TÉCNICO LISBOA

Roofline Analysis and Performance Characterization for Intel Integrated GPUs

Afonso Rodrigues de Carvalho

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Doctor Aleksandar Ilić
Doctor Leonel Augusto Pires Seabra de Sousa

Examination Committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Doctor Aleksandar Ilić
Member of the Committee: Doctor Mauricio Breternitz

September 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to thank Professor Aleksandar Ilić and Professor Leonel Sousa for accepting me and trusting me with this work and for all the guidance provided in our meetings. A big thank you to Rafael Campos, for the invaluable support provided throughout the Thesis, and for his patience in helping me overcome my struggles. Finally, thanks to my family, friends, and girlfriend for supporting me throughout this whole journey

Abstract

The constant need to support complex software applications has driven companies to enhance computers' processing speed and overall performance by developing new computer architectures. Several modern computers possess an Integrated Graphics Processing Unit (iGPU) inside the same chip as its Central Processing Unit (CPU). Not only it handles graphics functions, but it also executes any required general-purpose computation. It should relieve the CPU of some of its workload and process it more efficiently than its counterpart. To correctly predict the behavior of the iGPU, its state-of-the-art microarchitecture is thoroughly analyzed. This Thesis proposes to enhance characterization methodology to uncover the performance upper bounds of the iGPU through architecture microbenchmarking. It examines data related to throughput, memory bandwidth, power consumption, and the study of predictive models such as the Original Roofline Model or the Cache-Aware Roofline Model when applied to an iGPU. This Thesis uses the OpenCL programming model to microbenchmark the hardware architecture of an Intel Gen9.5 iGPU, characterizing its performance with the aid of the Roofline Model and other additional tools.

Keywords

Integrated GPU, Throughput, Memory Bandwidth, Power Consumption, Energy Efficiency, Roofline Model

Resumo

Com o recente desenvolvimento de computadores, a constante necessidade de aumentar a sua velocidade de processamento e de melhorar o seu desempenho geral tem levado as empresas a desenvolver e melhorar as arquiteturas dos computadores. Vários computadores modernos possuem uma Unidade de Processamento de Gráficos integrada (iGPU) dentro do mesmo *chip* que a sua Unidade Central de Processamento (CPU). O dever da GPU é não só a execução de funções gráficas, mas também a realização de qualquer cálculo de uso geral. Esta deve aliviar a CPU de algumas das suas cargas de trabalho e executá-las de forma mais eficiente do que a mesma. Para prever corretamente o comportamento da iGPU, a arquitetura do estado da arte é analisada minuciosamente. Esta Dissertação propõe-se a caracterizar o desempenho da iGPU através de vários critérios de ensaio relacionados com a computação, largura de banda de memória, consumo de energia e estudo de modelos preditivos como o Modelo Roofline Original ou o Modelo Roofline Cache-Aware quando aplicados a uma iGPU. Faz também uso do modelo de programação OpenCL de forma a medir e especificar a arquitetura de *hardware* de uma iGPU da Gen9.5 da Intel e caracteriza o seu desempenho com a ajuda do Modelo Roofline e algumas ferramentas adicionais.

Palavras Chave

GPU Integrada, Computação, Largura de Banda de Memória, Consumo de Potência, Eficiência Energética, Modelo Roofline

Contents

List of Figures

List of Tables

Listings

Acronyms

AI Arithmetic Intensity. 14, 16, 20, 21

API Application Programming Interface. 13, 18

ARF Architectural Register File. 11

CARM Cache-Aware Roofline Model. 2, 3, 14–17, 19–21, 32, 35, 37–44, 54, 55, 57

CLIP Clipper. 8

CPU Central Processing Unit. 2, 4, 6, 7, 9, 10, 13–22, 24, 26–28, 35, 46, 57

CS Command Streamer. 7, 8

DRAM Dynamic Random-Access Memory. 2, 6, 11, 15, 16, 19, 20, 24, 27, 28, 31, 32, 35, 46, 49, 50, 54, 57

DS Domain Shader. 8

EU Execution Unit. 9–13, 18, 30, 32, 35–38, 41, 46

FDTD Finite Difference Time Domain. 17

FLOP Floating Point Operation. 12, 14, 17, 20, 28

FLOPS Floating Point Operations per Second. 15, 20, 28, 36–39, 41, 42, 44, 45, 53, 54, 57

FPU Floating Point Unit. 12, 13, 31, 35, 36, 38, 39, 41, 42

GPGPU General Purpose Graphics Processing Unit. 2–4, 6, 8, 9, 11, 13, 22, 24, 57

GPU Graphics Processing Unit. 2–4, 6, 7, 12–14, 17–22, 24, 26–28, 33, 37, 45, 46, 54, 57

GRF General Register File. 11, 46, 54

GTD Global Thread Dispatcher. 8, 11

GTI Graphics Technology Interface. 7, 9, 49

HS Hull Shader. 8

I/O Input/Output. 6

IF Instruction Fetch. 11

iGPU Integrated Graphics Processing Unit. 2–4, 6–13, 17–19, 22, 24, 26–33, 35–39, 42, 44, 46, 49, 50, 53–55, 57

IMT Interleaved Multi-Threading. 12

INTOPS Integer Operations per Second. 28, 40

LLC Last Level Cache. 6, 7, 15, 18, 19, 35, 46, 49, 50, 54, 57

LTD Local Thread Dispatcher. 10, 11

MAD Multiply and Add. 12, 16, 18, 19, 26, 29, 30, 36, 37, 41, 42, 44, 45, 51, 53, 54, 57

OI Operational Intensity. 14, 15

OpenCL Open Computing Language. 2–4, 6, 13, 19, 22, 24–27, 29, 30, 33, 37, 38, 46, 50, 54, 57

ORM Original Roofline Model. 2, 14–16, 57

PAPI Performance Application Programming Interface. 24, 28, 33, 51

PCIe Peripheral Component Interconnect Express. 6, 27

RAPL Running Average Power Limit. 20, 24, 28, 51

S/F Strip/Fan. 8

SIMD Single Instruction Multiple Data. 11, 12, 36

SLM Shared Local Memory. 9–11, 13, 19, 35, 46

SMT Simultaneous Multi-Threading. 12, 13

SoC System-on-Chip. 6, 7, 27

SOL Stream Output Logic. 8

TA Thread Arbiter. 38

TE Tesselation Engine. 8

TS Thread Spawner. 8

URB Unified Return Buffer. 8

VF Vertex Fetch. 8

VFE Video Front End. 8

VS Vertex Shader. 8

W/M Windower/Masker. 8

1

Introduction

Contents

In the last decades, computer architectures have had significant developments due to emerging technologies. These were not exclusive to the sheer quantity of cores and processing power: out-of-order execution, instruction parallelism, cache hierarchies in the memory system, were all key improvements in the thriving field of computer architecture. The design of computer processors evolved from in-order single cores to superscalar execution, compacting multiple cores in a single chip, and more recently augmenting them through heterogeneous computation [1], with the aid of Graphics Processing Units (GPUs). A GPU is a specialized device in techniques that allow fast integer and/or floating-point computation to the detriment of control instructions and speculative execution. Due to its strengths, uses for it quickly started appearing in mobile phones and embedded systems for scientific, medical, or engineering purposes [2]. In modern times, Central Processing Units (CPUs) and GPUs are both used to accomplish the computation needs of the users efficiently. The introduction of GPUs for general-purpose computing tasks started gradually, as the devices were initially designed with the sole purpose of accelerating the computation of graphic workloads. Later, however, they evolved to incorporate relatively simple general-purpose cores, capable of processing a significant amount of instructions in short periods.

The increasing use of GPUs for scientific applications led to the popularization of General Purpose Graphics Processing Units (GPGPUs). For this purpose, programming models and frameworks such as Open Computing Language (OpenCL) [3] [4] were developed to allow for the execution of code in multiple devices, such as CPUs, GPU, and later accelerators. Integrated Graphics Processing Units (iGPUs) were introduced in a fraction of client line processors, with the particularity of being placed in the same silicon die as the CPU [5]. Therefore, they are much smaller in size than their discrete counterparts, having size limitations and power consumption restrictions, resulting in fewer cores and, consequently, in less throughput. Nevertheless, iGPUs, such as the Intel Processor Graphics Gen 9.5, present in Intel Core processors, share part of the memory subsystem and the main Dynamic Random-Access Memory (DRAM) with the CPU cores, leading to advantages such as fewer costs to communication in any data transfers between the devices, enabling concepts such as heterogeneous computation and GPGPU [6].

A thorough investigation of these architectures is critical so that the applications achieve the maximum potential an iGPU has to offer in GPGPU. The identification of these characteristics is feasible via benchmarking tests fully exploiting the capability of highly parallel computation units and the memory access bandwidth and latency. As means to keep developing computer architectures, performance models and profiling methods that delineate the performance of applications are fundamental. In the scope of this work, insightful models such as the Original Roofline Model (ORM) [7] and Cache-Aware Roofline Model (CARM) [8] [9] are used to characterize the behavior and performance of applications in iGPUs. These are valuable tools to guide programmers through optimizations to their code and help to portray the nature of the applications' boundaries. Tools such as Intel Advisor [10] have integrated

CARM onto its software. Additionally, Intel Advisor also can enable a deeper glance into developed microbenchmarks and the underlying architecture for their execution, providing crucial data for optimization practices and analysis methods.

1.1 Motivation

As computer architectures evolved, so did the software applications that are applied to them, requiring more and more computing power, forcing hardware developers to improve their products with rapid developments. As the notion of heterogeneous computation and its applications are still considerably recent, it is still somewhat of an uncharted area, especially when considering iGPUs. As such, their usage in GPGPU requires deep knowledge of the underlying architecture of the hardware and how to fully exploit those capabilities.

The emergence of programming models such as OpenCL [3] [4] is a step in this direction, but since OpenCL is a general platform, employed by multiple vendors, it is difficult to squeeze all of the best aspects of the Intel iGPU architecture, due to not being tailored to it. The high-level GPU programming code may not translate to the intended instructions or certain optimizations can defeat the purpose of the benchmark. This event arises due to an unpredictability factor ingrained in the generated assembly code from OpenCL benchmarks that cannot be removed. There are developed tools such as Intel Advisor [10] and insightful performance models like Roofline Models [7] [8] [9], which are integrated into the former, providing helpful data related to kernel execution on the hardware and provide general optimization advice. However, more low-level implementation details, often related to the interactions between the programming model and the GPU device, are harder to comprehend without more extensive testing. Some questions remain unanswered, such as how many kernels running the same operation need to be launched to maximize its throughput; how to evenly split the workload between the computation units; how high-level OpenCL code translates into low-level GPU instructions; what are the best performing data types for each operation and their vectorization level... This Thesis, based on deep knowledge of the iGPU architecture, aims to touch on those topics and attempt to comprehend the intricacies of the presented hardware.

1.2 Objectives

The objectives for this Thesis are the following:

- Development of a set of microbenchmarks, to characterize the performance of the Intel Gen 9.5 iGPU microarchitecture, on several fronts, including measurements of computation capabilities,

bandwidth and latency of different levels of the memory hierarchy, power consumption, and energy efficiency;

- Employ the usage of performance models and external tools such as Roofline Models to optimize such benchmarks and to validate the obtained results in the context of the studied architecture.

Other goals will have to be met to achieve the above, such as a thorough review of the microarchitecture of the iGPU, in particular, the distinctions between discrete GPU, the communication with the CPU, the memory accesses, and its general-purpose computing abilities. Additionally, it is required to perform a characterization of the OpenCL programming model for GPUs, in its effectiveness when associated with Intel microarchitectures.

1.3 Main Contributions

The main contribution that this Thesis aims to achieve is an assessment of the main general-purpose computation capabilities of the Intel Gen 9.5 iGPU microarchitecture and in-depth performance characterization of its behavior, with relation to throughput, memory access bandwidth and latency, power consumption, and energy efficiency. The characterization of these metrics relies on programming and performance models, including a set of developed microbenchmarks, presented throughout this Thesis, aiming to provide a solid foundation to future works in the field that require knowledge and understanding of the reviewed iGPU hardware architecture and performance.

1.4 Outline

This Thesis is presented as follows: Chapter 2 refers to the microarchitecture of an Intel iGPU and its behavior and utilities, focused on the topic of GPGPU computing. Additionally, it describes the primary programming model used to communicate and program the device, as well as and the interactions between it and the presented hardware. The relevant state-of-the-art work to this Thesis is also depicted in this section. An introduction to the Roofline Model applied to the CPU and discrete GPU in various specifications is also shown. In Chapter 3, the makings of the set of microbenchmarks set are revealed, including the host code and the developed kernels. Any external tools and software used to complement it are also mentioned. Chapter 4 presents experimental results derived from the set of microbenchmarks applied to an experimental setup consisting of an Intel Gen 9.5 iGPU. The intents are to characterize its performance in various metrics and validate its performance through Roofline Models and other external tools. Lastly, Chapter 5 presents conclusions to the Thesis and speculates about conceivable future works to be employed in this area.

2

Background and State-of-the-Art: Intel iGPU and Performance Models

Contents

A GPU is a processing unit specialized in graphics applications. While CPU architectures often possess a small quantity of powerful out-of-order cores, GPU architectures are characterized by employing multiple cores and focusing on the execution of arithmetic and logic instructions in a large amount. Making use of its available cores, a GPU manages to achieve high computation throughput. As a drawback to their architecture, GPUs do not perform as well as CPUs when executing conditional statements and speculative execution. Nowadays, modern GPUs can be exploited for more than their graphics handling capability: the requirement of intensive calculations in applications that can benefit from GPUs' characteristics led to the development of GPGPU.

GPGPU is a technique that rests in performing general-purpose computation tasks into the GPU to optimize the time efficiency of the processes' execution. A noteworthy drawback is the data transfer related to sending workloads from the motherboard to the GPU through an interconnecting bus. The use of iGPUs reduces this effect since they are located on the same die as the CPU, sharing a part of the CPU memory subsystem and the main memory, facilitating data transfers between both. However, this comes with the drawback of having size and power consumption restrictions due to the iGPU location in the hardware architecture.

In this Chapter, the architecture of an Intel Gen9.5 iGPU and its functionalities are described, as it is the base for the performance analysis work performed in this Thesis. Moreover, OpenCL is introduced and explained, as it is the adopted framework to develop microbenchmarks on the iGPU architecture. Following that, proposed methods and tools to aid in the performance characterization of the iGPU are also mentioned. Lastly, state-of-the-art works related to the topic of this Thesis are presented.

2.1 Intel GPU Microarchitecture

Modern CPU System-on-Chips (SoCs) include several interconnected components, purposely designed to aid the CPU in numerous tasks. Currently, most SoCs include an iGPU, such as Intel Core client line processors. Figure 2.1 provides a diagram of an Intel Core processor. Intel Core client line processors typically include CPU cores, an iGPU, a shared, sliced Last Level Cache (LLC) and a memory and Input/Output (I/O) controller, designated by System Agent [5], however, the components may vary depending on the architecture. All of these components are considered unique agents. The components are all connected through the same bus, designated by Ring Interconnect. It is a 32-byte wide, bi-directional data bus possessing different channels for request, snoop, and acknowledge signals. The architecture promotes the scalability and extensibility of components on the same die by connecting them through the bus [5].

The System Agent includes a DRAM memory management unit, I/O off-chip controllers like the Peripheral Component Interconnect Express (PCIe) bus. It is relevant to note that this architecture

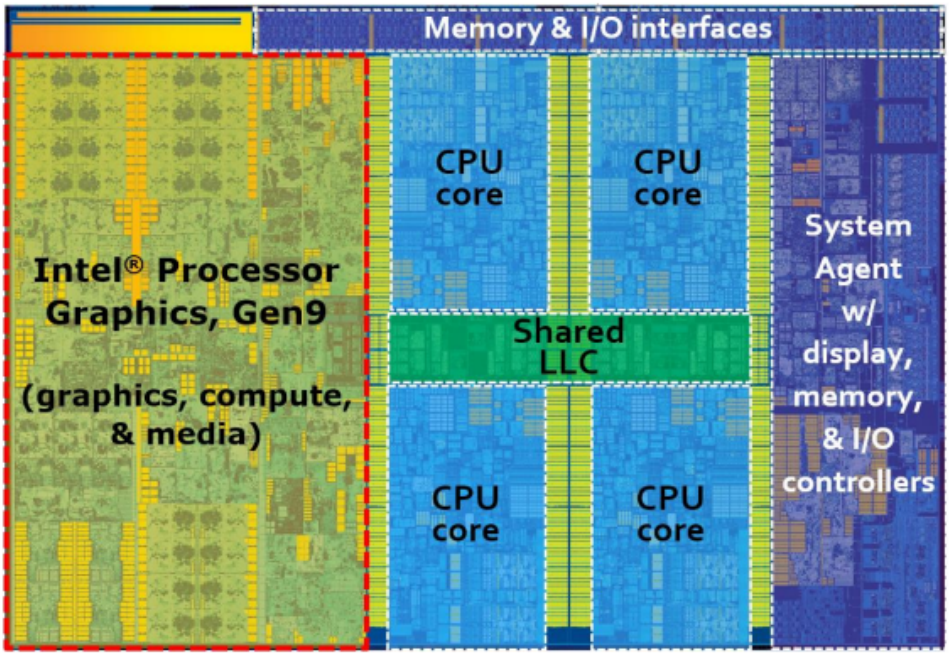

Figure 2.1: Intel Core i7 6700K SoC Architecture [5]

contains several unique clock domains, one per CPU core, an iGPU one, and one for Ring Interconnect. The LLC is shared between all CPU cores and iGPU. Each on-die core gets allocated one cache slice, although all cache slices still work as a single cache. An address hashing scheme maps the data to the correct cache slice [5].

The iGPU is divided into modules to aid the scalability of the architectures: Gen9.5 iGPUs have from one to three slice modules and one unslice module. Figure 2.2 presents an example of iGPU composition.

2.1.1 Unslice Architecture

The unslice, represented in Figure 2.2, has a different clock domain and power gating from the rest of the iGPU and can have a higher clock frequency than the slice. The unslice is responsible for fixed-function media capabilities such as Scalar Format Converter, Video Quality Engine, and Multi-Format Codex, and geometry fixed function pipeline for 3D workloads [5] [11].

The Graphics Technology Interface (GTI) acts as a bridge between the iGPU and the rest of the SoC, through the Ring Interconnect. It is located outside the fixed functions but still grouped in the unslice. The GTI implements atomic functions that can be accessed by the CPU or the GPU and power management controls for the latter. The GTI has a read/write bandwidth to the ring interconnect of 64 B/cycle [5].

The Command Streamer (CS) controls the flow of the execution of the fixed function pipeline and

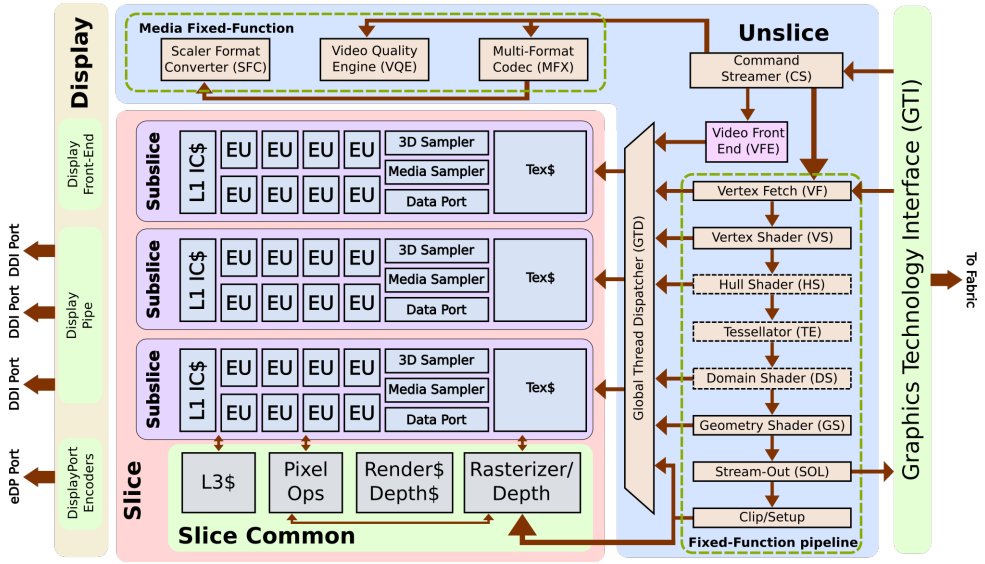

Figure 2.2: Intel Gen9.5 iGPU Architecture [11]

the media pipeline. It switches between pipelines and forwards command streams to any stage. The CS also allocates the Unified Return Buffer (URB) which is a globally shared and accessed buffer by the pipeline's blocks. The purpose of the latter is to allow transfers of data between threads or between threads and the 3D pipeline units, as means of message passing. The CS manages all GPGPU, 3D, and media pipelines [11].

