses oriented towards data offloading to GPU, taking into consideration also data reuse between accelerated loops. Using Advisor does not require special permissions on the machine used, and the variance between repeated measures was always zero.

Advisor analysis is capable of automatically detecting functions that could benefit

Figure 6.2: Comparison of the execution time for Arithmetic Intensity analysis across the different tools for the KernelGen benchmarks for the largest input dimension tested.



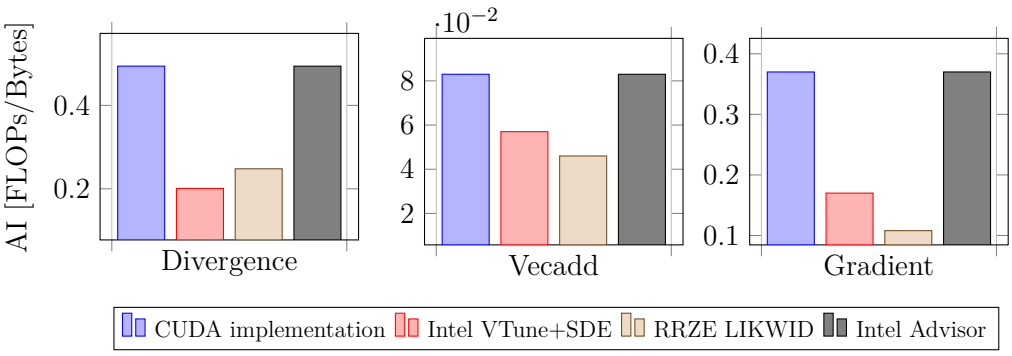
from offloading, which means that there is no need to use markers. However, its analysis is much more time-consuming, especially when modeling the data transfer with high precision. In Figure 6.2, we provide a comparison of the analysis required time for the three evaluation methods, showing only the results for the largest input dimension with the tested benchmarks. On average, VTune required 36% of the time required by Advisor, while LIKIWID required only 5%.

To validate our analysis, we compared the obtained results with the Arithmetic Intensity calculated through the CUDA implementation of the benchmarks executed on the GPU. In Figure 6.3, we present the results of the AI evaluation for three representative benchmarks, using the CUDA implementation as a golden reference. Overall, the comparison matched our expectations and validated our findings. Notably, Advisor emerged as the most suitable method oriented towards GPU offloading for assessing the AI of functions. The memory accesses modelled by the tool could also help developer identifying opportunities to optimize data movement between the host and device, even after the GPU implementation has already been written, thus addressing any potential performance issues.

While arithmetic intensity analysis is a useful metric to identify GPU offload opportunities, it cannot be the only parameter considered, as it does not account for dependencies between computations, memory bandwidth limitations, or conditional statements. To improve our analysis we modeled the speed up on GPU versus the CPU counterpart, using Intel Advisor Offload Modeling analysis and the Static Time Analysis (STA) tool OpenTimer as a test bench.

To evaluate the potential advantage, the tool compares the execution time of each function on the machine being used with the estimated time on a modeled GPU. This

Figure 6.3: Three example results of AI evaluation obtained using the tools compared with CUDA implementation of the Kernelgen Test Suite and input dimensions *512 512 512*



comparison is limited to GPUs produced by Intel, although configuration tuning is possible. If the speedup achieved is greater than 1, then the offloading is recommended. However, the results obtained are specific to the hardware configurations used in the evaluation, and may not be universally applicable.

The top five suggestions of functions that can be accelerated on GPU, automatically identified by Advisor on a design for which it was required to report the critical paths, were found to be not correct. Two functions were part of a library external to the project, one involved multiple complex C++ data structures not worth to menage on GPU and the last two had a data dependency which makes not possible to parallelize the loop taking advantage of the acceleration.