The 3D pipeline is composed of multiple stages. To start, the Vertex Fetch (VF) is responsible for obtaining vertex data from memory and writing the data to the URB after reformatting. The Vertex Shader (VS) processes the received vertices by shading and transferring them to VS threads by contacting the Global Thread Dispatcher (GTD). The Hull Shader (HS) is responsible for the shading of the patch primitives as part of the tessellation process. Primitives are the shapes formed by the obtained vertices, such as triangles. The Tesselation Engine (TE) uses the data received from HS to tessellate the vertices into domain point topologies. The Domain Shader (DS) processes the domain points into vertices. These are sent through stream-out buffers to memory by the Stream Output Logic (SOL). In the Clipper (CLIP) stage, the objects that require slicing by the scene are cut. The Strip/Fan (S/F) stage sets up objects in the scene, and the Windower/Masker (W/M) rasterizes them, which consists of transforming the primitive shapes into pixels to be displayed on the screen [11].

The media and GPGPU functionalities share the pipeline. After the CS parses a command, the Video Front End (VFE) prepares threads for the Thread Spawner (TS) to dispatch them. The latter interacts with the GTD, which spawns the parent threads originated from VFE or child threads. The GTD is the bridge between the unslice and the slices when in need of any computation [11].

2.1.2 Slice Architecture

A slice is a component of the iGPU which is responsible for actual computation and the most relevant for purposes of GPGPU. A slice is divided into smaller regions designated subslices, which in turn contain the Execution Units (EUs), which compute the instructions given to the iGPU. Figure 2.3 depicts the general composition of a slice. In most Gen9.5 products, a slice consists of 3 subslices, which contain up to 24 EUs. A slice includes a banked L3 data cache solely for iGPU usage and a smaller but highly banked Shared Local Memory (SLM). Finally, it contains fixed functions for atomics and barriers [5].

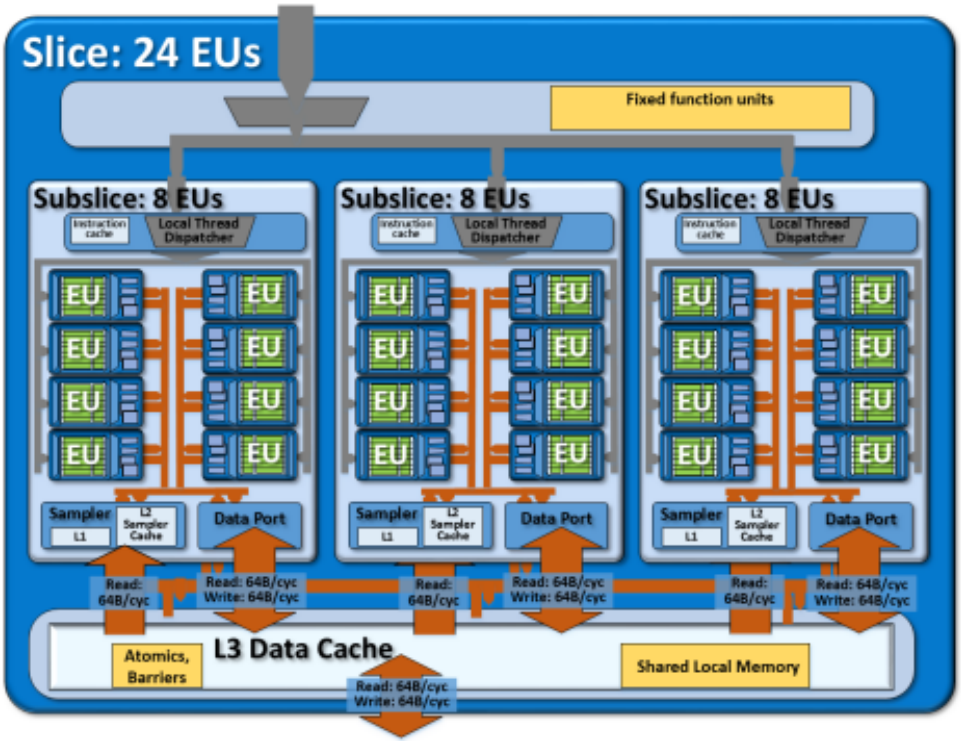

Figure 2.3: Intel Gen9.5 iGPU Slice Architecture [5]

In Gen 9.5 products, the L3 data cache shares the coherence domain with the CPU, meaning that if the CPU changes the data in the memory subsystem, the L3 cache refreshes the value of that data, to keep it coherent with the CPU. The cache has a capacity of 512 KB or 768 KB per slice (depending on the product), with 64 B cache lines. It has a read/write bandwidth to the GTI of 64 B/cycle. The cache splits itself into the contained subslices, while still acting aggregated as a single cache via an L3 fabric that is also expandable through multiple slices [5], aggregating the L3 caches of multiple slices. The subslice data ports and samplers have their dedicated memory interface to the cache, both have a read bandwidth of 64 B/cycle, and the data port also has a write bandwidth of 64 B/cycle, which can be used to align data in cache lines. Therefore, the total cache bandwidth expands to 192 B/cycle, for all subslices. The maximum cache bandwidth appears for read/write accesses that are cache-line aligned

and adjacent within a cache line. Data that goes through the L3 cache includes computation kernels that miss the subslice instruction cache and sampler data that misses the L1 and L2 caches [5].

The SLM serves as programmer-managed memory since the programmer selects which data to store in the memory block. The SLM is also a scratchpad memory, meaning that it has a high-speed and small capacity design for fast retrieval of data. It is located inside the L3 cache, having the same read/write bandwidth as the former, but more highly banked. Due to this, it can obtain full bandwidth for accesses that are not cache-line aligned or adjacent in memory. Each EU in a subslice shares the same data in the SLM unit, which can be up to 64 KB, for a total of 192 KB, assuming there are three subslices. The SLM is not coherent with the L3 cache nor with the CPU domain [5].

Each slice has logic to support barriers across groups of threads as another option to compiler-based approaches. It supports up to 16 thread groups per subslice. The atomic read/write/modify operations have 32-bit operands and are supported to L3 cache or SLM [5].

2.1.3 Subslice Architecture

Each subslice contains multiple EUs, a Local Thread Dispatcher (LTD), a sampler, and a data port. Figure 2.4 illustrates the architecture of a subslice. Gen9 and Gen9.5 products have from six to eight EUs per subslice. Apart from the computation units, a subslice aggregates an LTD with its dedicated instruction cache, a data port, and a sampler.

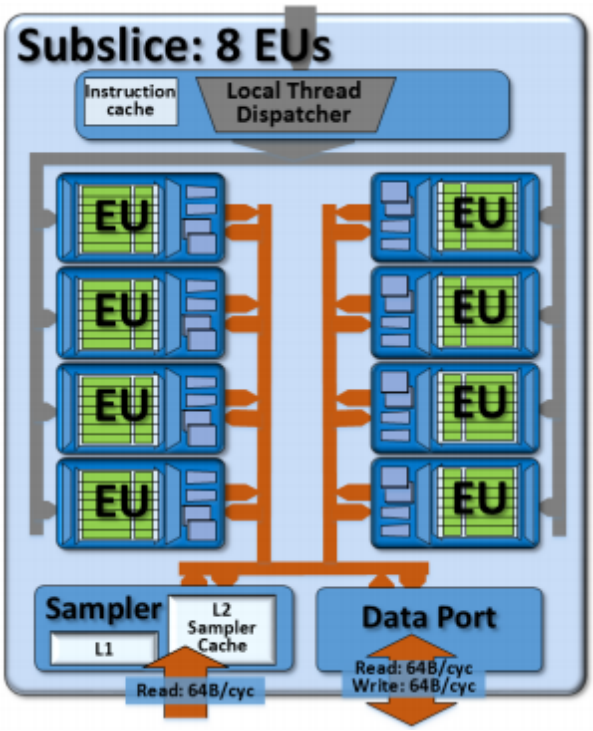

Figure 2.4: Intel Gen9.5 iGPU Subslice Architecture [5]

The LTD manages the distribution of threads among the EUs contained in the subslice. It receives the threads given by the GTD, but it does not interact back. The distribution of the threads to the EUs is made as uniformly as possible [5].

The sampler is a memory fetch unit with read-only capabilities that samples textures and image surfaces. It owns dedicated L1 and L2 data caches. It possesses logic to support dynamic decompression of block compression texture formats. The sampler's fixed-function logic enables address conversion, clamping, and sampling filtering modes. The sampler is usually not relevant for GPGPU; it is mostly required for 3D workloads. The sampler has a 64 B/cycle read bandwidth to the L3 cache [5].

The data port is the memory unit that is in charge of load and store operations. It has a 64 B/cycle write and read bandwidth to the L3 cache. To maximize memory bandwidth, it tries to coalesce scattered memory requests into one 64 B cache line. All memory load/store, Single Instruction Multiple Data (SIMD) scatter/gather and SLM accesses travel through the data port [5].

2.1.4 Execution Unit Architecture

The EU is the basic computation unit in this iGPU architecture. It computes all operations for 3D, media, or GPGPU kernels. Figure 2.5 contains the diagram of a Gen9.5 EU. All EUs have components for hardware thread management and four computation units. Thread management starts on the Instruction Fetch (IF) unit, which selects which instructions are allocated slots on specific EUs, up to a maximum of seven hardware threads at any given instant. From there, the thread arbiter picks a thread from the pool of threads that are ready to be executed, up to four at the same time, given there are only four computation units. The choice depends on the type of instructions and computation that the threads require from the EU.

A hardware thread has two batches of registers. It has a significantly sized General Register File (GRF), which consists of 128 registers of 32 B, which adds up to 4 KB of data per thread. These store data for the operation sources and destination and can be accessed as a SIMD-8 vector of 32-bit data elements or as SIMD-16 16-bit data elements, highlighting the flexibility of the addressing modes. Registers can also combine to obtain wider registers if needed. Each thread, besides the GRF, also has an Architectural Register File (ARF). These are not involved in the functional unit operations but keep track of the architectural state of execution of the thread [5].

The four computation units are responsible for executing all instructions present in the selected threads. Two of the computation units execute solely non-arithmetic, non-logic requests. The Send unit handles all load/store requests to memory or sampler operations, making use of the data port of the subslice and depending on the location of the L3 cache or the DRAM [5]. The branch unit handles SIMD the divergence and the convergence between kernels. Therefore, if kernels have diverging branches, the thread will execute both of them serially, while keeping track of it and masking some of the kernel

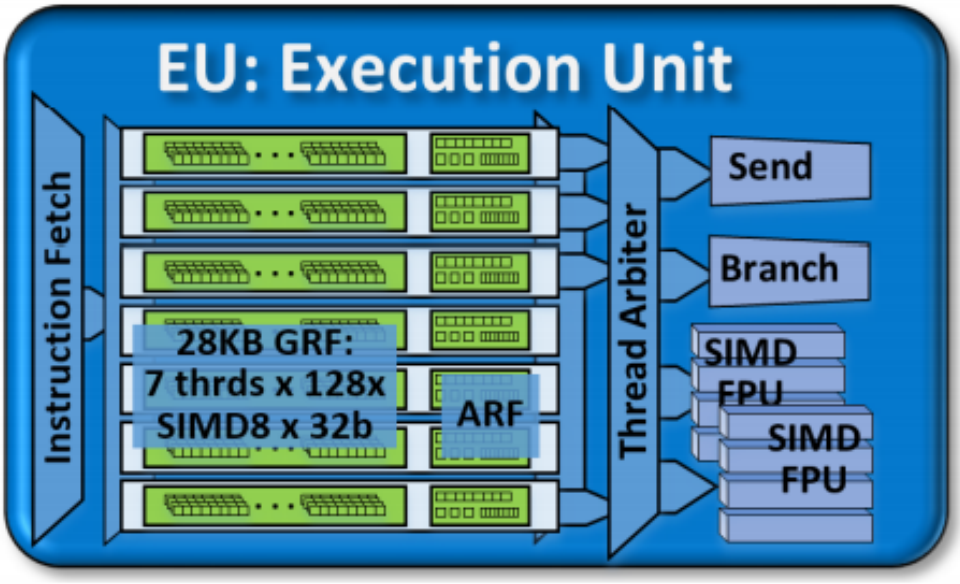

Figure 2.5: Intel Gen9.5 iGPU Execution Unit Architecture [5]

instances to indicate which ones will run, based on the branch condition [5].

The two Floating Point Units (FPUs) are the primary functional units of an EU, supporting both floating-point and integer operations. They have half-precision and single precision on floating-point computation, however only one of them can compute double-precision operations. Both FPUs can execute instructions on 4 32-bit data operands concurrently (SIMD-4), as the data bus is 128-bit wide. Again, it can support operations with SIMD-8, SIMD-16, or SIMD-32 operands, they instead take 2, 4, or 8 cycles to finish, respectively. The fully pipelined execution of the FPUs conserves the throughput efficiency [5].

The FPUs support single-precision Multiply and Add (MAD) instructions in a single cycle. Therefore, the maximum throughput per cycle per EU consists of 2 (MAD) * SIMD-4 * 2 FPU = 16 Floating Point Operation (FLOP)/cycle. For double-precision, the amount of operations in a cycle drops in half, and since only one FPU can handle this task, it drops again in half to only 4 FLOP/cycle [5].

The EU can execute up to four different threads per cycle, one per functional unit. It alternates its execution between Simultaneous Multi-Threading (SMT) and Interleaved Multi-Threading (IMT), the former meaning that multiple of the hardware threads run at the same time on different functional units. The latter means the executed thread switches between the available threads [5]. Ideally, the EU should always take advantage of SMT by having more than one functional unit processing an instruction per cycle. Since they are fit for different kinds of instructions, each EU would have to load specific instructions to keep full occupancy of the functional units, making full use of SMT, which is not likely to happen during the majority of the GPU's execution time. The IMT executed by the thread arbiter is fine-grained, which means that after the execution of a kernel instance placed in one thread, it will pick a different one

to execute and not the same one [5]. Both FPUs must work simultaneously in a large portion of the execution period to achieve the peak single-precision floating-point throughput. However, the nature of the instructions might not allow SMT, for example, if all seven hardware threads contain the same instruction for different datasets since double-precision operations are only supported by one FPU, only one FPU would be active.

2.2 OpenCL Programming Model

Most programming frameworks and languages get their code and instructions executed directly by the CPU. to properly characterize and benchmark the behavior of a GPU, software that supports GPU programming is a necessity. OpenCL is an open-source framework for unified programming in several platforms, such as CPUs and GPUs, among many product vendors. Therefore, standard OpenCL code can be performed on many distinct platforms, from computers to mobile devices to embedded systems, to parallelize and better optimize the execution speed of many real-world applications [3]. Consequently, it can also be used in many heterogeneous systems, coordinating parallel computation across different processors. All these perks establish OpenCL as a solid tool to aid in the exploration of GPGPU. OpenCL was adopted and supported by many vendors as their main GPU programming framework. As a result, compared to Application Programming Interfaces (APIs) such as CUDA, OpenCL is available in a wider range of target devices, such as GPUs, accelerators, and FPGAs.

OpenCL introduces the concept of host and device, the latter being the platform where the kernel function is executed. The host is the platform that selects the target device, copies the data to it, selects one or more kernels, and queues them up for execution on the device. The kernel functions are coded by the programmer and can have any purpose. Ideally, they should be designed to benefit from the device's architecture, adhering to parallelization, for example. The quantity and the distribution of kernel instances to run on the computation units of the device are up to the programmer. OpenCL defines those spawned instances as work items and work-groups as a group of work items. By tuning each parameter, the workload distribution can be adjusted to achieve higher performance.

The computation abilities of Intel Processor Graphics Gen9.5 iGPUs can be exploited using the OpenCL programming model. In this architecture, a single work-group shares the same local memory space. For kernels that use SLM, runtimes map all instances of a work-group to EU threads in a single subslice. Thus, all kernel instances within a work-group share the same 64 KB. Therefore, an application's access to SLM should scale with subslice number [5]. Each work-group is mapped into a subslice, therefore several instances are required to reach a GPU's maximum potential performance [12]. The EU threads in one subslice run all work items from a specific work-group. If there is not enough capacity for the incoming work items, the execution is stalled, forcing the EU threads to wait, meaning parallelization

was not possible. This event can occur when a subslice has full occupancy or when there was inefficient resource usage [12].

OpenCL usage requires manipulation of the above-referred parameters, coupled with the manufactured GPU kernels to obtain data that can be extracted to better characterize the performance of the hardware for specific operations.

2.3 Performance Characterization Methods

To evaluate how the execution of a real-world application behaves on a system, profiling tools and models that allow characterization and analysis of the results must be considered. The program execution can infer conclusions and validations about the system architecture. Models that objectively classify the main bottlenecks of the application are the ORM and the CARM. Both models provide information related to the number of operations executed and the amount of data transferred between the processor and the memory system, provide information regarding the type of workload and possible optimization steps to take to increase the device's performance. Tools such as Intel Advisor also aid in the analysis of performance results, by providing the Roofline Models for each run application, the assembly generated code for more profound analysis, and some generally useful info about GPU resources' usage.

2.3.1 Roofline Modeling

Insightful models such as roofline modeling are helpful tools that facilitate the analysis and optimization processes of real-world applications through a straightforward and perceptive approach. This Thesis covers two roofline models, the ORM [7] and the CARM [8]. Both models have the common goal of characterizing the nature of the upper boundary of the application, whether it is memory-bound or compute-bound. It provides a 2D graph, providing insight into the optimizations are required to reach the theoretical boundaries. Roofline models were originally applied to CPUs but are broad enough to adapt to GPUs.

The ORM and the CARM define the metrics Operational Intensity (OI) and Arithmetic Intensity (AI), respectively, (in FLOPs/β_D) as the amount of executed operations (ϕ in FLOPs) per accessed byte of the memory traffic (B_D in β_D/s). These are similar but fundamentally different metrics since the memory traffic notion is distinct between them. When referring to the general concept, the notation Intensity or I is used throughout this Thesis and is defined as

$$I = \frac{\phi}{B_D}. \tag{2.1}$$

Note that the memory traffic metric is different for each method. Figure 2.6 portrays the contrast be-

tween the models. The ORM characterizes it as the bandwidth between the LLC and the main memory, disregarding any requests that generate cache hits and do not reach out to the DRAM. On the other hand, the CARM establishes the memory traffic as the actual bandwidth from the viewpoint of the core for all memory blocks.

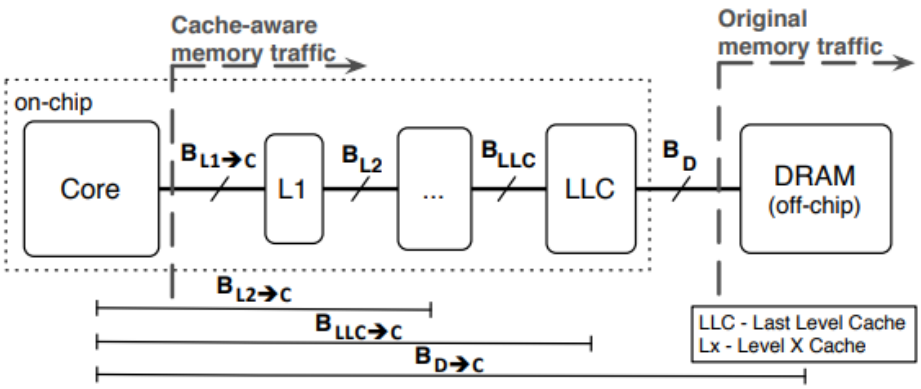

Figure 2.6: Comparison between ORM's and CARM's notion of memory traffic [8]

The metric above defined is the horizontal axis of the roofline models. The vertical axis corresponds to the throughput of the application, labeled as Performance, in Floating Point Operations per Second (FLOPS). Considering F_p, in FLOPS, as the peak floating point throughput of the device, the maximum attainable performance ($F_a(OI)$ in FLOPS) is obtained according to equations 2.2 and 2.3, for the ORM and the CARM, respectively

$$F_a(I) = \min\{B_D * I, F_p\} \tag{2.2}$$

and

$$F_a(I) = \min\{B(\beta) * I, F_p\}\,. \tag{2.3}$$

The only distinction relates to the utilized bandwidth, the ORM defines the bandwidth of a memory level, usually from the LLC to the DRAM, while the CARM can substitute the function $B(\beta)$ by $B_{L1\to C}$, $B_{L2\to C}$, $B_{L3\to C}$ or $B_{D\to C}$, in the case of a general CPU, and obtain the bandwidth from the core to that memory level. Figure 2.7 presents examples of the ORM and the CARM plots for a specific device.

In the ORM, for a given OI of the application and an obtained throughput value, the nature of the upper boundary can be immediately deducted: if the point is below the roof-shaped part, the application is memory-bound; otherwise, it is compute-bound. The ORM can benefit from the addition of more ceilings that may help characterize the required optimizations.

In the CARM, a significant unexplored area of the roofline plot is now revealed, corresponding to the

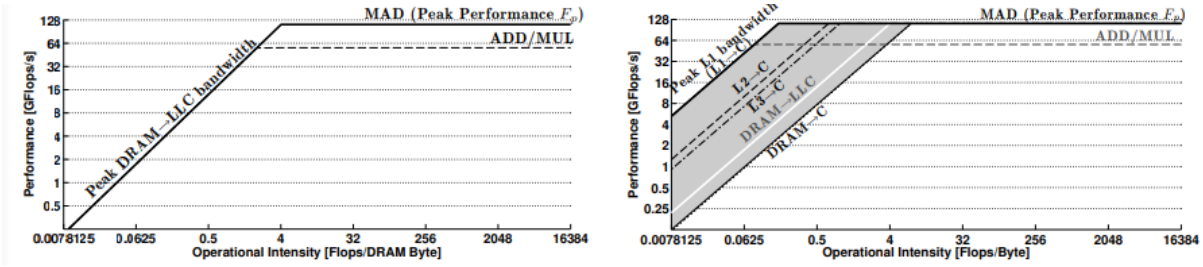

Figure 2.7: Comparison between ORM and CARM plots [8]

caches' bandwidth, the highest being the L1 cache. If the tested application sits in the middle of the plot, the results can be inconclusive regarding the essence of the boundary. The CARM can also profit from the inclusion of new ceilings for optimization purposes. MAD instructions, ADD/MUL instructions, or SIMD utilization are examples of new roofs to include in both models.

The models retain a few key differences. From Figure 2.7, the memory roof for the CARM is presented lower than its counterpart in the ORM. It is due to the measurement of bandwidth in the CARM, which takes into account the passage through all caches, lowering the DRAM bandwidth. Other differences reside in distinct problem sizes. Figure 2.8 shows tests for both roofline models for various amounts of iterations.

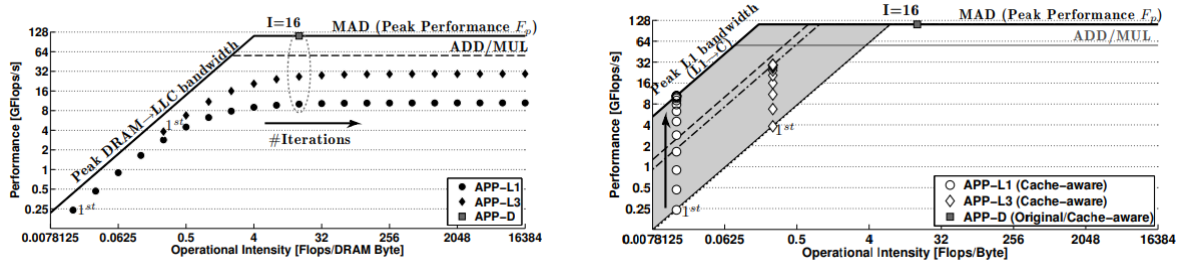

Figure 2.8: Comparison of ORM (left) and CARM (right) for varying size benchmarks [8]

Two different problem size applications were tested, APP-L1 and APP-L3, each one limited by the corresponding cache level of a particular CPU. In the ORM, the AI gradually increases with the problem size, which can modify an application from memory-bound to compute-bound or vice-versa, while for the CARM it does not change, meaning a specific application will always be either memory-bound or compute-bound. This distinction between the models relates to the different takes on memory traffic for the notion AI, as shown in Figure 2.7. Note that with the iteration increases for both APP-L1 and APP-L3, the CARM plot slowly reaches the peak L1 and L3 bandwidths, respectively, while no peak is achieved in the side of ORM. For APP-D, limited to the DRAM and compute-bound, both models obtain a similar plot, independent of the iteration quantity. Due to the key distinctions pointed out between and to the integration of the CARM in Intel Advisor, the CARM will be the preferred model throughout this work.

2.3.2 Intel Advisor

Intel Advisor is a set of performance profiling and analysis tools designed to help improve and optimize applications to run on Intel processor architectures. The software application provides survey analysis, identifying vectorization or threading opportunities and trip counts analysis, recognizing function loops and characterizing them. Additionally, it can perform FLOP analysis by measuring operation metrics and bandwidth and roofline analysis, with the integration of the CARM, illustrating the location of the applications on the plot.

2.4 State-of-the-Art Works

Although there are not many dedicated papers to studying iGPU in comparison to their dedicated counterparts, some relevant works for this Thesis, even about CPU and simulations, are present in Table 2.1. These papers include algorithms tested in iGPU, scheduling methods between CPU and GPU, roofline developments, and others.

Table 2.1: Relevant state-of-the-art works

Paper	Year	Platform	Objective
[13]	2012	CPU+iGPU	Comparison of FDTD algorithm
[14]	2014	CPU+GPU	Adaptive heterogeneous scheduling
[15]	2016	iGPU	Adaptive graphics performance model
[16]	2016	CPU+iGPU	Heterogeneous scheduling through work-stealing
[17]	2016	Simulator	Evaluating the effect of last-level cache
[18]	2016	iGPU	Analysis of work-group reduce
[19]	2016	iGPU	Analysis of work-group broadcast
[20]	2016	CPU	Power and energy CARM to CPU
[21]	2017	GPU	Performance, power and energy CARM to GPU
[22]	2017	iGPU	Improve sort algorithms' performance
[23]	2017	iGPU	Adaptive graphics power sensitivity model
[24]	2018	iGPU, Simulator	Performance characterization and modeling

In [15] and [23], the graphics portion of the Intel iGPU is put to the test through the benchmarking of games and other 3D applications. Both papers resort to an offline feature selection on which a regression technique was applied, and an online learning method. They use a lightweight recursive least-squares method to accurately predict the change in frame time and power, respectively.

The authors of [13] compared the execution of the Finite Difference Time Domain (FDTD) algorithm in an iGPU to a discrete GPU. The FDTD is a method to model electromagnetic fields, requiring several iterations of FLOPs. The established procedure was to divide the chunks between the host (CPU) and device (iGPU) and exchange data after one chunk calculation and repeat these steps. The programming

API used was DirectCompute. As for the final results, the iGPU managed to obtain almost twice the throughput of a low range discrete GPU. It should be noted that a small dataset was used, which allowed the iGPU to prevail over the discrete one.

In [14], a compiler/framework for heterogeneous systems was designed, to distribute the workload between the CPU and other devices. This work was developed using the OpenMP API. The compiler translates specific regions of the code and translates them to GPU kernels and generates a CPU parallelized version in case the communication with the GPU fails. The compiler collects data from the program the builds the performance models; however, the final evaluation is only known at runtime. A prediction is generated with the obtained data of the advantages of offloading the work to the GPU. According to the result, either of the generated code versions gets executed.

The paper [16] tackles the same problem with a different approach, planning affinity-aware work-stealing from one of the devices to the other when stalled. Since a CPU and a GPU have different operation frequencies, the CPU tends to steal excessive work from the GPU, which is not always beneficial. In this concept, it is crucial to reduce the amount of steals and maximize the transferred data per steal. The paper uses lightweight online scheduling to distribute the initial work among the devices and hierarchical work-stealing. As such, the stealing attempts between different devices are mitigated. This approach obtained from 20% to 100% improvements in several tested benchmarks.

In [24], a significant performance characterization is made to an Intel iGPU. Most of it will be covered since it is relevant to this work. These tests were made in the Intel Skylake [6] and Intel Kabylake [6] iGPU architectures.

In this article, through the usage of OpenCL kernels, set up with work-groups composed of 32 work-items each, the number of work-groups was varied, for single-precision and double-precision for MAD operations. The authors concluded that the peak throughput is achieved with 32 work-items per work-group and for 96 and 192 work-groups for SKL and KBL architecture, respectively. For reduced work-group amounts, the throughput is low due to the lack of operations to saturate the EUs.

The next microbenchmark of [24] consists of a single-threaded random access test for varying set sizes, to obtain the access times to different memory blocks. For the smaller set sizes, the access time proved to be constant for the SKL iGPU, spiking at a size of 512 KB, which is the standard size of the L3 cache. It is expected, as any larger work set size forces the iGPU to access the LLC. Whenever the set size becomes larger than the L3 cache, the access time does not remain constant for the LLC accesses. Therefore, the authors believe that the iGPU does not take advantage of the full capacity of the LLC or that some space may be reserved for the CPU.

A memory coalescence test was also performed with a pointer chasing microbenchmark, with varying strides between accesses. For a single work-group, the bandwidth is quite low as not enough memory accesses are made. For increasing numbers of work-items and work-groups, the memory requests

increase too, which increases the obtained bandwidth. A wider stride also makes more requests, so fewer work-groups and work-items are required to achieve maximum bandwidth.

The work developed in [17] shares similarities with [24] by analyzing the effect of the LLC in computation. The gem5 simulator was used for this work, representing an integrated CPU-GPU system. The LLC proved to cause at least a slight speedup in the tested microbenchmarks, particularly in operations requiring fine-grained synchronization and atomic function usage. Data sharing between the devices also presented improvements, causing reduced memory access latency.

The papers [19] and [18] intend to explore the Intel iGPU microarchitecture and to test OpenCL kernel algorithms for work-group broadcast, and work-group reduce operations, analyzing the hardware behavior. The used Intel Runtime for GPUs was Beignet, which converts the OpenCL kernels into the iGPU assembly instruction set through an LLVM compiler. The low-level assembly was thoroughly analyzed in both papers.

The broadcast operation consists of having a hardware thread writing to shared local memory and reading the data from all other hardware threads. On the other hand, the reduction operation reads from all hardware threads and writes the data to one of them. In the paper, the best throughput and lowest latency are achieved for 64 and 128 work-items per work-group. Note that for MAD operations, tested in [24], 32 work-items were enough to achieve maximum throughput, however, for the tested operations in [19] it falls short of the expectations.

In the reduction addition operation, two methods were tested, a thread message-passing variant and a SLM variant. In the former, each thread uses the send unit to write the processed data to another hardware thread, until all passed a message. The last one passes the final data back to the first hardware thread, which writes the data into all of the others and to the main memory. In the SLM variant, each thread writes its partial result to the SLM. Each thread would then read all values from SLM and finish processing the result of the reduction. Similarly to [19], 64 work-items per work-group achieve the best throughput of all the options. Also, the SLM variant seems to obtain generally better values for all iterations, which induces that the message-passing and barriers overhead are a relevant issue that sets the thread message-passing variant back.

In [22], an integrated CPU + GPU system was used to speed up the execution of a sorting algorithm, serially, using OpenCL. Two kernels were tested, one sorts data in the DRAM, the other in the SLM, the latter gets and sends data by chunks from the DRAM to the SLM. Obtained results reported up to half the execution time thanks to the iGPU offloading. The take from this work is that the usage of SLM significantly reduces the impact on the DRAM speed. Due to sharing the memory subsystem, iGPUs are particularly useful in executing offloaded work from CPUs.

This Thesis will also cover an extension of CARM for power consumption and energy-efficiency, which was developed in [20]. The approach starts by defining three power consumption domains, the

core P_c, the uncore P_u, and the package domain P_p, the latter corresponding to the whole chip. Microbenchmarks for throughput and bandwidth ran on specific CPU. The power and energy levels were obtained through the Running Average Power Limit (RAPL) interface. Figure 2.9 illustrates the variation of the three power domains with increasing data traffic for the memory subsystem test and increasing FLOPs for the throughput test.

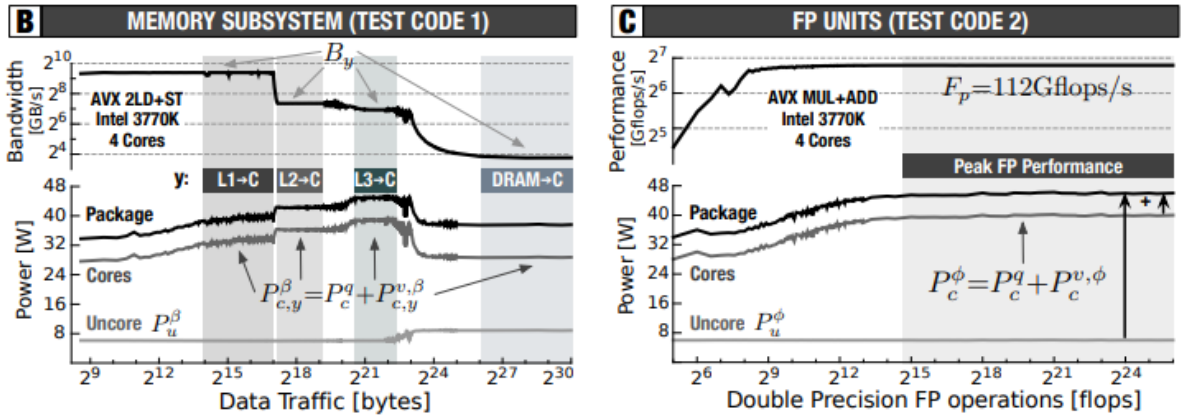

Figure 2.9: GPU performance and bandwidth tests with power measurement [20]

From the bandwidth test, analyzing the core domain, the relatively static power value in all memory blocks is noticeable. The power increases for higher cache levels due to the activity of the lower-level caches but decreases from the L3 cache to the DRAM. It happens due to a significant decrease of bandwidth $B_{D\rightarrow C}$ when compared to $B_{L3\rightarrow C}$, which keeps the caches stalled and therefore reduces the power consumption [20]. The uncore domain has static power all around, except for a slight increase from the L3 to the DRAM, because of more utilization of memory-controller and interconnects [20]. In the performance test, the power consumption attains a constant value as expected once the FLOP peak is achieved. It starts at a lower value due to a reduced amount of operations that do not saturate the pipeline [20]. In the power CARM, the ridge point for the core domain is expected to be the value for the highest power consumption, since both memory operations and FLOPSs are at their maximum amount [20]. Regarding the uncore domain, its power value remains constant for any memory operations at cache levels and any FLOPs, dropping lower for any DRAM accesses, as seen before. Figure 2.10 shows a Total Power CARM, a Total Energy CARM and a Total Energy-efficiency CARM for the executed tests.

The Total Power CARM incorporates the power consumption for all memory levels while also considering the gradual transitions between them. The energy consumption part of the CARM appears to be constant along the memory-bound region and increases with the increment of AI along the compute-bound part, for all memory level roofline traces.

Finally, the energy-efficiency plot is similar to the standard CARM, in the sense that it achieves

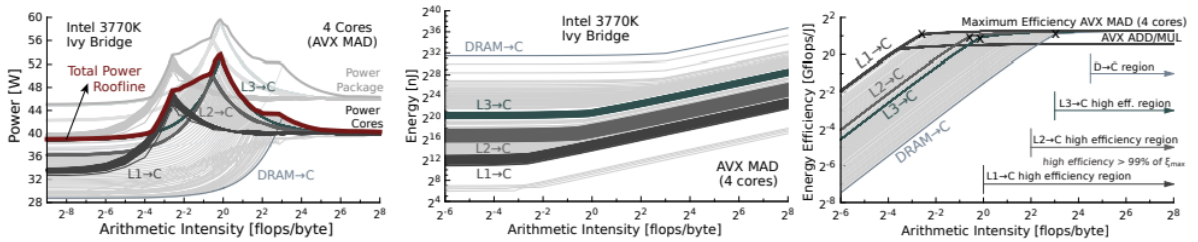

Figure 2.10: Total Power CARM (left), Total Energy CARM (middle) and Total Energy-Efficiency CARM (right) [20]

maximum values in the compute-bound zone, with the L1 cache needing a lower AI threshold to achieve maximum energy-efficiency. The higher the memory level, the higher is the threshold.

The above-referred approach for roofline models was tested in CPU devices. In [21], a CARM for GPUs is proposed. The tests and validation were done on multiple dedicated Nvidia GPU architectures. The first step to fabricate a roofline model for a GPU taking into consideration performance, power consumption, and energy-efficiency should be to reach the theoretical upper-bounds of the architecture. The developed microbenchmarks in [21] target the execution units for throughput, the shared local memory, and L1 cache, the L2 cache, and the device memory. Figures 2.11 and 2.12 present the obtained roofline models for throughput, power and energy-efficiency, respectively.

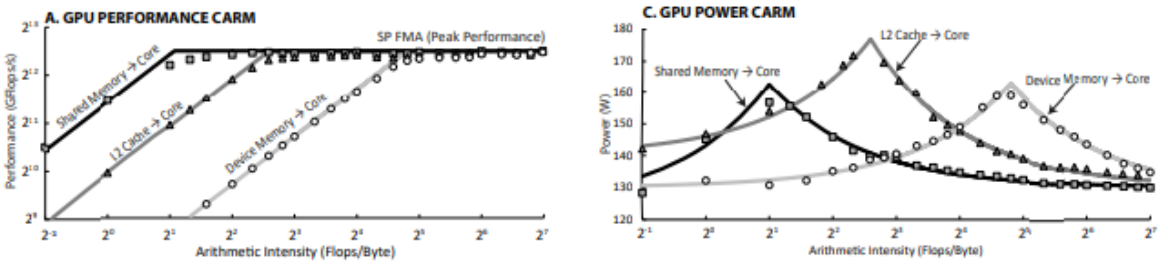

Figure 2.11: GPU CARM for performance and power [21]

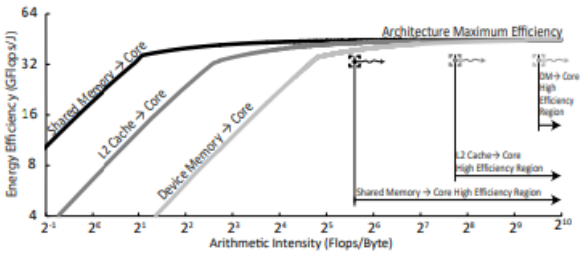

Figure 2.12: GPU CARM for energy-efficiency [21]

2.5 Summary

The Chapter introduced and described the microarchitecture of an Intel Gen9.5 iGPU, covering its most relevant aspects for GPGPU usage, while also characterizing the behavior of other functions such as the 3D or the media pipelines. Deep knowledge of the hardware architecture is imperative to actively analyze and comprehend specific behaviors when microbenchmarking the hardware architecture. The Chapter also introduced OpenCL, which is the programming model utilized during this Thesis to develop kernels to microbenchmark the iGPU architecture, as well as some tools to aid the interpretation and reasoning based on the obtained results. The insightful roofline modeling technique was covered in-depth, for CPU and discrete GPU devices, in performance, power, and energy-efficiency. Lastly, the Chapter state-of-the-art works mostly regarding benchmark tests run in iGPUs, reflecting on the achieved results and authors' conclusions.

3

Intel iGPU Performance Characterization

Contents

Various metrics are of critical importance in the performance of any modern GPU. Due to cost, size, or other physical restraints, a compromise in performance always has to be reached. Utilization of this hardware for GPGPU demands a high level of parallelization in the code to reach its upper compute bounds. Memory bandwidth is also a valuable metric, as the necessity to transfer data between CPU and GPU or GPU and DRAM is very much present. Lastly, energy expenditure is also relevant due to the iGPU's location in the architecture. To characterize and evaluate the performance and energy-efficiency potential of this iGPU, the work developed in the scope of this Thesis includes several microbenchmarks and evaluations involving the above-referred metrics.