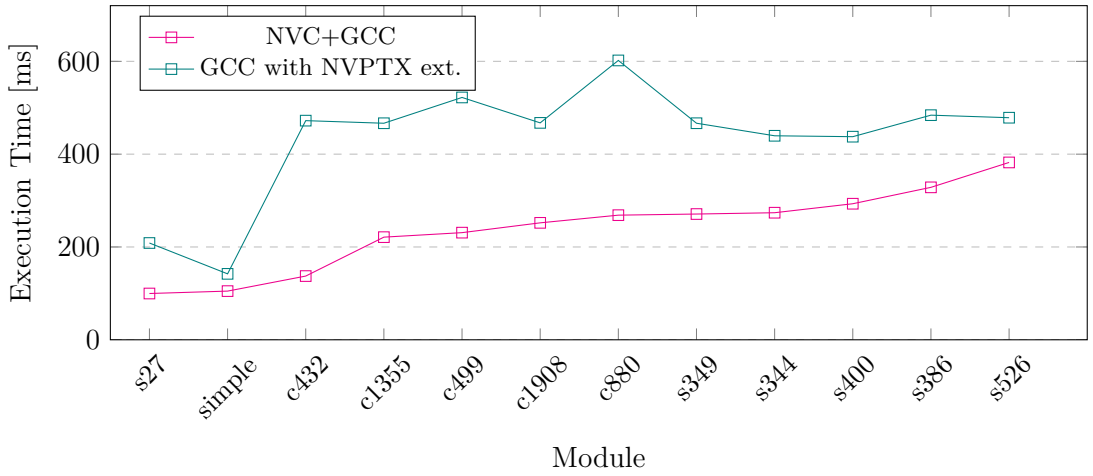
However, by sorting the analysis results by increasing AI, we were able to identify loops that could be accelerated. Among five with the highest AI values identified, four were part of the RC propagation phase in the circuit. We successfully ported the function with highest AI value from OpenTimer, which is part of the RC delay computation, to GPU with minimal code changes using OpenACC directives. Their use proved to be highly effective, enabling even non-expert developers to implement GPU code with ease.

The collected execution times demonstrated that performing a Depth-First Search (DFS) traversal on a graph, which is part of the offloaded function, is not an efficient operation to execute on the GPU due to the high number of data movements required. On average, the CPU version took 36.28% of the time required by the best solution of accelerated code. However we were able to improve the efficiency of data transfers to the GPU introducing the batching and vectorization techniques. Batching had a minimal impact on the execution time, making it a crucial resource for leveraging the full capabilities of the GPU. Moreover, the evaluation of AI of the loops during the developing phase, helped us to evaluate when it was more convenient to offload or to keep the execution on the CPU below a certain AI threshold.

Finally, we tested two methods for compiling the code to accelerate, and found that using the NVIDIA NVC compiler to create an accelerated static library, followed by compiling the remaining CPU code using a standard version of GCC, yielded better execution times on 23 out of 26 benchmarks, with an average execution time that was 63% less

compared to using the GCC compiler with NVPTX support. Figure 6.4 shows the results for a sample of the tested modules. However, creating the static library required a more invasive compilation approach, including changes to the CMakeLists file, compared to compile the entire project with GCC with NVPTX support, which enables OpenACC directive to work on GPU almost out-of-the-box.

Figure 6.4: Comparison of the execution times for the two compilation methods to accelerate OpenTimer code.



Chapter 7

Conclusions

GPGPUs are a valid answer to the increasing demand for more computing power to effectively tackle tasks such as scientific simulation and machine learning. Their high level of parallelism and compute performance make it possible to process large amounts of data in parallel, which is something that traditional CPUs are not well-suited for. However, writing code that runs on GPUs can be challenging for several reasons, including that GPU vendors have different programming model and libraries, which can take time to learn, and it is not easy to identify functions that are worth offloading without prior knowledge or assumption, especially in large code-bases.

One way to assess GPU offload opportunities is to study the Arithmetic Intensity (AI) value of functions, and being able to automatically compute it can help developers expedite the evaluation process. We studied three methods for automatically evaluating AI values, including using respectively Intel VTune with SDE, RRZE LIKWID, and Intel Advisor. Among these methods, Intel Advisor stood out for its ability to provide an evaluation of AI that was specifically oriented towards data offloading.