This Chapter focuses on the approach taken to attempt to characterize a Gen9.5 Intel iGPU. It involves several microbenchmark tests, developed in the C/C++ language, via the OpenCL programming model for GPU programming. The set was purposely designed to point out the main aspects of the hardware architecture and determine the upper bounds of performance in the iGPU. Therefore, the Chapter includes an in-depth description of the standard common-ground OpenCL source code, enforced in the vast majority of all OpenCL applications. A discussion in time and performance metrics' measurement is presented. Additionally, an overview of external tools adopted to improve the quality of the microbenchmark set, such as Performance Application Programming Interface (PAPI), is provided. PAPI is an external library inserted onto the benchmark set that allows access to collections of performance measurement data, resorted to in this Thesis to seize energy expenditure metrics through the RAPL interface [25]. Lastly, the compiled GEN Assembly code from the developed kernels is introduced, as its analysis can capture compelling statistics about the device.

3.1 OpenCL Benchmark Architecture

The microbenchmark set was developed in C/C++, employing OpenCL to operate in the iGPU. OpenCL has a set of initialization functions used independently of the type of kernels chosen. Consequently, these functions are common to almost all OpenCL benchmarks. Their general purpose is to select the device to work on, whether CPU, GPU, or other, to create a queue for all the commands to execute on that device, and allocate the needed device memory for the data objects. Afterward, the data has to be copied to the device so the latter can eventually execute the kernels. If required, the output data is transferred back to be read by the host. Due to the similar architecture of many benchmarks, the variations between the different proposed tests demand different kernels and redefined object data types and sizes, as the main data transfer and kernel execution processes do not change. Figure 3.1 presents a detailed flow chart of the general microbenchmark setup and behavior, describing all the established methods from the start to the end of the application. As a complement, Listing 3.1 illustrates and explains the selected functions in the source code to achieve the tasks at hand. Note that only the

functions preceded by ’cl’ are external OpenCL functions.

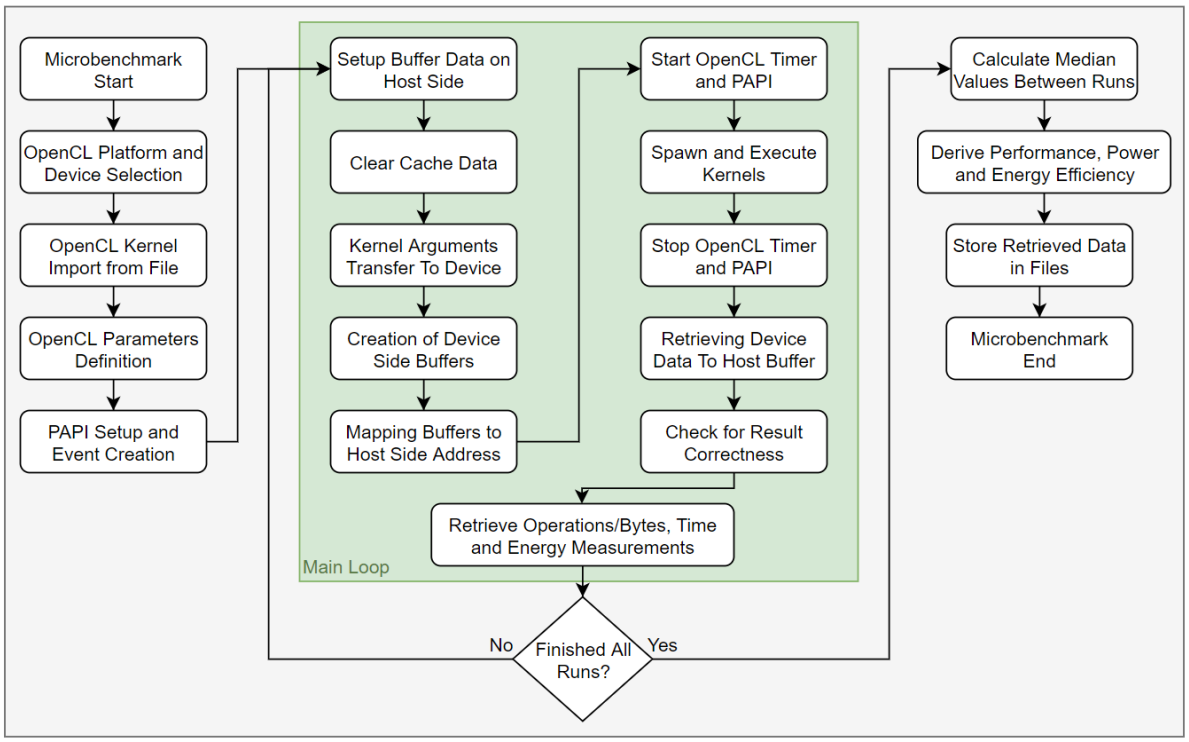

Figure 3.1: OpenCL Microbenchmark Architecture and Behavior Flowchart

Listing 3.1: OpenCL Benchmark Architecture

```
% Functions to identify and select the device to send the workload to
clGetPlatformIDs(platform, ...)
clGetDeviceIDs(device, platform, ...)

% Initializing a kernel command queue for the above selected device
clCreateContext(device, ...)
clCreateCommandQueue(context, device, ...)

% Read the binary/text kernel source file and compile OpenCL kernels from it
clCreateProgram(context, kernel_file, ...)
clBuildProgram(program, device, ...)

% Set up kernel and its arguments
clCreateKernel(program, ...)
clSetKernelArg(kernel, device_buffer_A, ...)
```

```
clSetKernelArg(kernel, device_buffer_B, ...)
...

% Allocate buffer objects in the host memory (CPU)
alloc_buffer_A(host_buffer_A, data_size)
alloc_buffer_B(host_buffer_B, data_size)
...

% Create the buffers in the device memory (GPU)
clCreateBuffer(context, host_buffer_A, ...)
clCreateBuffer(context, host_buffer_B, ...)
...

% Copy the buffer from the host memory (CPU) to the device memory (GPU)
clEnqueueWriteBuffer(queue, device_buffer_A, host_buffer_A, ...)
clEnqueueWriteBuffer(queue, device_buffer_B, host_buffer_B ...)
...

% Execute the kernels for given data size parameters
clEnqueueNDRangeKernel(queue, kernel, global_size, local_size, ...)

% Copy the buffer back from the device memory (GPU) to the host memory (CPU)
clEnqueueReadBuffer(queue, device_buffer_A, host_buffer_A, ...)
```

OpenCL provides a list of all compatible hardware devices, from several vendors, that are accessible in the machine running the set of microbenchmarks. It includes multiple CPUs, GPUs and FPGAs. To achieve this, the functions clGetPlatformIDs() and clGetDeviceIDs() choose a device to be the target setup for the application. In this application, an Intel Gen 9.5 iGPU is selected through these commands. Following that, several initialization functions are required to set up the OpenCL kernels. These include the creation of a command queue and a context for the kernels on the selected device; The kernels selected for microbenchmarking are stored in .cl files and parsed to string types through the host code. Multiple kernels are present in one .cl file, corresponding to different vectorization levels for the same GPU operation. The intended vectorization data type is given as an argument of the application, so only the respective kernel gets copied. The given kernel input string is a required parameter for the clCreateProgramWithSource(), and consequently, the clCreateKernel() functions, allowing the compilation of the kernels to start their execution later. By this step, kernel compiler flags can be defined. The maximum achievable throughput in this architecture requires MAD operations and high amounts of parallelization. To do so, the flag '-cl-mad-enable' must be set. Other available flags involve compiler optimizations and

math simplifications, both of which were not used in the Thesis, as any hidden optimization can give unpredictable results and compromise the validity of the run tests.

The integration of the iGPU on the same SoC as the CPU directly implies that the former does not have dedicated memory space to copy the buffers transferred by the host. Therefore, the requested objects are again stored in the main system's DRAM, as it is the main memory used by the iGPU. As such, OpenCL supports mapping the device buffer to the host buffer's address, enabling a feature designated by 'zero copy'. This mapping process is exploited in the clCreateBuffer(), clEnqueueWriteBuffer(), and clEnqueueReadBuffer() functions, through specific flag usage. This specification can be an advantage compared to dedicated GPUs, which might require the device to receive a fully copied buffer transferred via PCIe, as they usually possess dedicated memory. The execution of the actual kernel in the clEnqueueNDRangeKernel() function depends on the given values for 'global_size' and 'local_size'. Those parameters define how many global work items are executed and how many work items a work-group contains, respectively. The variation of these parameters is present in a significant amount of tested microbenchmarks, not only to vary the tests in problem size but also to vary the organization of these work-items inside the iGPU. Therefore, the called function generates several work-items with the given kernel equal to the value present in the 'global_size' parameter. Depending on the operational intensity of the kernel, the iGPU parallelizes the requests to reach acceptable performance values. After the kernel execution and full retrieval of buffer data by the host side, a result check is performed to ensure that the GPU operations correctly took place and fulfill all expected results.

To microbenchmark and measure statistics from the hardware's behavior, a precise profiling tool is required. The chosen option is also integrated into the OpenCL programming model in the form of events, which ensures reliability and reduced overheads. Listing 3.2 presents the required functions to profile an OpenCL kernel.

Listing 3.2: OpenCL Profiling Tool

```
1 % The parameters define when to start and end profiling the application
2 clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, &time_start, ...)
3 clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, &time_end, ...)
4 time = (cl_double)(time_end-time_start)
```

The function parameters described in Listing 3.2 ensure the measurement only from the start of an event to its end. The selected event was the clEnqueueNDRangeKernel() function which provides the sole calculation of the kernel time execution. This choice is justified since the only interest here is to measure the work done by the GPU, provided by the developed kernels. This profiling method is used consistently and with the same precision among all developed kernels and benchmark tests in this Thesis to ensure coherency between results. The performance measurements for throughput and

bandwidth were calculated manually, by assessing the number of data operations executed, the size of the data buffer objects, and their respective data types.

The throughput and bandwidth calculation require the amount of executed operations and the number of transferred bytes, respectively. These parameters are calculated according to the size of the transferred objects, the size of the selected data type, and the number of iterations done on each operation. As such, throughout this Thesis, throughput results are presented in FLOPS or Integer Operations per Second (INTOPS) and bandwidth results are displayed in bytes per second. The execution time, operations/bytes, and energy expenditure measurements are collected for multiple runs and their medians are calculated, in order to avoid displaying outlier values and improve the consistency of the results.

PAPI was relied upon in order to obtain the amount of energy spent during the execution of a kernel in the GPU. Those values, coupled with previously obtained data on execution time, throughput and bandwidth, power requirements, and energy efficiency values are derived. Listing 3.3 illustrates the usage of PAPI functions to achieve the above-mentioned results. PAPI creates an event set, where each event is a specific data collection, which may or may not be supported by the device being benchmarked. In this benchmarking set, the handled event is pertained to the RAPL interface and designated 'rapl::RAPL_ENERGY_GPU', collecting data of energy expenditure on the GPU. Through other PAPI events, a collection of data of the heterogeneous system CPU+GPU or of energy consumption on DRAM accesses is also possible. The data collection begins and ends on command with the functions PAPI_start() and PAPI_end(), inserted right before and right after the kernel execution function, respectively, in order to properly measure the iGPU energy consumption throughout the whole kernel execution. The collected data is stored into a buffer, which is then used, together with other measurements such as kernel execution time, executed operations, and transferred bytes; to derive power requirements (in Watt or Joule/second) and energy efficiency in (GFLOP/Joule).

Listing 3.3: PAPI Functions

```
% Library inialization and creation of a set of multiple events
PAPI_library_init()
PAPI_create_eventset()

% Define events to add to the set
PAPI_add_named_event(event_set, event_name)
...

% Start and stop data collection
PAPI_start()
```

```
11 ...
12 <kernel execution>
13 ...
14 PAPI_stop()
```

The main goal of this implementation was to experiment and confirm the hardware theoretical boundaries, by pushing its workload to the limit in both throughput and bandwidth. Both power requirements and energy-efficiency measurements were also considered relevant and, as such, were derived from the microbenchmarks. The relevant obtained results are presented in 4.

3.2 OpenCL Kernels for iGPU Architecture Microbenchmarking

An OpenCL kernel is an object, containing custom functions that can be executed on OpenCL compatible devices. Through host programming, the amount of kernels that are spawned and their organization among OpenCL work-groups and work-items is completely customizable. Through the help of OpenCL functions such as get_global_id() or get_local_id(), which return the index of the containing work-item in their respective group, one can manipulate the workload that executes on each work-item to improve parallelization and remove unnecessary calculations, aiming for faster execution times by evenly splitting computation by computation units. The amount of spawned kernels is defined in the clEnqueueNDRangeKernel() function by the 'global_size' and 'local_size' parameters. Each work-item requires elements from one or two buffer objects, calculates the result of a specialized operation, and outputs the result onto one of the arrays. The index of the element being accessed corresponds to the index of the work-item relative to all spawned work-items, through the OpenCL function get_global_id(). Work-items are scattered all over the iGPU since no dependencies between operations allow for the maximum amount of concurrent execution and no serialization. This is the main process of all developed kernels, which are presented in Table 3.1.

Table 3.1: Developed OpenCL Kernels for iGPU Architecture Microbenchmarking

Listing	Measurement	Operations	Data Type	Vectorization
3.4	Throughput	Addition	Int, Float, Double	1, 2, 4, 8, 16
3.5	Throughput	MAD	Float, Double	1, 2, 4, 8, 16
3.6	Bandwidth	Load/Store	Float	1, 2, 4, 8, 16
3.7	Bandwidth/Latency	Load	Float	1

For throughput kernels, three distinct data types are tested, floating-point single-precision, floating-point double-precision, and integer. Buffers consist of different vector types for the developed kernels that support them. They range from scalar to vector2, vector4, vector8, and vector16. The change in the

size of each element due to vector types was also taken into account and adjusted so that every iteration had the same amount of executed operations, to observe any compilation changes given different buffer vectorizations and any performance differences among them. These data types are tested for both addition and MAD operations, with the latter having '-cl-mad-enable' included in the kernel compiler flags. Only MAD is expected to achieve maximum throughput, as the operation implies both multiplication and addition are done in one single clock cycle, effectively doubling the throughput of an ADD instruction. Listings 3.4 and 3.5 depict examples of the developed kernels to attempt to reach the maximum throughput possible in the architecture, in ADD and MAD operations, respectively.

Listing 3.4: ADD Scalar Single Precision Kernel

```
1 __kernel void add_float(__global float *a, __constant float *b)
2     i = get_global_id(0)
3     if i < DATASIZE
4         for j < ITERATIONS
5             a[i] = a[i] + b[i]
```

Both kernels run the same operation for a set number of iterations. This amount is inversely proportional to the vectorization level per element to preserve the number of operations between tests. The buffers are large enough so that each work-item can work on a single element of the array. The logic behind both kernels is to have a large number of iterations, forcing the iGPU to fill up with the same instructions, spread over all EUs. Therefpre, each work-item processes a single element of the array for organization purposes. Note that the two buffers are in distinct OpenCL memory spaces. Both translate into the entire memory hierarchy of the system, but the constant memory space is read-only.

Listing 3.5: MAD Scalar Single Precision Kernel

```
1 __kernel void add_float(__global float *a, __constant float *b)
2     i = get_global_id(0)
3     if i < DATASIZE
4         for j < ITERATIONS
5             a[i] = a[i] * b[i] + b[i]
```

Through testing, the global prefix on both buffers would yield much lower throughput. This occurrence implies that the 'b' buffer was not being cached and requested the element loads from the main memory. As a result, the buffer 'b' is defined in the constant memory level since it does not need to be written to in any of the kernels, read-only accesses are sufficient. It achieved near-maximum throughput during performance characterization tests in both addition and MAD tests, as they share the same kernel, data

types, and the same FPU usage, differing only in the instruction itself.

To achieve the maximum possible bandwidth, the L3 cache, present inside a slice of the iGPU has to be exploited to the maximum. This approach requires significant reuse of the values stored in the cache, to avoid, as much as possible, cache misses and travel times to the DRAM, which has much lower bandwidth. This result is achieved by forcing repeated loads from the data stored in the cache. Bandwidth tests were executed for floating-point single-precision and double-precision data types. The exclusion of integers is due to similar size and instruction execution relative to the float data type, originating naturally equal results. The first presented kernel includes both load and store instructions, for all vectorization ranges, while the second one only runs a scalar pointer chasing algorithm, attempting to minimize the number of store instructions and deriving the memory bandwidth measurement from load instructions. A single-threaded version of the second kernel is also used to measure memory latency to all caches and main memory. Listings 3.6 and 3.7 illustrate the differences between the memory kernels.

Listing 3.6: Load/Store Vector4 Single Precision Kernel

```
1 __kernel void load_store_float(__global float4 *a, __constant float4 *b)
2     i = get_global_id(0)
3     for j < ITERATIONS
4         a[i] = b[i]
```

The kernel presented in Listing 3.6 attempts to achieve the maximum possible bandwidth for increasing buffer data sizes, as having each work-item fetch the same values repeatedly intends to maximize the access trips to the L3 cache, to mask the latency of the few DRAM accesses. As such, the underlying assembly code behind the compiled kernel is almost exclusively filled with load and store instructions, ideally to the L3 cache, maximizing the usage of all subslices' data ports. Similar kernel attempts were created with load instructions exclusively, without requiring a second instruction type in stores. However, the compiler would proceed to kill these attempts. It understood that the loaded data would not be stored in memory, thus having no use, resulting in empty kernels. This behavior induced attempts that resulted in the kernel displayed in 3.7.

Listing 3.7: Pointer Chasing Scalar Single Precision Kernel

```
1 __kernel void pointer_chase_float(__global float *a)
2     i = get_global_id(0)
3     for j < ITERATIONS
4         ptr = id
5         for i < DATASIZE
```

```
6               ptr = a[ptr]
7           a[id] = (ptr + STRIDE) mod DATASIZE
```

The pointer chasing algorithm in Listing 3.7 traverses the entire array depending on the constant stride value. Variations in the stride can change the access patterns, and variations in the buffer data size can affect which memory region has the required data to fetch. As referred before, the sole measurement is the load instructions, as a store instruction is only present at the end of the kernel to preserve the integrity of the loop. Each work-item traverses through the entire array, jumping between elements according to the stride value. After traversing through the whole array, it stores the corresponding value not to compromise the strides of the array accesses. These are run in an outer loop of more iterations to saturate the EUs on instructions and verify that none end up stalled for long periods, as other kernel attempts without the loop proved to achieve worse performance.

For latency tests, a variation of the same kernel is performed, with the difference that only one work-item is needed, as no saturation on the computation units is required to measure the memory access time. As such, the get_global_id() function is not necessary for this implementation. In this, the kernel execution time is divided by the number of data transfer operations made. The outer loop proves to be useful as it reduces the impact of the beginning cache misses on the obtained results.

3.3 Additional Analysis Tools

A direct comparison of the implemented benchmark with the tested hardware limits can be obtained with the help of the Intel Advisor software. Furthermore, a roofline model report is provided as well. The computation and memory boundaries for different steps and optimizations are marked, suggesting options for further performance improvement of the application. The CARM is the depicted model for the ability to illustrate more clearly the requests done to all the memory hierarchy as opposed to solely the DRAM. CARM reports are presented and further analyzed for some of the implemented kernels and compared to the obtained throughput results. Furthermore, through Intel Advisor or terminal commands, one can obtain the underlying GEN Assembly code, generated by the compiler, to be executed on the device. An in-depth analysis of the low-level code can be required to understand specific behaviors of the iGPU. The GEN Assembly code allows for easier comprehension of the generated instructions, of the register region allocation for the data employed in every instruction, and of the required debugging process for each kernel [26].

3.4 Summary

In this Chapter, the developed OpenCL microbenchmark set is showcased and explained in-depth, as the application behavior is of significant importance to achieve valid results for performance characterization of the iGPU. These benchmarks' goal is to attain measurements that reach the upper theoretical limits of the GPU in throughput and memory bandwidth and to assess the required power and consequent energy efficiency values while running the kernels. The Chapter introduces the handled profiling tool in the entire benchmark set, associated with the OpenCL programming model, and external tools such as PAPI, which is the provider of collections of GPU data related to energy consumption values. The kernels designed to complement the microbenchmark set are also described in this Chapter, as well as the general GEN Assembly code generated by them, with the latter being analyzed in Intel Advisor. The analysis of the generated GEN Assembly code can provide deep insight and justification to the behavior of the iGPU in certain situations when the obtained data is unpredictable.

4

Experimental Evaluation and Data Analysis

Contents

The Intel Gen 9.5 iGPU architecture was the target of an extensive performance characterization, in terms of computation throughput, memory bandwidth and latency, power consumption, and energy efficiency. Further analysis was conducted through the usage of CARM, through the Intel Advisor software. The experimental setup and its specifications for all executed tests are described in this Chapter. Furthermore, the Chapter contains the experimental results, both for microbenchmark measurements and CARM analysis plots, solidified by a discussion about the relevant data.

4.1 Experimental Platform and Setup

All tests were executed on the same setup. The target processor for this microbenchmarking set was an Intel Core i5-8300H CPU, a 64-bit x86 processor. It is based on the Coffee Lake microarchitecture, operating at a frequency of 2.3 GHz. The CPU is partnered by an Intel UHD Graphics 630 iGPU, which was the target device of this work. The UHD Graphics 630 is included in the Gen 9.5 architecture line thoroughly analyzed in Chapter 2. The relevant specifications of the hardware are present in Table 4.1. Additionally, the system's main memory is an 8 GB DRAM, and the shared LLC has a capacity of 8 MB. Both the DRAM and the LLC are shared between CPU and iGPU.

Table 4.1: Target device specifications

Parameter	Description
Host CPU	Core i5-8300H
Target iGPU	UHD Graphics 630
Slices	1
Subslices	3
EUs	23
HW Threads per EU	7
SIMD FPUs per EU	2
Base Frequency	350 MHz
Maximum Frequency	1000 MHz
L3 Data Cache	512 KB
L3 Cacheline	64 B
SLM (per subslice)	64 KB

The target iGPU contains 24 EUs, although one of them is reserved for die recovery purposes, leaving it unavailable for computation purposes. Consequently, one of the subslices has only 7 EUs for the purpose. The remaining subslices have all 8 EUs fully operational for the computation duties. This particularity can cause some imbalance in the workload distribution inside the iGPU. However, it is accounted for during the tests through adjusted measurements of the iGPU's capabilities.

4.2 Intel iGPU Throughput Characterization

To properly analyze the throughput capabilities, the kernels presented in Listings 3.4 for addition, and 3.5 for MAD were developed. These tackle the regular one-cycle arithmetic operations and the special case of multiplication and addition in one operation, respectively. However, before diving into the achieved results, it is crucial to identify the theoretical peak throughput of the target iGPU. Assuming 32-bit single-precision floating-point (or simply float) is the used data type, it is given by

$$\text{Max ADD FLOPS} = \text{ADD} * \text{SIMD4} * 2\ \text{FPU} * \text{EU} * \text{Frequency} \tag{4.1}$$

or by

$$\text{Max MAD FLOPS} = \text{MAD (2)} * \text{SIMD4} * 2\ \text{FPU} * \text{EU} * \text{Frequency}, \tag{4.2}$$

depending on the executed operation. Since the FPUs in every EU are physically composed of 4 SIMD lanes, each FPU can produce 4 operations per clock cycle. Coupled with the second FPU, each EU can provide up to 8 instructions per cycle. In the target iGPU, specifically, the 23 EUs, coupled with a maximum frequency of 1000 MHz (or 1 GHz), support up to 4 * 2 * 23 * 1 GHz = 184 GFLOPS, for arithmetic operations such as addition or subtraction. Since a MAD instruction achieves both addition and multiplication in one clock cycle, technically the achieved throughput is exactly the double the previous result, standing at 368 GFLOPS for MAD. However, if the provided data is now composed of double-precision floating-point (or double) elements, the required cycles to produce the same number of operations relative to the float data type are cut to half, due to its doubled size of 64 bits per element. Additionally, the compute architecture of Gen 9.5 iGPUs only designed one FPU to be capable of supporting double-precision operations and other advanced math functions. As such, the maximum throughput for the double data type given by flooding the iGPU with ADD kernels only reaches up to 184 GFLOPS / 4 = 46 GFLOPS. For MAD operations, the peak is obtained under similar circumstances, 368 GFLOPS / 4 = 92 GFLOPS. Table 4.2 illustrates the maximum theoretical throughput, depending on the operation and chosen data type.

Table 4.2: Theoretical Maximum Throughput of the Experimental Setup

Operation	Data Type	FPUs	Throughput [GFLOPS]
Addition	Float	2	184.0
Addition	Int	2	184.0
Addition	Double	1	46.0
MAD	Float	2	368.0
MAD	Double	1	92.0

The GPU performance was evaluated for three different data types: 32-bit integer, 32-bit float, and 64-bit double, for scalar and vectorized types. In addition operations, integer, and float data types were expected to behave similarly close to the discussed values, as no information in the architecture arguments otherwise. The double data type was expected to achieve one-fourth of the others' throughput. MAD operations are only supported in floating-point operations, so only the latter two have their results presented below. The throughput plots are varied on the horizontal axis by two metrics. The first one is the gradually increasing amount of defined OpenCL work-groups ranging from 1 to 512, having each work-group containing 32 work-items as recommended by OpenCL on the target machine. This essentially results in a data size increase. The second one maintains constant data size but varies in the organization of work-group divisions, by gradually increasing work-items per work-group, from 1 to 256, but similarly reducing work-groups to retain the original data size. The plots derived from the second approach are displayed in a logarithmic axis to better highlight the transitions between key values. Figures 4.1 and 4.2 reflect the variation in work-groups and the variation in work-group size, respectively, for single-precision floating-point ADD operations, while Figure 4.3 illustrates the obtained CARM.

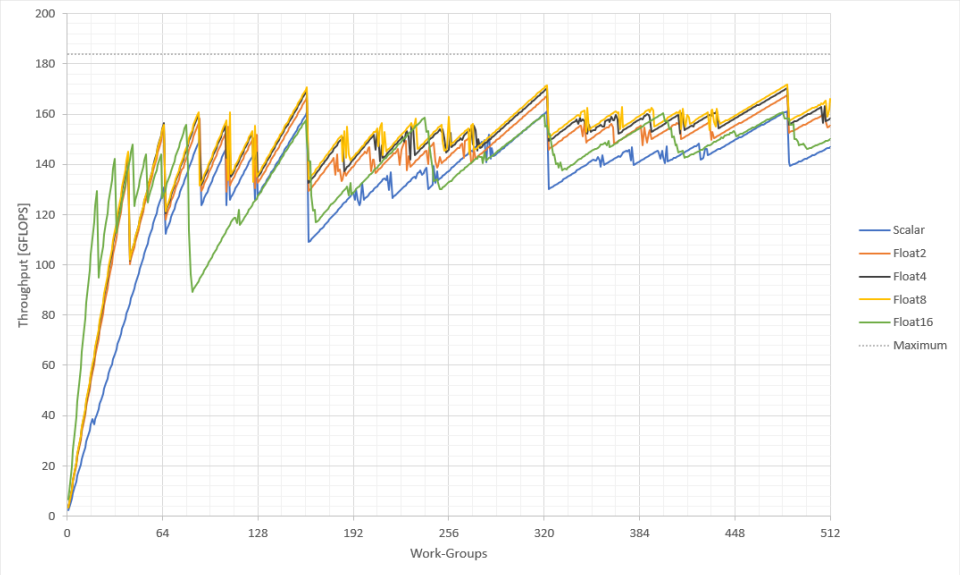

Figure 4.1: Float ADD Operation Throughput with increasing total Work-Groups

The maximum throughput achieved in this setup was 172 GFLOPS out of 184 GFLOPS, approximately 93.5% of the iGPU's theoretical peak. From both Figures 4.1 and 4.2, the best results come from vectorized data types Float4 and Float8 and the worst from Scalar and Float16. The plot continuously displays a gradual increase and sudden decrease in performance throughout the entire test. The oscillating nature of the plot is tied to the workload imbalance among the EUs in different subslices. This occurrence is especially aggravated by the imbalance in the number of EUs in the target setup, which changes how the best performance is achieved. The number of work-groups for which the ADD kernel

was executed on the iGPU reached the peak value of 172 GFLOPS are 161, 322, and 483 work-groups. All of these are multiples of 161, which is, not coincidentally, the maximum amount of hardware threads that this iGPU can have at the same time stored in all combined EUs. In the Gen 9.5 iGPU microarchitecture, each EU can have up to 7 hardware threads with a kernel ready to execute, from which the Thread Arbiter (TA) selects the one to insert in the recently freed up FPU. It means that full occupation of the threads is achievable by deploying 7 * 23 = 161 work-groups in them, and the same logic applies to the multiples of 161 work-groups. For a setup with 24 available EUs, the maximum performance would be achieved at 7 * 24 = 168 work-groups and multiples instead.

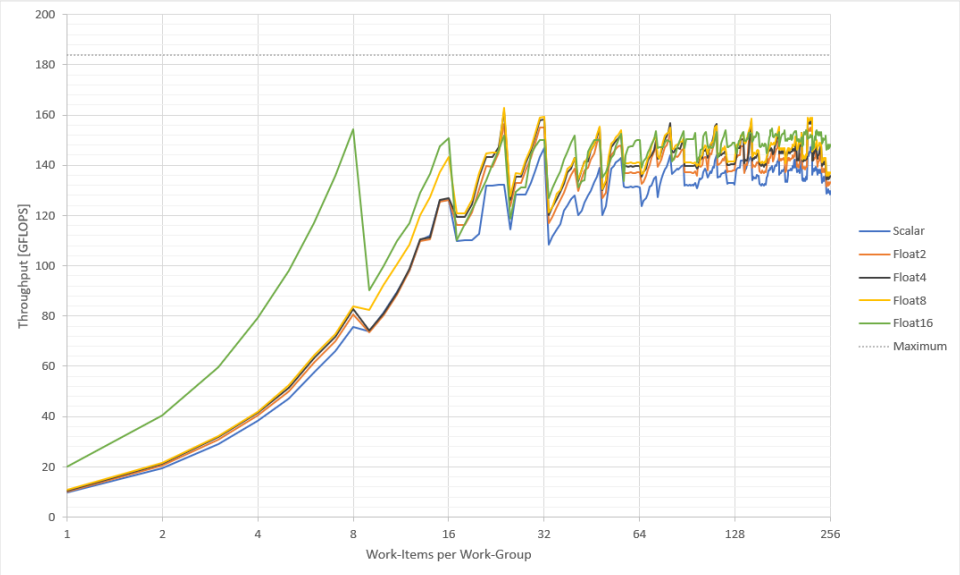

Figure 4.2: Float ADD Operation Throughput with increasing Work-Items per Work-Group

From analyzing the variation in work-group size in Figure 4.2, the minimum value where all vectorization data types come close to the achieved practical peak is at 32 work-items per work-group which reinforces the OpenCL and Intel recommendations related to minimum work-group sizes. For lower values, all of them naturally perform worse due to the lack of saturation which results in stalls and very poor latency hiding by the EUs. However, the more vectorized elements need fewer work-items per work-group to reach higher throughput values, since they naturally require fewer elements to saturate the hardware, considering each one requires more operations than a scalar or a lower vector size data type.

The CARM analysis result for the microbenchmark containing the addition kernel on 3.4, presented in Figure 4.3 illustrates five dots, one for each vectorization level. All of them are on the same horizontal axis due to having the exact same amount of operations between them and requiring from and transferring to memory buffer objects of the same size. The colored section depicts the ceiling that the specific kernel should achieve, given its purpose. The ceilings are calculated by Intel Advisor, on the

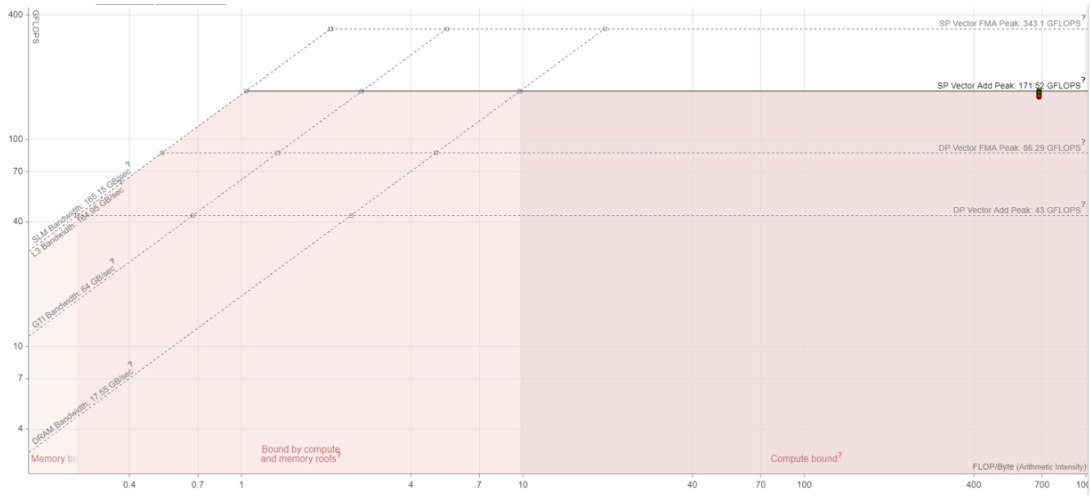

Figure 4.3: Float ADD Operation CARM

target iGPU. All the kernels achieve values from 169 to 172 GFLOPS which fall in line with the peak reached by the software. A possible explanation for the distance between this range of values and the theoretical maximum is that the setup iGPU can take a slight hit to performance due to having it process graphics and applications such as Intel Advisor on top of the set of microbenchmarks.

The integer data type occupies the same size as the float data type and makes use of the same FPUs, so in theory, the plots obtained should be similar to the ones pictured before. Figures 4.4 and 4.5 present the variation in throughput with dependence on the number of work-groups and work-group size, respectively. Figure 4.6 illustrates the CARM for integer ADD operations.

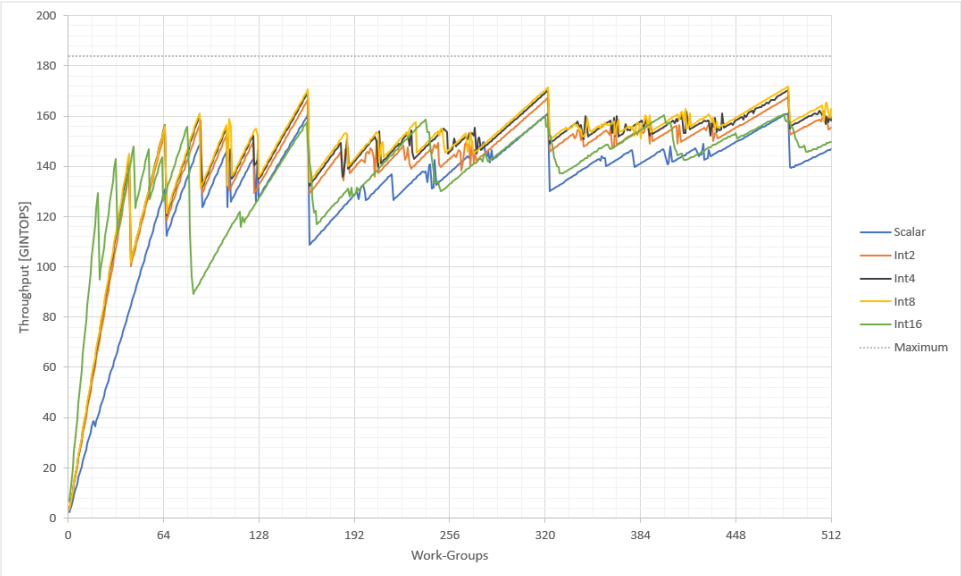

Figure 4.4: Int ADD Operation Throughput with increasing total Work-Groups

The plot featured in Figure 4.4 is almost identical to its float counterpart, having the best results for multiples of 161 work-groups, at about 171/172 GINTOPS. Additionally, Int4 and Int8 perform the best, while Scalar and Int16 drop to lower values relatively, achieving around 160 GINTOPS at most.

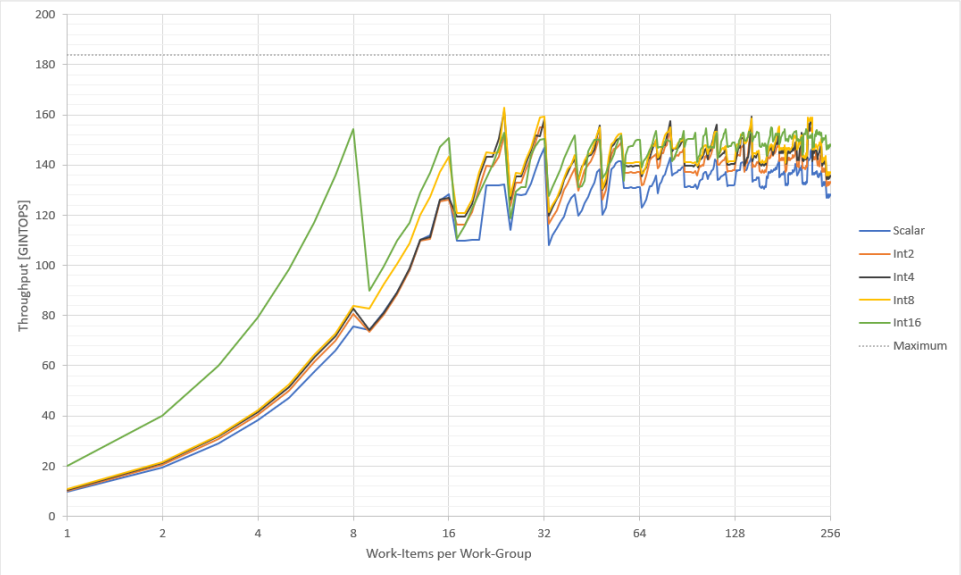

Figure 4.5: Int ADD Operation Throughput with increasing Work-Items per Work-Group

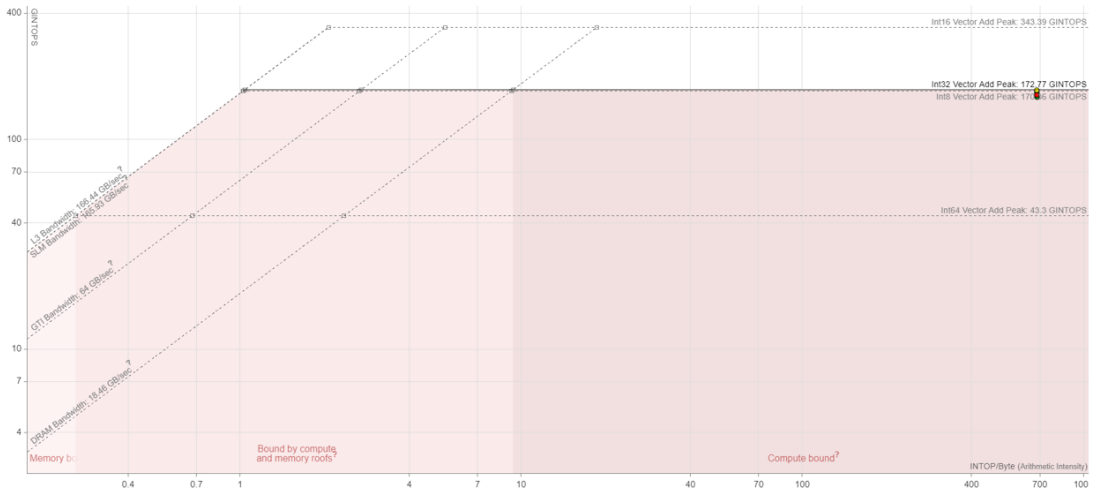

Figure 4.6: Int ADD Operation CARM

The variation in work-group size in Figure 4.5 also depicts the same picture, as 32 work-items are the minimum for decent performance levels for every vectorized data type. The faster convergence to the peak values in Int8 and Int16 is also noticeable for 8 and 16 work-items per work-group. The provided CARM in Figure 4.6 has a similar structure to the float one, as the same maximum throughput is achieved. Just like Figure 4.3, the kernels are situated far to the right side, indicating both are heavily

compute-bound, which is expected, given the high number of additions executed on the same elements, per memory data transfer. The top roof, labeled Int16 Vector Add Peak, is achievable in the architecture with 16-bit Integers, such as the short data type, matching the throughput peak of a float MAD operation kernel.

For the double data type, having only one FPU available for computation places the computation boundary at a fourth of the single-precision kernels. Figures 4.7 and 4.8 show the performance changes with increasing work-groups and increasing work-group size, respectively, and Figure 4.9 provides the Intel Advisor generated CARM.