Given the limits on the AI evaluation as a standalone metric for GPU offload potential, we improved our analysis including the modeling of the speed up on GPU versus the CPU counterpart, providing further hints thanks to the Intel Advisor *Offload Modeling* feature, using as end-to-end approach evaluation the Static Time Analysis (STA) tool OpenTimer, chosen because it operates in a similar area as the Synopsys team where the internship was carried out.

The results showed that Advisor modeling analysis had difficulties analyzing data dependencies or complex data structures in the functions it suggested for offloading. However, by sorting the analysis results by AI, we were able to select and port the function with the highest measured value to GPU with minimal code changes using OpenACC directives.

Between the top five functions with highest AI, estimated by Intel Advisor in OpenTimer, four were part of the RC delay computation stage. A study by Guo et al. [13] showed that this stage, together with levelization and timing propagation tasks, had the highest offload opportunities, with the potential to bring up to 3.69x speedup on the execution time of OpenTimer. However computing a timing graph, which is a fundamental operation in STA, is an extremely challenging task to accelerate since it involves irregular

memory access, dynamic data structures, and recursion. Therefore, taking advantage of GPU parallelism in their study required the development of GPU-efficient data structures and substantial algorithmic changes.

This shows that AI is an important first step in identifying functions that could benefit from GPU parallelism. However, AI alone is not enough information to completely estimate possible advantages of GPU execution, because the evaluation of the function is based on the CPU implementation, which may not have considered the possibility of offloading or developing techniques to increase performance, such as batching or vectorization, to fully realize the benefits of GPU execution.

By using OpenACC directives, we were able to write code that can be executed on the GPU, regardless of the brand, with minimal modifications on the source code. We illustrated two ways to compile the project with accelerated code, both of which had a low impact on the project build chain. This makes it easy for developers to port their applications to the GPU, even for those who are new to this field.

This was also confirmed in the "Application Experiences on a GPU-Accelerated Arm-based HPC Testbed" research [8], where ten teams from several universities collaborated on a project to port and benchmark various High Performance Computing (HPC) applications to GPU in Arm-based HPC platforms, also using OpenACC directives. The results showed that the porting process was straightforward for every application, requiring only minor modifications to the build system to run the applications on the GPU.

The technologies are proved to be mature and reliable, and future advancements such as a more tightly integrated memory between CPU and GPU will further enhance their advantages, overcoming the data transfer bottleneck.

Bibliography

[1] AMD. «Introducing DNA Architecture». In: *AMD Docs* (2019). URL: https://www.amd.com/system/files/documents/rdna-whitepaper.pdf.

[2] Cedric Andreolli, Zakhar Matveev, and Vladimir Tsymbal. «Modeling Heterogeneous Computing Performance with Offload Advisor». In: *Proceedings of the International Workshop on OpenCL*. ACM, Apr. 2020. DOI: 10.1145/3388333.3388665.

[3] Toru Baji. «Evolution of the GPU Device widely used in AI and Massive Parallel Processing». In: *IEEE 2nd Electron Devices Technology and Manufacturing Conference (EDTM)* (2018). DOI: 10.1109/edtm.2018.8421507.

[4] J Bhasker and Rakesh Chadha. *Static timing analysis for nanometer designs*. 2009th ed. Springer, Apr. 2009. ISBN: 9780387938202.

[5] Matthias Christen, Olaf Schenk, and Yifeng Cui. «Patus for convenient high-performance stencils: Evaluation in earthquake simulations». In: *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2012. DOI: 10.1109/sc.2012.95.

[6] Matthias M. Christen. *PATUS Quickstart*. 2012. URL: https://github.com/matthias-christen/patus/blob/master/doc/quickstart/quickstart.pdf.