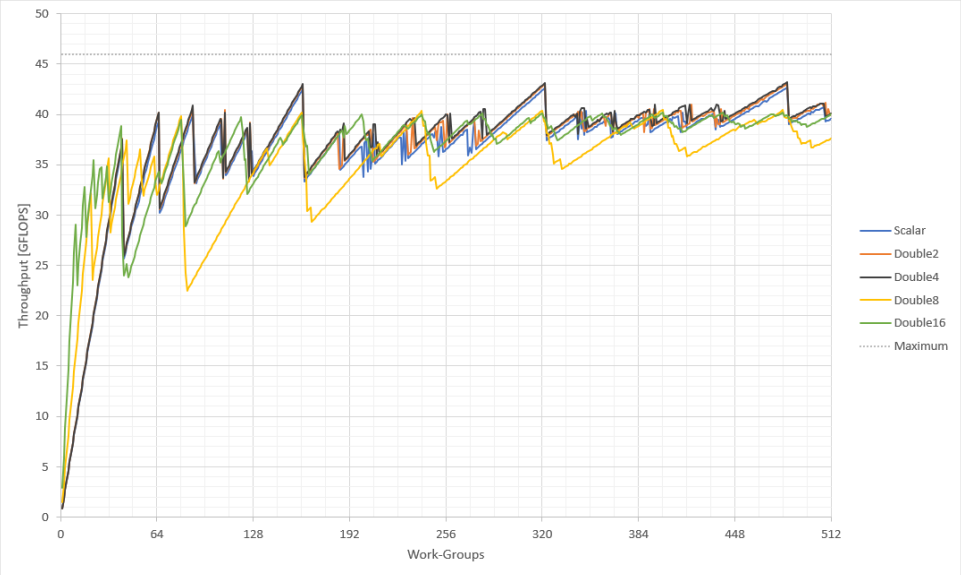

Figure 4.7: Double ADD Operation Throughput with increasing total Work-Groups

The maximum achieved peak, in Figure 4.7, with double-precision floating-point buffer objects is 43.0 GFLOPS out of 46.0 GFLOPS, which represents 93.5% of the theoretical total. Compared to the float ADD microbenchmark, Scalar, Float2, and Float4 now provide the best results, while Float8 and Float16 are well below. The difference, in this case, is that the restriction of usage to a single FPU per EU now implies that saturation of the EU with the workload occurs faster. Hence, the amount of work-groups needed for the lower vectorized kernels to achieve decent computation values is much lower than in single-precision. Also, considering each element now has a 64-bit size, higher vectorized kernels struggle with register size restrictions, compromising the number of operations they output.

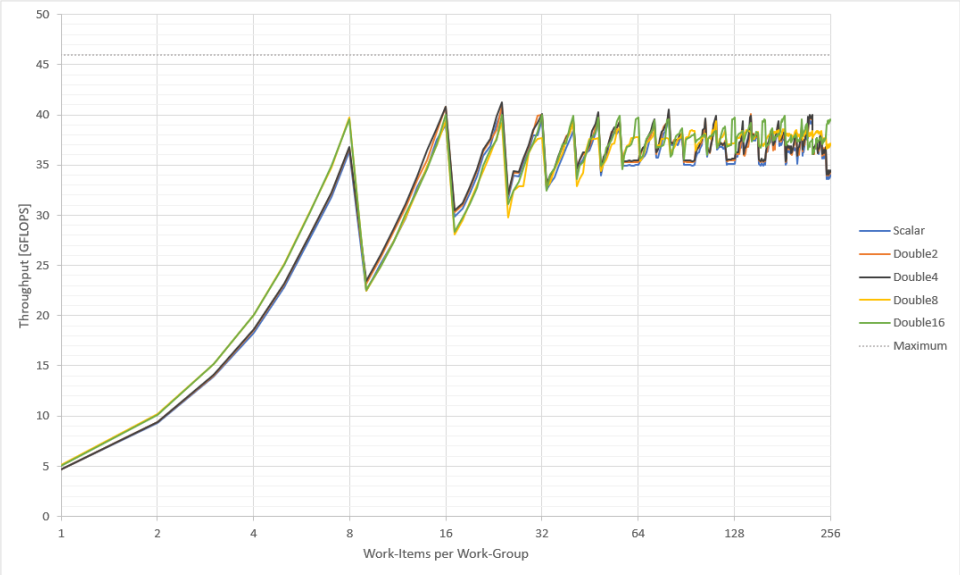

Figure 4.8: Double ADD Operation Throughput with increasing Work-Items per Work-Group

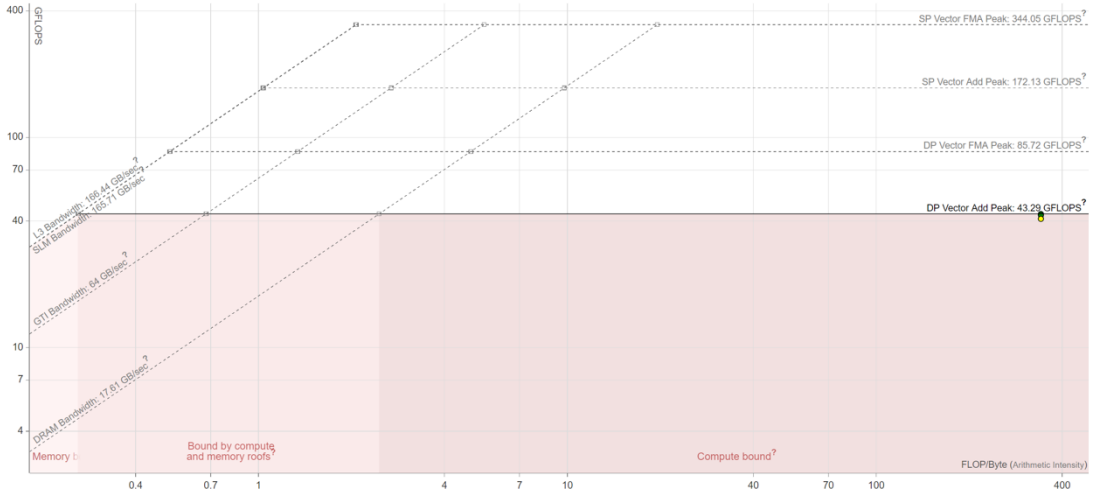

Figure 4.9: Double ADD Operation CARM

In Figure 4.8, as discussed previously, there are barely any differences in the number of work-items needed to achieve values close to the computation upper bound among vectorized data types, due to the restrained usage of the FPUs. Apart from that, all the kernels behave similarly, indicating no significant discrepancies in throughput for different granularity of work-items in work-groups. As a way to confirm values obtained in Figure 4.7, the CARM in Figure 4.9 achieves values in the range of the 42/43 GFLOPS, in the compute-bound section of the Roofline, resulting in a quarter of the throughput of single-precision and 1/8 of the achievable by the iGPU. MAD operations were used in two kernels, for single and double-precision floating-point. Identical plots were traced for both, present for single-

precision in Figure 4.10 with increasing work-groups and Figure 4.11 with increasing work-items per work-group. The CARM obtained in Intel Advisor is illustrated in 4.12.

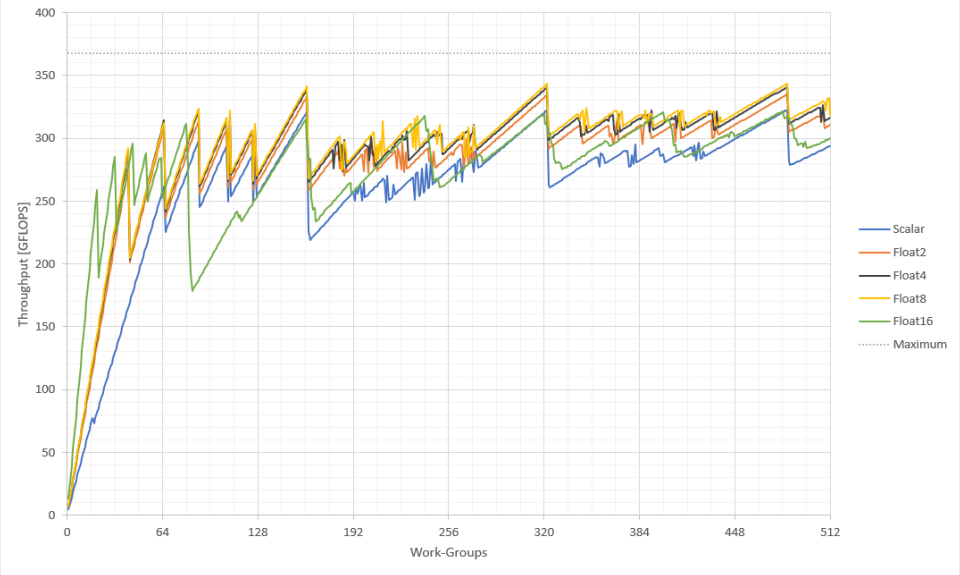

Figure 4.10: Float MAD Operation Throughput with increasing total Work-Groups

In Figure 4.10, all performance curves are equivalent to the ones in the addition kernel, but with double the obtained throughput, as expected. Float4 and Float8 are the best performers, and Scalar and Float16 are the worst ones.

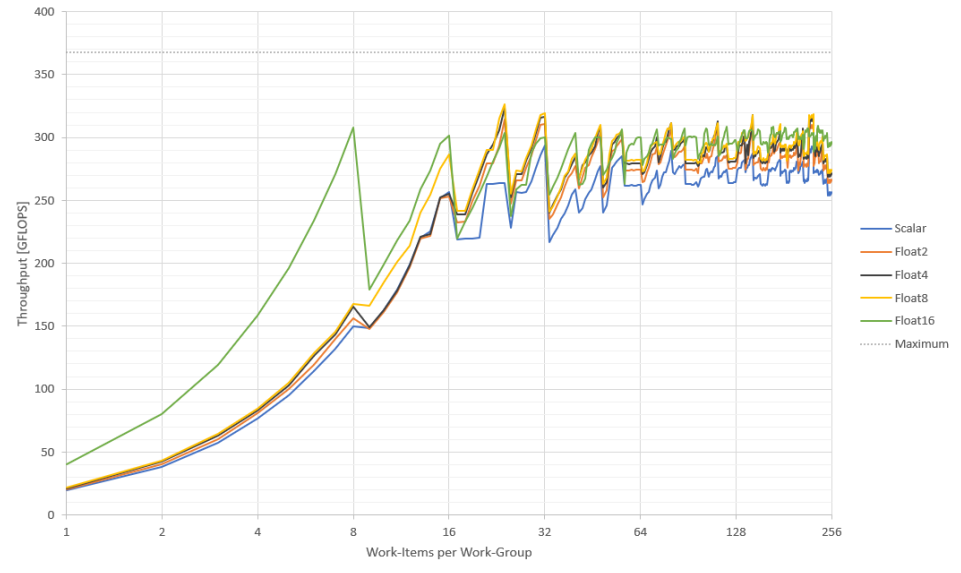

Figure 4.11: Float MAD Operation Throughput with increasing Work-Items per Work-Group

Again, in Figure 4.11, the behavior of each kernel falls in line with the previous ones, as Float8 and

Float16 scale faster for fewer work-items and all vector types are close in computation performance, apart from Scalar which falls a bit below the others. The CARM in Figure 4.12 confirms the achieved data in Figure 4.10, reaching close to the documented maximum for the architecture of the target iGPU, at 345 GFLOPS out of 368 GFLOPS, for 93.8% of the total value.

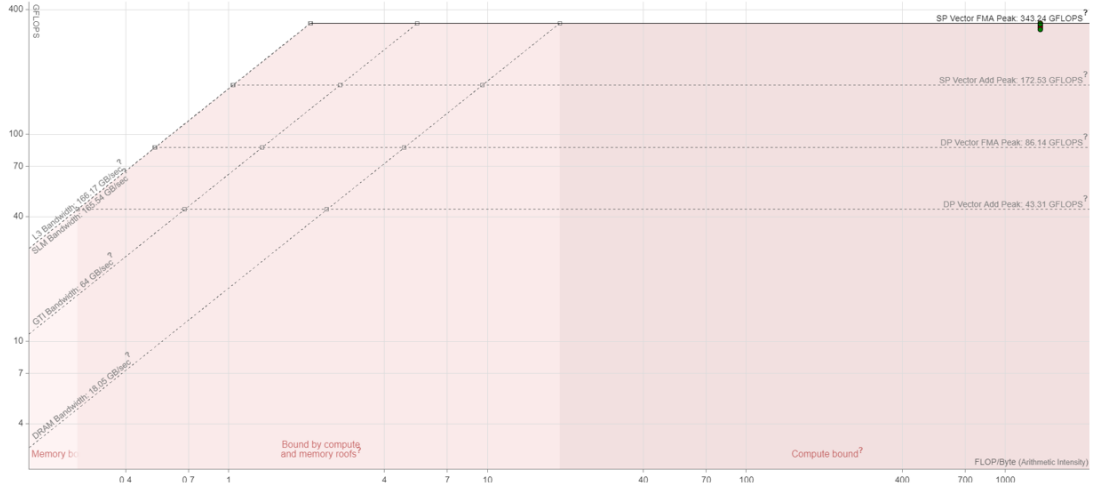

Figure 4.12: Float MAD Operation CARM

The MAD operation was also inserted in a double-precision kernel, of which Figure 4.13 depicts the variation in work-groups and Figure 4.14 shows the variation of its work-items. The respective produced CARM is presented in Figure 4.15.

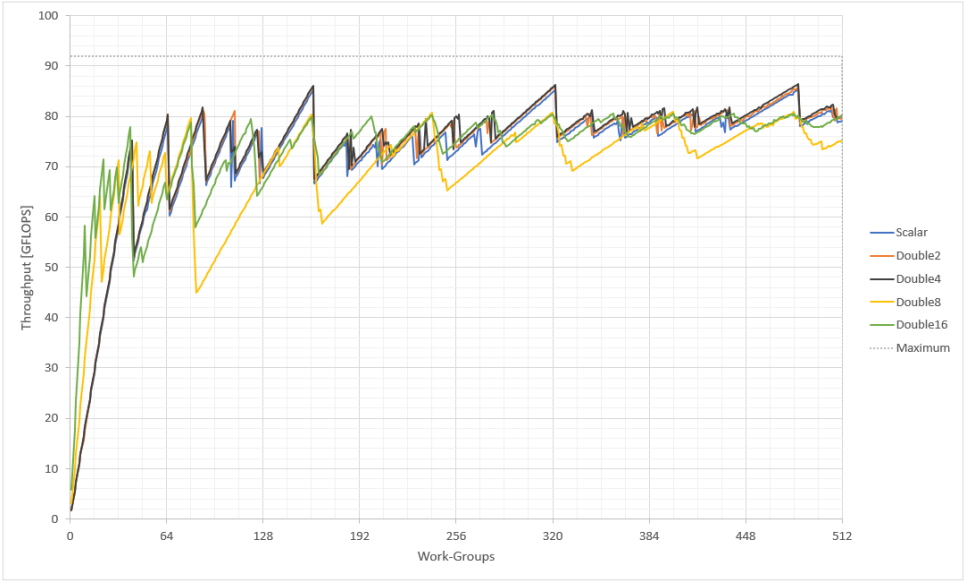

Figure 4.13: Double MAD Operation Throughput with increasing total Work-Groups

In Figure 4.13, the maximum achieved by the double-precision kernel is 86.4 GFLOPS out of 92.0

GFLOPS, covering 93.9% of the GPU's capabilities when it comes to computation power. Identically to the double-precision addition kernel, Scalar and lower vectorized data types are the best performers. On the other hand, Double8 and Double16 only reach about 80 GFLOPS at their best point.

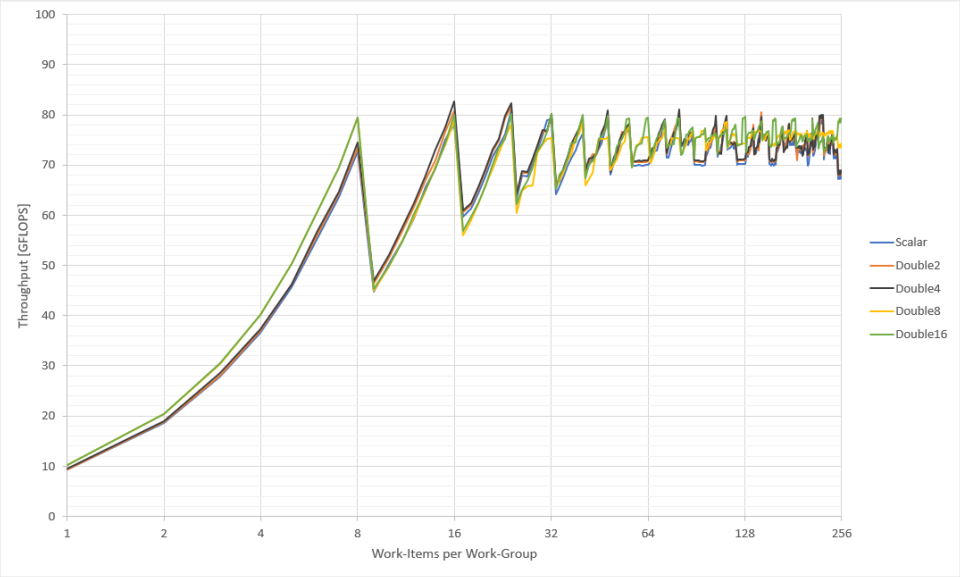

Figure 4.14: Double MAD Operation Throughput with increasing Work-Items per Work-Group

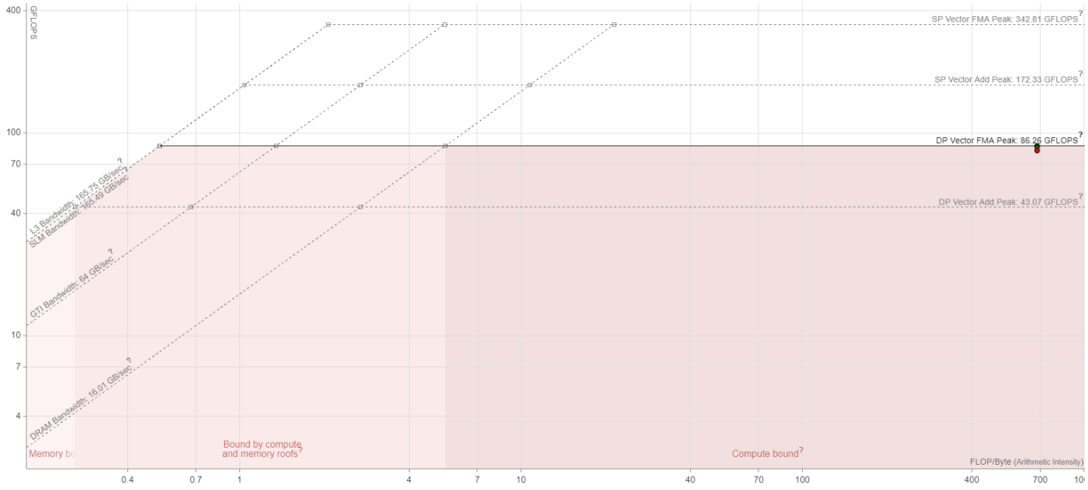

Figure 4.15: Double MAD Operation CARM

Figure 4.14 shares similarity with the addition kernel plot in Figure 4.8, as all five performance curves follow the same trajectory over the course of the horizontal axis through the increase of work-groups. The kernel reaches, as confirmed in Figure 4.15, around 86 GFLOPS, which is double the value of the addition kernel but a fourth of the single-precision MAD kernel, hitting the corresponding rooftop.

4.3 Intel iGPU Memory Bandwidth and Latency Characterization

The memory bandwidth of the iGPU is a vital metric, as it can frequently be the major bottleneck of an application to be run on a GPU. This is especially the case in dedicated GPUs, but in the end, it is very dependent on the software and on the operations that require execution. The studied iGPU has a dedicated L3 cache, which contains a separated portion for the SLM, a shared LLC with the CPU, and the system main memory DRAM, also shared with the main processor. The latter two require data to traverse the Ring Interconnect of the architecture, to go from iGPU to LLC/DRAM, or vice-versa. Therefore, if these requests are a significant portion of the workload, the achieved bandwidth by the iGPU is distant from its potential maximum bandwidth. Supposing all required data is in the L3 Cache, located inside the iGPU, having each subslice with 64 B read/write bandwidth per cycle on the data port. Given the number of subslices and the frequency of the iGPU clock, the maximum bandwidth is given by

$$\text{Max Bandwidth} = \text{Read/Write Bandwidth} * \text{Subslices} * \text{Frequency}, \tag{4.3}$$

which on this setup achieves 64 B * 3 Subslices * 1 GHz = 192 GB/s.

Each EU has one Send unit, and each GRF register in an EU can contain up to 32 B of data. In theory, if multiple EUs have send instruction requests ready to be executed, to fetch values from the cache, during the whole execution time of the kernel, the maximum bandwidth would be achieved. The kernel in Listing 3.6 intends to achieve that, by having two large floating-point buffers, one to load an element from and one to send the same element to. Repeating this instruction for several iterations forces mass usage of the L3 cache, to try to mitigate the latency of any necessary DRAM accesses. This microbenchmark, similarly to the throughput performance characterization ones, varies the number of OpenCL work-groups and their size, in Figures 4.16 and 4.18, respectively.

Contrary to what was observed in the throughput performance characterization, changes in the vectorization level of the chosen data type have a much more drastic effect. The best bandwidth was achieved by the single-precision Float2 and Float4 kernels, at approximately 178 GB/s and 181 GB/s, for a relative bandwidth of 92.7% and 94.3%, respectively. The Scalar kernel sees a significant drop-off, to a maximum of 120 GB/s, meanwhile, Float8 and Float16 reach no more than 90 GB/s and 45 GB/s. The GEN Assembly code [26] for Float4 and Float8 is compared in Figure 4.17. 'Send' instructions are the loads and 'sends' are the stores. The first register is the destination one and the other registers are sources. The values in front of the instruction code indicate the execution size and the execution offset, respectively. From analyzing the GEN Assembly code behind the generated kernels, Scalar, Float2, and Float4 are identical in instruction count and order, varying only the way the registers are addressed and which ones are addressed in each instruction. Each load in the kernel is compiled into two send

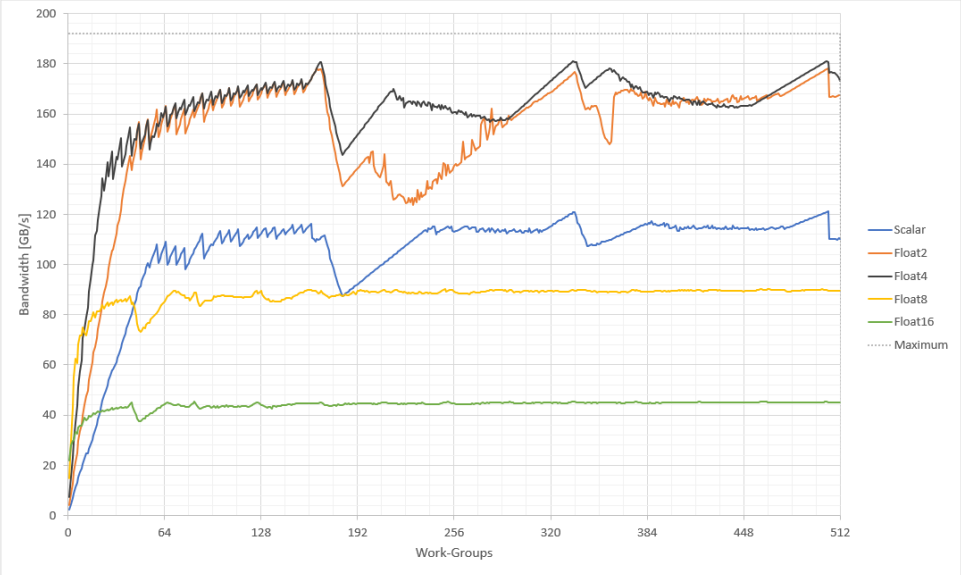

Figure 4.16: Float Load/Store Operation Bandwidth with increasing total Work-Groups

instructions, executed in SIMD16 of 16-bit words (represented by a 'w'), with the second one offset by 16 words, totaling up to the maximum of 64 B. However, in Float8 and Float16, each load is compiled into SIMD8 instructions of 32-bit floats (represented by an 'f'), without offset, so each send instruction is a new memory request. Therefore, Float8 only makes use of half of the maximum bandwidth. Float16 requires double the number of elements, requiring four instead of two send instructions, of which each one is a new memory request, consequently making use of only a quarter of the available bandwidth. This falls in line with obtained results as 90 GB/s and 45 GB/s are, respectively, half and one-fourth of the attained maximum of 181 GB/s.

```
LABEL216
        send (16|M0)              r10:w    r116    0xC         0x4805001  //  wr:
        send (16|M16)             r18:w    r118    0xC         0x4805001  //  wr:
(W)     add (1|M0)                r8.0<1>:d     r8.0<0;1,0>:d     16:w
(W)     cmp (16|M0)   (lt)f0.0    null<1>:d     r8.0<0;1,0>:ud    0x800:uw
(W)     cmp (16|M16)  (lt)f0.0    null<1>:d     r8.0<0;1,0>:ud    0x800:uw
        sends (16|M0)             null:w   r5      r10     0x20C       0x4025000
        sends (16|M16)            null:w   r114    r18     0x20C       0x4025000
        send (16|M0)              r26:w    r116    0xC         0x4805001  //  wr:
        send (16|M16)             r34:w    r118    0xC         0x4805001  //  wr:
        sends (16|M0)             null:w   r5      r26     0x20C       0x4025000
        sends (16|M16)            null:w   r114    r34     0x20C       0x4025000
        send (16|M0)              r42:w    r116    0xC         0x4805001  //  wr:
        send (16|M16)             r50:w    r118    0xC         0x4805001  //  wr:
        sends (16|M0)             null:w   r5      r42     0x20C       0x4025000
        sends (16|M16)            null:w   r114    r50     0x20C       0x4025000
        send (16|M0)              r58:w    r116    0xC         0x4805001  //  wr:
        send (16|M16)             r66:w    r118    0xC         0x4805001  //  wr:
        sends (16|M0)             null:w   r5      r58     0x20C       0x4025000
        sends (16|M16)            null:w   r114    r66     0x20C       0x4025000
        send (16|M0)              r74:w    r116    0xC         0x4805001  //  wr:
        send (16|M16)             r82:w    r118    0xC         0x4805001  //  wr:
        sends (16|M0)             null:w   r5      r74     0x20C       0x4025000
        sends (16|M16)            null:w   r114    r82     0x20C       0x4025000
```

```
(W)     cmp (8|M0)    (lt)f0.0    null<1>:ud    r5.0<0;1,0>:ud    0x400:uw
        sends (8|M0)              null:ud  r118    r7      0x10C       0x2026000
        sends (8|M0)              null:ud  r3      r11     0x10C       0x2026000
        send (8|M0)               r17:f    r119    0xC         0x2406001  //  wr:
        send (8|M0)               r21:f    r15     0xC         0x2406001  //  wr:
        add (8|M0)                r15.0<1>:ud   r119.0<8;8,1>:ud  0x10:uw
        sends (8|M0)              null:ud  r118    r17     0x10C       0x2026000
        sends (8|M0)              null:ud  r16     r21     0x10C       0x2026000
        add (8|M0)                r16.0<1>:ud   r118.0<8;8,1>:ud  0x10:uw
        send (8|M0)               r27:f    r119    0xC         0x2406001  //  wr:
        send (8|M0)               r31:f    r25     0xC         0x2406001  //  wr:
        add (8|M0)                r25.0<1>:ud   r119.0<8;8,1>:ud  0x10:uw
        sends (8|M0)              null:ud  r118    r27     0x10C       0x2026000
        sends (8|M0)              null:ud  r26     r31     0x10C       0x2026000
        add (8|M0)                r26.0<1>:ud   r118.0<8;8,1>:ud  0x10:uw
        send (8|M0)               r37:f    r119    0xC         0x2406001  //  wr:
        send (8|M0)               r41:f    r35     0xC         0x2406001  //  wr:
        add (8|M0)                r35.0<1>:ud   r119.0<8;8,1>:ud  0x10:uw
        sends (8|M0)              null:ud  r118    r37     0x10C       0x2026000
        sends (8|M0)              null:ud  r36     r41     0x10C       0x2026000
        add (8|M0)                r36.0<1>:ud   r118.0<8;8,1>:ud  0x10:uw
        send (8|M0)               r47:f    r119    0xC         0x2406001  //  wr:
        send (8|M0)               r51:f    r45     0xC         0x2406001  //  wr:
        add (8|M0)                r45.0<1>:ud   r119.0<8;8,1>:ud  0x10:uw
        sends (8|M0)              null:ud  r118    r47     0x10C       0x2026000
        sends (8|M0)              null:ud  r46     r51     0x10C       0x2026000
```

Figure 4.17: Comparison of GEN Assembly code for Float4 (left) and Float8 (right) Load/Store Bandwidth Kernels

Bandwidth shows large discrepancies between benchmarks run with distinct amounts of work-items per group. Analysis of Figure 4.18 affirms this statement, as Float2 and Float4 kernels show the most

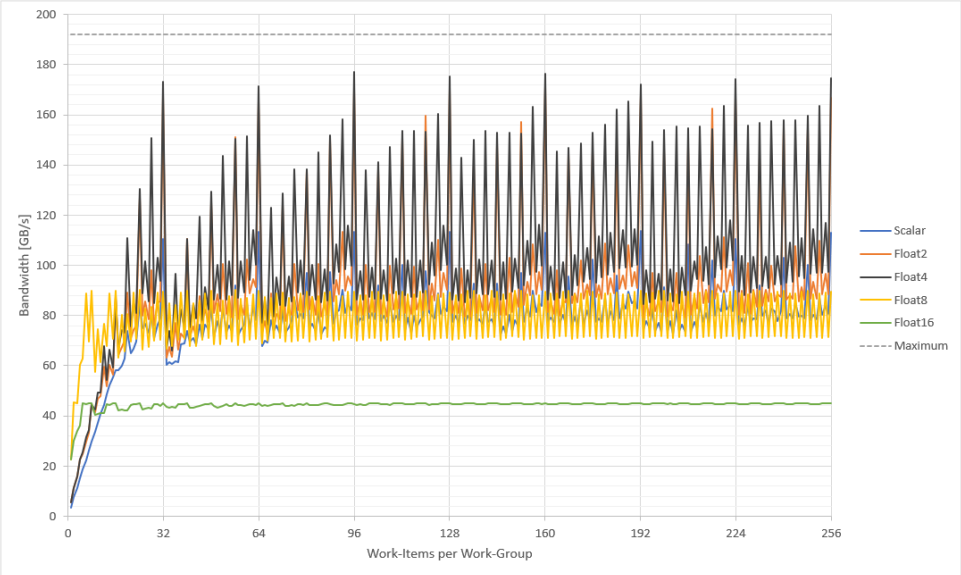

Figure 4.18: Float Load/Store Operation Bandwidth with increasing Work-Items per Work-Group

oscillation between achieving close to the 180 GB/s value discussed in Figure 4.16 and half of that. Purely from observation of the plot, work-items that are multiples of 32 are the only quantities that can reach the highest values. The same kernel detailed in Listing 3.6 was employed to buffers of the 64-bit, double-precision floating-point data type, with its results present in Figures 4.19 and 4.20, for an increasing number of work-groups and an increasing number of work-items in a group, respectively.

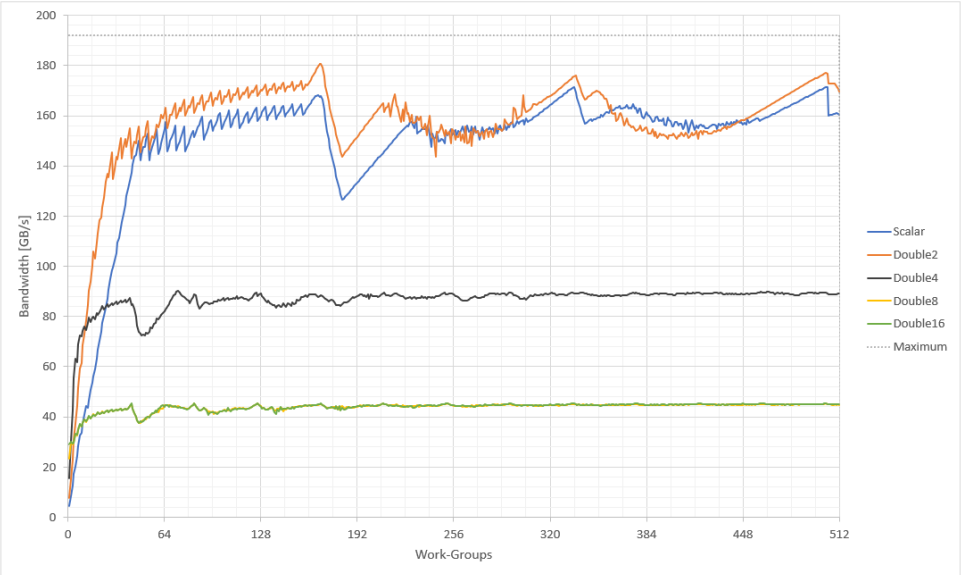

Figure 4.19: Double Load/Store Operation Bandwidth with increasing total Work-Groups

In the plot presented in Figure 4.19, the vectorized data types switch places, as the best memory

bandwidth is obtained in Scalar and Double2 data types, while Double4 now equals the performance of Float8 in 4.16. Double8 and Double16 have the same bandwidth at about a quarter of the theoretical upper bound. Given that the double data type contains 64-bit elements instead of 32-bit, understandably, the behavior explained previously shifts to lower vectorization levels.

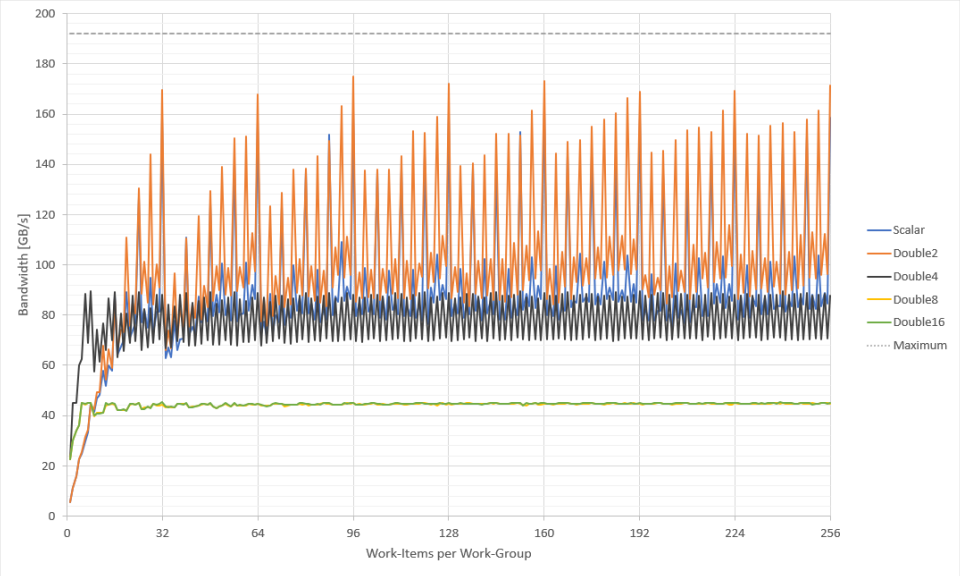

Figure 4.20: Double Load/Store Operation Bandwidth with increasing Work-Items per Work-Group

In Figure 4.20, the same behavior as in Figure 4.18 is observed here. Significant swings in bandwidth are present for the vectorized typed achieving higher bandwidth, with multiples of 32 work-items per work-group reaching the maximum of the performance curves in the plot, for Scalar and Double2. Double8 and Double16 show the least amount of variance, consistently reaching around 45 GB/s, regardless of the number of work-items picked.

To evaluate the entire system memory hierarchy, a test to attempt to attain maximum bandwidth is not enough, as this only reflects the effects of the L3 Cache on the iGPU. The kernel in Listing 3.7 uses a scalar, variable-sized, floating-point single or double-precision array with a stride of 16, which is the same as the number of float elements that fit into a cacheline. Afterward, it uses a pointer chasing algorithm, where each individual work-item will start on its own element and stride to a new cacheline, requesting it from memory, for several iterations. The theory behind argues that for small-sized arrays, the entire array is able to fit inside the L3 cache, therefore, regardless of the distinct access patterns, nothing should change on the bandwidth field. However, as soon as the array extends the size of the L3 Cache, the bandwidth should drop closer to the GTI bandwidth of 64 GB/s. Gradually increasing the size would result in a similar drop for DRAM whenever the array size surpasses the LLC size. For this microbenchmark, 4096 work-groups of 32 work-items were used. The results of the tests are presented in Figure 4.21.

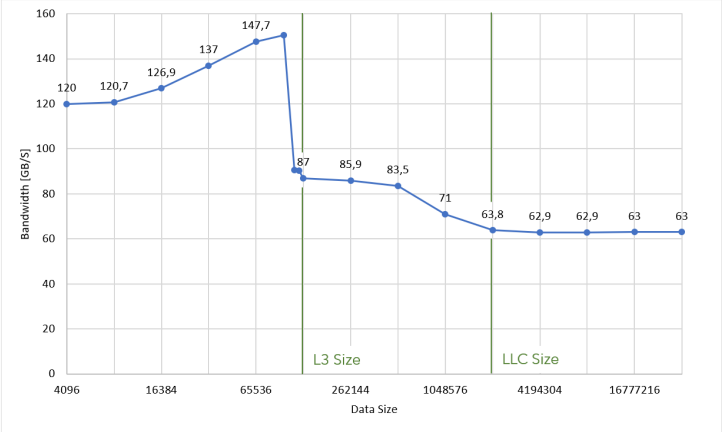

Figure 4.21: Float Pointer-Chasing Algorithm for Bandwidth with increasing Data-Size

In previously shown data, kernels on scalar float arrays were picking up around 120 GB/s. In the test displayed in Figure 4.21, it increases up to 148 GB/s for the size of 65536 elements, which is exactly half of the L3 cache size. In the next iteration, at 131072, the full capacity of the L3 cache is not enough to fit in the entire array, most likely due to having to store other unrelated variables, forcing the iGPU to transfer data from the LLC, and dropping the resulting bandwidth to 87 GB/s. Eventually, the exponential increase of the workload size reduces the bandwidth to 64 GB/s, vaguely matching with the theoretical value. However, after surpassing the LLC capacity, the bandwidth stays constant, indicating that it still manages to make use of previous memory systems to some degree of success, avoiding being fully reliant on data transfers from and to the DRAM, which would have resulted in a bandwidth of around 16 GB/s, according to the documented values.

Memory latency was also measured through a similar pointer-chasing kernel. However, it was single-threaded, meaning only one work-group with one work-item would execute the kernel. The stride 16 on the array would force fetching new cachelines in every cycle, and the average access time was measured by dividing the OpenCL event profiling timer by the number of accesses made. Figure 4.22 displays the attained data.

The obtained plot in Figure 4.22 shows a staircase shape, which is to be expected in a latency microbenchmark such as this one since all three memory blocks' are highlighted by constant access times. Additionally, a steep increase in access times happens when the array data size increases to a point where it no longer fits in the previous memory block, thus having to order requests from the following one, taking a progressively longer time to fetch the data. These results also match the ones obtained from paper [24].