[7] Lauro B. Costa, Samer Al-Kiswany, and Matei Ripeanu. «GPU support for batch oriented workloads». In: *2009 IEEE 28th International Performance Computing and Communications Conference*. IEEE, Dec. 2009. DOI: 10.1109/pccc.2009.5403809.

[8] Wael Elwasif et al. *Application Experiences on a GPU-Accelerated Arm-based HPC Testbed*. 2022. DOI: 10.48550/ARXIV.2209.09731.

[9] Toru Fujita et al. «Efficient GPU Implementations for the Conway's Game of Life». In: *2015 Third International Symposium on Computing and Networking (CANDAR)*. IEEE, Dec. 2015. DOI: 10.1109/candar.2015.11.

[10] Jacopo Pati GitHub. *Case Study: OpenTimer GPU Accelerated Using OpenACC directives*. Mar. 2023. URL: https://github.com/include-jacopo/OpenTimer-CaseStudy.

[11] Jacopo Pati GitHub. *CMakeLists to Create a Static Library for OpenTimer*. Mar. 2023. URL: https://github.com/include-jacopo/OpenTimer-CaseStudy/blob/main/NetAccelerated/CMakeLists.txt.

[12] Thomas Gruber. *LIKWID Perfctr Wiki*. Nov. 2022. URL: https://github.com/RRZE-HPC/likwid/wiki/likwid-perfctr.

[13] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. «GPU-accelerated static timing analysis». In: *Proceedings of the 39th International Conference on Computer-Aided Design*. ACM, Nov. 2020. DOI: 10.1145/3400302.3415631.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055.

[15] Tsung-Wei Huang and Martin D. F. Wong. «OpenTimer: A high-performance timing analysis tool». In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2015, pp. 895–902. DOI: 10.1109/ICCAD.2015.7372666.

[16] Tsung-Wei Huang et al. «OpenTimer v2: A New Parallel Incremental Timing Analysis Engine». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.4 (Apr. 2021), pp. 776–789. DOI: 10.1109/tcad.2020.3007319.

[17] Intel. *Calculating "FLOP" using Intel Software Development Emulator (Intel SDE)*. Mar. 2015. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/calculating-flop-using-intel-software-development-emulator-intel-sde.html.

[18] Intel. *Intel Advisor*. Mar. 2023. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html.

[19] Intel. «Intel Advisor User Guide». In: *Intel Documentation* (Feb. 2023). URL: https://www.intel.com/content/dam/develop/external/us/en/documents/advisor-user-guide.pdf.

[20] Intel. *Intel Software Development Emulator (Intel SDE)*. Oct. 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/tool/software-development-emulator.html.

[21] Intel. *Intel VTune Profiler*. Feb. 2023. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[22] Intel. «Intel VTune Profiler User Guide». In: *Intel Documentation* (Feb. 2023). URL: https://www.intel.com/content/dam/develop/external/us/en/documents/vtune-profiler-user-guide.pdf.

[23] Intel. *Intel VTune Supported Architectures and Terminology*. Feb. 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/system-requirements/vtune-profiler-system-requirements.html.

[24] Intel. *ITT APIs Basic Usage and Configuration*. Dec. 2022. URL: https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top/api-support/instrumentation-and-tracing-technology-apis/basic-usage-and-configuration/configuring-your-build-system.html.

[25] Intel. *ITT APIs Repository*. Feb. 2023. URL: https://github.com/intel/ittapi.

[26] Intel. *Monitoring Integrated Memory Controller Requests in Intel Core processors*. Feb. 2016. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/monitoring-integrated-memory-controller-requests-in-the-2nd-3rd-and-4th-generation-intel.html.

[27] F. Lekien and J. Marsden. «Tricubic interpolation in three dimensions». In: *International Journal for Numerical Methods in Engineering* 63.3 (2005), pp. 455–471. DOI: 10.1002/nme.1296.

[28] Antonino Calà Lesina et al. «FDTD Method and HPC for Large-Scale Computational Nanophotonics». In: *NATO Science for Peace and Security Series B: Physics and Biophysics*. Springer Netherlands, 2017, pp. 435–439. DOI: 10.1007/978-94-024-0850-8_25.