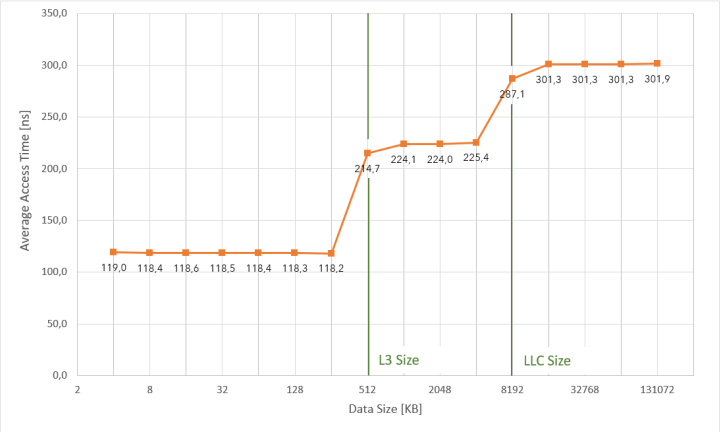

Figure 4.22: Float Pointer-Chasing Algorithm for Latency with increasing Data-Size

4.4 Intel iGPU Power and Energy Efficiency Characterization

Both measurements of power consumption and energy efficiency were derived from energy expenditure metrics, collected through PAPI which provides data collections of the RAPL interface. Power requirements were calculated through the division of the energy expenditure the kernel execution time. On the other hand, energy efficiency was calculated by measuring the amount of executed operations or transferred bytes, divided by the amount of energy spent during the kernel execution. These readings were measured for several operations, but due to their similarity, only floating-point MAD operation kernel presented in Listing 3.5 and floating-point bandwidth measurement presented in Listing 3.6 are displayed. Figures 4.23 and 4.24 depict power requirement and energy efficiency reading for the floating-point MAD kernel, while Figures 4.25 and 4.26 do the same for the floating-point memory bandwidth kernel.

The power kernel for the MAD operation appears to be comparable between different vectorization levels, all inserted in the range from 8 W to 10 W readings, apart from oscillations when the workload saturation is low. It is also noticeable in Figure 4.23 that the requirement in terms of power is slightly higher for less vectorized kernels, with Scalar requiring the most and Float16 requiring the least. This occurrence proved to be constant among all throughput kernels tested for power requirements. The energy efficiency readings in Figure 4.24 reflect parts of both the throughput characterization plot and the power readings plot, as it assimilates itself to the first one. However, it distances the performance curves of the different vectorization levels due to their energy expenditure. Scalar and Float2 were the kernels that required the most power, and as such, decrease in energy efficiency relative to Float4, Float8, and Float16, all of which are more energy efficient.

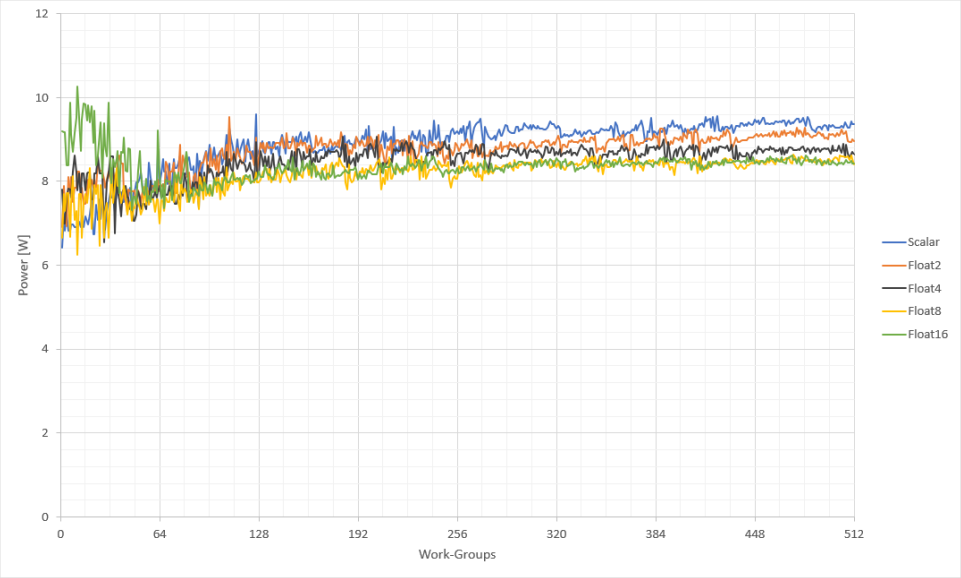

Figure 4.23: Float MAD Operation Power with increasing total Work-Groups

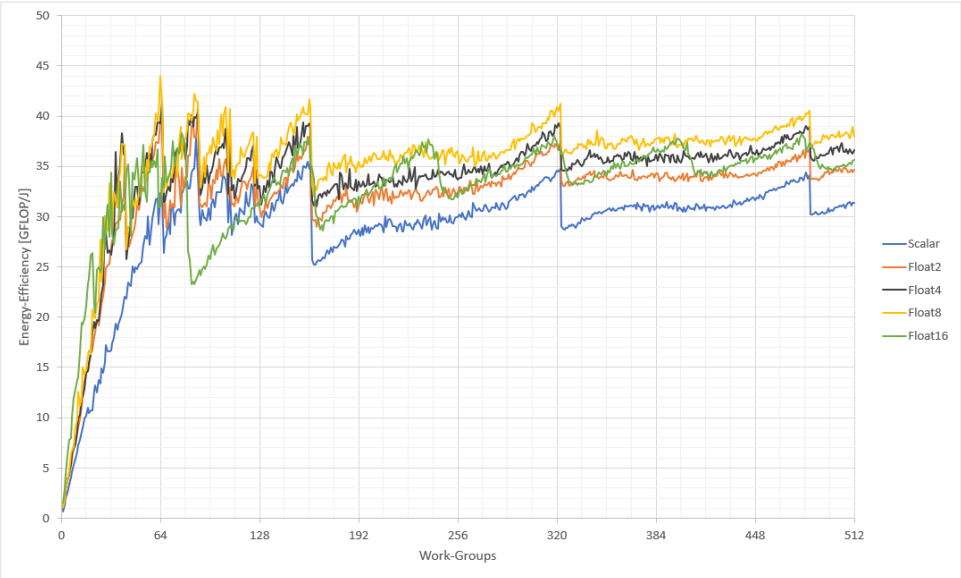

Figure 4.24: Float MAD Operation Energy Efficiency with increasing total Work-Groups

Alike in memory bandwidth measurements, the power readings displayed in Figure 4.25 are much more distanced between different vectorized kernels. Scalar and Float2 reach up to 10 W of power while Float16 only requires 8 W. Higher vectorization levels prove again to require less power than their counterparts. The results in Figure 4.26 depict a similar image to Figure 4.16, although slightly swayed by the energy expenditure swings. Due to those, Float4 shines as the best performer in bandwidth measurements and energy efficiency while Scalar drops down to near Float8 in these readings.

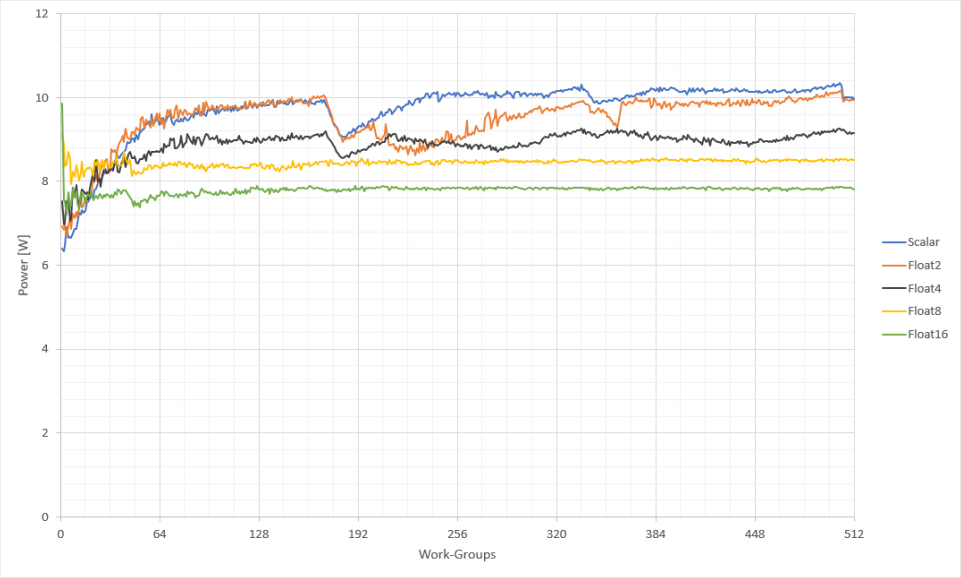

Figure 4.25: Float Load/Store Operation Power with increasing total Work-Groups

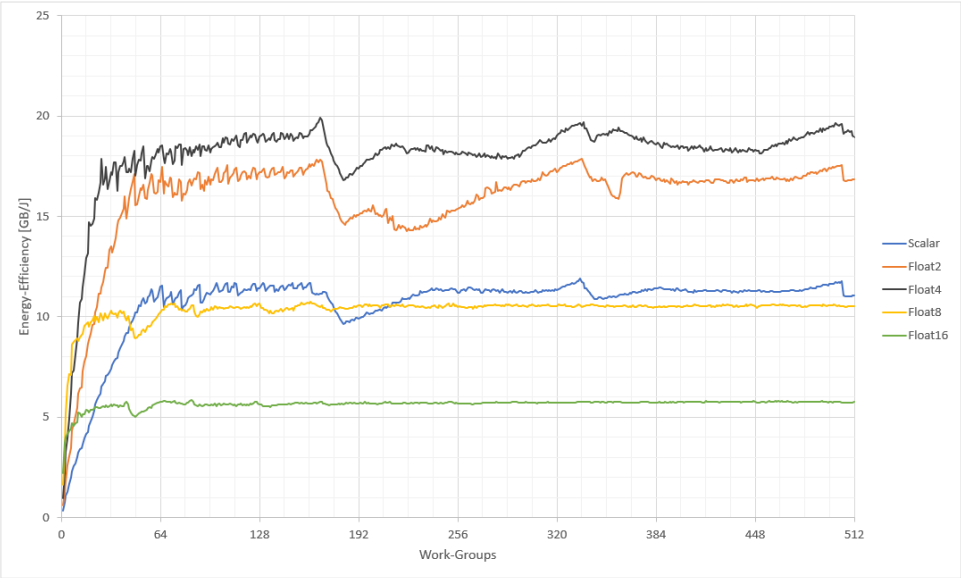

Figure 4.26: Float Load/Store Operation Energy Efficiency with increasing total Work-Groups

To conclude the display of the characterization done on the Gen 9.5 iGPU architecture, a more general review is taken. In the throughput computation department, finding the right balance among the operations contained in the workload seems essential, as the oscillating nature of the displayed plots shows that an uneven split workload can significantly lower the performance. For the addition operation kernel, Float4 and Float8, accompanied by Int4 and Int8, achieved the peak of 172 GFLOPS out of a possible 184 GFLOPS. For MAD operations, the same applies, but instead at absolute values of 345

GFLOPS out of 368 GFLOPS. Using the 64-bit double-precision floating-point data type, the obtained values are located at 43 out of 46 GFLOPS for addition kernels and 80 out of 92 GFLOPS for MAD kernels. All of these values achieve around 93%-95% of their respective upper bound, which coincides with the Intel Advisor generated roofs, employed in CARM. Note that the setup iGPU, the target of all the microbenchmarks, was simultaneously handling the graphic computation portion of the machine, as the main GPU of the computer. Otherwise, slight increases in the performance results can potentially be reached. Overall, different vectorization types only slightly impact the throughput of the iGPU.

Memory bandwidth proves to be a different case, as the 64B/cycle read/write bandwidth on the data port and the 32 B GRF registers are tailored to specific vectorization levels. As observed both in the plots and in the underlying GEN Assembly code, Float2 and Float4, for single-precision and Scalar and Double2, for double-precision, are the best performers, all reaching around 180 GB/s out of a potential 192 GB/s. Float16, Double8, and Double16 managed to reach only 45GB/s, due to requiring four times the requests to the DRAM, as their elements no longer fit in the GRF registers. Float8 and Double4 reached 90 GB/s for the same reason but making use of half the bandwidth per cycle, instead. For the data types that achieved the most bandwidth, OpenCL work-groups of multiples of 32 work-items are the most optimized.

As for the developed pointer-chasing algorithm, it provided a clearer view of the memory hierarchy. Although the clear drop in bandwidth when the buffer data no longer fits in the L3 Cache is noticeable, the transition from the LLC to the DRAM is harder to spot. This occurrence derives from the unpredictability of the order on which work-items are running. The necessary saturation in operations to achieve higher bandwidth has the backhanded effect of having a lot of data stored in the caches, avoiding a lot of main memory accesses. A better test to detect the plateaus in the memory system is the developed latency microbenchmark. The single-threaded usage of a pointer-chasing stridden array allows for much more control in the execution and guarantees access to the desired memory block. As such, the access times were registered at around 119 ns, 224 ns, and 301 ns, respectively for the L3 cache, the LLC, and the DRAM, with clear increases in latency as soon as the data size surpasses the size of the previous memory block where it was accommodated.

Power requirements are relatively constant, no matter the workload provided, at a range of 8 W to 10 W. Although a consistent factor is the slightly higher power requirement, the lower the vectorized level of the data type. This variance is even more prominent on bandwidth tests. In the energy efficiency department, given relatively constant power requirements, the achieved results are similar to the equivalent throughput or bandwidth plot. Nevertheless, Scalar and Vector2 data types tend to fall below Vector8 and Vector16 data types, given they achieve equal throughput, due to the higher energy expenditure.

4.5 Summary

Given an experimental setup composed of an Intel Gen 9.5 iGPU, which had its hardware architecture described in Chapter 2, this Chapter intended to characterize various important metrics such as throughput, memory bandwidth and latency, power consumption, and energy efficiency, for multiple data types and several vectorization levels. This process involved developing the microbenchmark set and using other tools delved into in Chapter 3. The obtained results were wholly interpreted and analyzed, providing some knowledge of the usually hidden behaviors and specifications of the microarchitecture. They also serve as a solid ground for future works to further investigate the architecture through other metrics or to develop performance characterization models such as CARM for power and energy efficiency.

5

Conclusions and Future Work

Modern iGPUs are significantly under-explored relative to their counterparts, dedicated GPUs. The size and location restrictions are serious drawbacks that make it inconceivable for an iGPU to have the same computing power as a discrete one. However, their virtues are originated from their restraints. This work takes as motivation the concept of usage of an iGPU for general-purpose computation, GPGPU, which, due to which its affiliation with the computer's main processor, is conceivable. Therefore, the work developed during this Thesis relates to building a foundation by thoroughly characterizing the performance of an Intel iGPU designed according to the Gen 9.5 microarchitecture.

The Thesis aimed to search for attainable ways to reach the documented boundaries of the iGPU's potential, to attempt to prove its viability as a GPGPU, through its high parallelization capabilities and shared memory hierarchy system. It is based on several Intel documents that describe in detail the microarchitecture of the Gen 9.5 iGPU and acknowledges state-of-the-art works and benchmark tests done with similar goals. In this Thesis, special attention is paid to performance characterizing models, in particular to Roofline Models, the ORM and the CARM, introduced as a tool to further analyze any defining results. Their application to CPUs as well as to discrete GPUs is an important contribution to this work and any future works related to the concepts of this Thesis.

The approach taken involved developing a microbenchmark set in OpenCL, intending to provide a collection of kernels for iGPU execution. These were built to search for the upper compute-bound and memory bounds that the architecture of an Intel Gen 9.5 iGPU supports. Coupled with additional external tools, energy expenditure, power requirements, and energy-efficiency readings were also taken and analyzed. In the experimental setup, kernels' execution in the iGPU provided up to 172 out of 184 theoretical GFLOPS for single-precision floating-point addition and 345 out of 368 GFLOPS for the MAD operation of the same data type. Roofline Models generated for these benchmarks confirmed the obtained results as the practical roofs for all the throughput tests run. As for bandwidth, identical data types provided up to 180 out of 192 maximum GB/s for the Float4 vectorization level, achieving roughly 94% of its capabilities. Finally, in power and energy efficiency, scalar-type kernels proved to require more power and, consequently, be less energy-efficient. On the other hand, higher vectorized types can output more data per energy expended.

Future works for these topics involve deeper characterization of useful metrics, such as the application of power requirements and energy-efficiency for a CARM-like model for this iGPU microarchitecture. Furthermore, works relating both CPU and iGPU for heterogeneous systems purposes can be developed, either with a focus on computation performance-wise or exploration of the effects of the shared memory system LLC or DRAM, contributing to an approach to using GPUs for general purposes alongside a CPU.

Bibliography

[1] A. Peleg and B. Ashbaugh and D. Helmly, "MICRO48-Tutorial on Intel® Processor Graphics: Architecture and Programming." [Online]. Available: software.intel.com/sites/default/files/Faster-Better-Pixels-on-the-Go-and-in-the-Cloud-with-OpenCL-on-Intel-Architecture.pdf

[2] J. Peddie, "Is it Time to Rename the GPU?" [Online]. Available: https://www.computer.org/publications/tech-news/chasing-pixels/is-it-time-to-rename-the-gpu

[3] Khronos, "The open standard for parallel programming of heterogeneous systems." [Online]. Available: www.khronos.org/opencl/

[4] ——, "The opencl™ c 2.0 specification." [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_C.pdf

[5] S. Junkins, "The Compute Architecture of Intel® Processor Graphics Gen9," 2015. [Online]. Available: https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf

[6] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual."

[7] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, 04 2009.

[8] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, Jan 2014.

[9] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, and Z. A. Matveev, "Performance analysis with cache-aware roofline model in intel advisor," in *2017 International Conference on High Performance Computing Simulation (HPCS)*, July 2017, pp. 898–907.

[10] Intel, "Intel Advisor." [Online]. Available: https://software.intel.com/en-us/advisor

[11] "Gen9.5 - microarchitectures - intel." [Online]. Available: en.wikichip.org/wiki/intel/microarchitectures/gen9.5

[12] A. Peleg and B. Ashbaugh and D. Helmly, “Maximize Application Performance On the Go and In the Cloud with OpenCL on Intel Architecture.” [Online]. Available: software.intel.com/sites/default/files/Faster-Better-Pixels-on-the-Go-and-in-the-Cloud-with-OpenCL-on-Intel-Architecture.pdf

[13] R. G. Ilgner and D. B. Davidson, “A comparison of the fdtd algorithm implemented on an integrated gpu versus a gpu configured as a co-processor,” *2012 International Conference on Electromagnetics in Advanced Applications*, pp. 1046–1049, 2012.

[14] A. Chikin, J. N. Amaral, K. Ali, and E. Tiotto, “Toward an analytical performance model to select between gpu and cpu execution,” in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2019, pp. 353–362.

[15] U. Gupta, J. Campbell, U. Y. Ogras, R. Ayoub, M. Kishinevsky, F. Paterna, and S. Gumussoy, “Adaptive performance prediction for integrated gpus,” in *Proceedings of the 35th International Conference on Computer-Aided Design*, ser. ICCAD ’16. New York, NY, USA: ACM, 2016, pp. 61:1–61:8. [Online]. Available: http://doi.acm.org/10.1145/2966986.2966997

[16] N. Farooqui, R. Barik, B. T. Lewis, T. Shpeisman, and K. Schwan, “Affinity-aware work-stealing for integrated cpu-gpu processors,” in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2851141.2851194

[17] V. Garcıa, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Pena, “Evaluating the effect of last-level cache sharing on integrated gpu-cpu systems with heterogeneous applications,” in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2016, pp. 1–10.

[18] G. Lupescu, E. Slusanschi, and N. Tapus, “Analysis of opencl work-group reduce for intel gpus,” in *2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, Sep. 2016, pp. 417–423.

[19] ——, “Analysis of thread workgroup broadcast for intel gpus,” in *2016 International Conference on High Performance Computing Simulation (HPCS)*, July 2016, pp. 1019–1024.

[20] A. Ilic, F. Pratas, and L. Sousa, “Beyond the roofline: Cache-aware power and energy-efficiency modeling for multi-cores,” *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 52–58, Jan 2017.

[21] A. Lopes, F. Pratas, L. Sousa, and A. Ilic, “Exploring gpu performance, power and energy-efficiency bounds with cache-aware roofline modeling,” in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 259–268.

[22] G. Lupescu, E. Sluşanschi, and N. Tăpuş, “Using the integrated gpu to improve cpu sort performance,” in *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, Aug 2017, pp. 39–44.

[23] F. Paterna, U. Gupta, R. Ayoub, U. Y. Ogras, and M. Kishinevsky, “Adaptive performance sensitivity model to support gpu power management,” in *Proceedings of the 1st Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems*, ser. ANDARE ’17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3152821.3152822

[24] P. Gera, H. Kim, H. Kim, S. Hong, V. George, and C. Luk, “Performance characterisation and simulation of intel’s integrated gpu architecture,” in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2018, pp. 139–148.

[25] “Performance Application Programming Interface.” [Online]. Available: https://bitbucket.org/icl/papi/wiki/Home

[26] R. Ioffe, “Introduction to gen assembly.” [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-gen-assembly