[29] Dmitry Mikushin. *KernelGen Stencil Performance Test Suite for CPU and GPU compilers*. Feb. 2022. URL: https://github.com/dmikushin/kernelgen-perf-tests.

[30] Dmitry Mikushin et al. «KernelGen – The Design and Implementation of a Next Generation Compiler Platform for Accelerating Numerical Models on GPUs». In: *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE Computer Society, 2014, pp. 1011–1020. DOI: 10.1109/IPDPSW.2014.115.

[31] John Nickolls and David Kirk. *Appendix C: Graphics and Computing GPUs*. 5th. Morgan Kaufmann Publishers Inc., 2013. ISBN: 9780124077263.

[32] NVIDIA. *Achived FLOPs*. Mar. 2023. URL: https://docs.nvidia.com/nsight-visual-studio-edition/4.6/Content/Analysis/Report/CudaExperiments/KernelLevel/AchievedFlops.htm.

[33] NVIDIA. *CUDA Toolkit*. Mar. 2023. URL: https://developer.nvidia.com/cuda-toolkit.

[34] NVIDIA. «GPU Performance Background». In: *NVIDIA Docs* (2022). URL: https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html.

[35] NVIDIA. «NVIDIA A100 Tensor Core GPU Architecture». In: *NVIDIA Docs* (2020). URL: https://nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[36] NVIDIA. *NVIDIA Nsight Compute*. Mar. 2023. URL: https://developer.nvidia.com/nsight-compute.

[37] NVIDIA. *NVIDIA Nsight Systems*. Mar. 2023. URL: https://developer.nvidia.com/nsight-systems.

[38] NVIDIA. *NVIDIA Visual Profiler*. Mar. 2023. URL: https://developer.nvidia.com/nvidia-visual-profiler.

[39] Georg Ofenbeck et al. «Applying the Roofline Model». In: IEEE, 2014. DOI: 10.1109/ispass.2014.6844463.

[40] OpenACC. *OpenTimer. More science, Less Programming.* Mar. 2023. URL: https://www.openacc.org.

[41] openacc-standard.org. «OpenACC Programming and Best Practices Guide». In: *OpenACC Documentation* (May 2021). URL: https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0_0.pdf.

[42] J.D. Owens et al. «GPU Computing». In: *Proceedings of the IEEE* 96.5 (2008). DOI: 10.1109/JPROC.2008.917757.

[43] GitHub Repository. *OpenTimer - High-Performance Timing Analysis Tool.* Mar. 2023. URL: https://github.com/OpenTimer/OpenTimer.

[44] RRZE. *LIKWID "MEM_SP" Performance Group for Skylake Architecture.* Apr. 2021. URL: https://github.com/RRZE-HPC/likwid/blob/master/groups/skylake/MEM_SP.txt.

[45] RRZE. *LIKWID Repository.* Feb. 2023. URL: https://github.com/RRZE-HPC/likwid.

[46] Tolga Soyata. *GPU Parallel Program Development Using CUDA*. 1st. Chapman and Hall/CRC, 2018. DOI: 10.1201/9781315368290.

[47] J. Treibig, G. Hager, and G. Wellein. «LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments». In: *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures* (2010). DOI: 10.1109/ICPPW.2010.38.

[48] NVIDIA Website. *NVIDIA HPC SDK.* Mar. 2023. URL: https://developer.nvidia.com/hpc-sdk.

[49] NVIDIA Website. *NVIDIA Parallel Thread Execution.* Mar. 2023. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[50] Spack Website. *Spack - A Flexible Package Manager.* Mar. 2023. URL: https://spack.io.

[51] Charlene Yang. «Hierarchical Roofline Analysis: How to Collect Data using Performance Tools on Intel CPUs and NVIDIA GPUs». In: *CoRR* (Oct. 2020). DOI: 10.48550/ARXIV.2009.02449.