TÉCNICO LISBOA

Performance Analysis of Intel Gen9.5 Integrated GPU Architecture

Helder Francisco Pereira Duarte

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Doctor Aleksandar Ilic

Examination Committee

Chairperson: Doctor António Manuel Raminhos Cordeiro Grilo
Supervisor: Doctor Aleksandar Ilic
Member of the Committee: Doctor Ricardo Jorge Fernandes Chaves

June 2018

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 my family and friends, whose support was paramount to the completion of this thesis, and to INESC-ID Lisboa, for providing the facilities in which this thesis was developed. In particular I would like to thank Diogo Marques for his tips that helped drive my research forward and, of course, to Prof. Aleksander for his incredible patience in dealing with me.

Resumo

Recentemente os CPUs vêm equipados com placas gráficas integradas. Este acoplamento tem o potencial de oferecer ganhos de desempenho consideráveis caso as ditas GPUs sejam usadas como aceleradores. No entanto, placas gráficas integradas têm dificuldade em atingir os níveis de desempenho que placas discretas proporcionam devido ao menor número de núcleos. Contudo, a sua proximidade com o CPU significa uma partilha de dados com menos sobrecargas associadas. Ademais, as vantagens de partilhar a hierarquia de memória com o processador e o consumo de energia mais baixo que as placas discretas à custa de desempenho permite atingir níveis de eficiência energética mais elevados. É objetivo desta tese caracterizar a GPU integrada da Intel e avaliar os seus tectos de desempenho por forma a estudar a sua eficiência energética.

Através da exposição dos contadores de desempenho da GPU, certas intuições podem ser retiradas acerca de como extrair o máximo de desempenho possível da arquitectura e como melhor explorar a hierarquia de memória partilhada.

Palavras-chave: Placa Gráfica Integrada, Desempenho, Potência, Eficiência Energética.

Abstract

CPUs have since the last years come equipped with integrated GPUs. Such coupling has the potential to offer performance improvements if said GPUs are used as accelerators. However, integrated GPUs can hardly reach the performance levels obtained with discrete GPUs due to having lower core counts. Nevertheless, their proximity to the CPU allows for sharing of data with less overhead. Furthermore, the advantages of sharing the same memory hierarchy and the lower power consumption at the cost of performance, allows for high energy efficiency gains. This thesis aims to characterize the Intel GPU micro-architecture and to benchmark its performance upper-bounds to study the limits of its energy efficiency.

Through exposing the Intel GPU's performance counters, greater insights can be gleamed about how to extract peak performance from the architecture and how to best exploit the shared memory hierarchy.

Keywords: Integrated GPU, Performance, Power consumption, Energy efficiency.

Contents

List of Figures

List of Tables

List of Algorithms

List of Code

List of Acronyms

ALU Arithmetic and Logic Unit

API Application Programming Interface

BAR Base Address Register

BDF Bus Device Function

Blitter BLock Image Transferrer Engine

CLIP Clipper

CPU Central Processing Unit

CS Command Streamer

CUDA Compute Unified Device Architecture

DDR Double Data Rate

DP Double Precision

DRAM Dynamic Random Access Memory

DS Domain Shader

EU Execution Unit

FLOP Floating Point Operation

FMA Fused Multiply Add

FPU Floating point Processing Unit

GFLOPS Giga Floating Point Operations per Second

GPU Graphics Processing Unit

GS Geometry Shader

GTD Global Thread Dispatcher

GTI Graphics Technology Interface

HPC High Performance Computing

HS Hull Shader

IGPU Integrated Graphics Processing Unit

ILP Instruction Level Parallelism

ISA Instruction Set Architecture

JIT Just In Time

LLC Last Level Cache

LTD Local Thread Dispatcher

MAD Multiply and ADd

MLP Memory Level Parallelism

MMIO Memory Mapped Input Output

OA Observation Architecture

PAPI Performance API

PCIe Peripheral Component Interconnect Express

PCI Peripheral Component Interconnect

PMU Performance Monitoring Unit

PP0 Power Plane 0

RAPL Running Average Power Limit

SDK Software Development Kit

SF Strip/Fan

SKU Stock Keeping Unit

SLM Shared Local Memory

SOL Stream Output Logic

SP Single Precision

SoC System on Chip

TE Tesselation Engine

URB Unified Return Buffer

VF Vertex Fetch

VS Vertex Shader

VUE Vertex URB Entry

WM Windower/Masker

Chapter 1

Introduction

The emergence of General Purpose Graphics Processing Unit (GPGPU) computing followed the trend when GPUs shifted from mere fixed function accelerators limited to matrix algebra workloads, to more programmable processors capable of addressing the demand for more general purpose computation to the graphics pipeline. In turn, and coupled with the increasing parallel capabilities of GPUs, the first efforts to address the GPU's parallel performance started surging. These first attempts were set back by having to rely on graphics libraries and working with shaders, instead of abstracting and generalizing the type of data being worked with. This sparked the effort to develop programming environments targeting the GPU while allowing the programmer to abstract away from the conventional graphical programming environment. The most commonly used GPGPU computing environments in use today are Compute Unified Device Architecture (CUDA), targeting NVidia's GPUs and OpenCL, a standard for heterogeneous computing targeting different brands of GPU, Field Programmable Gate Arrays (FPGA) and Central Processing Unit (CPU) manufacturers.

With the ability to perform general purpose computing on the massively parallel processors that are GPUs, these started to be integrated into supercomputing servers, being mainly used for scientific computing and simulation. Nevertheless, these novel workloads for GPUs started to raise concerns over power consumption, and so this became increasingly more important as time passed.

By this point, manufacturers started to group GPU and CPU in the same die, opening up feasibility for heterogeneous systems. These integrated GPUs, with their proximity to the CPU hold greater energy efficiency potential than their discrete counterparts, at the expense of lower theoretical performance roofs. Notwithstanding, integrated GPUs have now taken flight and research work in their feasibility is in demand.

1.1 Motivation

In order to tackle growing computational demands from modern applications, processor architectures suffered a shift to more parallel structures by adding more cores into the die. This change provided a higher performance ceiling and software developers were able to speedup up several algorithms by

making adjustments to the algorithms themselves to make use of parallelization. This was not enough however, and a variety of scientific simulation workloads such as weather forecast, fluid dynamics and computer vision demanded higher parallelization to afford a reasonable execution time. Such processors started to be integrated in clusters with thousands to millions of cores to fulfill the demand for these workloads. However, with such a high number of cores and processing power comes considerate energy consumption, which forced developers to take a look at different approaches to solving these massively parallel workloads. This sparked the advent of using GPUs and their native parallel processing power to tackle these problems. Even with a lower frequency and with less developed cores in regard to branch prediction and instruction level parallelism (ILP), two techniques recent processors excel at and are used to further increase performance, the GPUs were able to achieve consistent gains in performance due to the sheer amount of parallel processing power they were capable of outputting. They thus started to be used more and more in High Performance Computing (HPC) and nowadays are a common staple in supercomputers and clusters.

Despite the growing use of GPUs for general purpose computing, there still remained the problem that the GPU and CPU reside far apart and require a high bandwidth interconnect, namely, PCI-Express (PCIe), in order to move data to and fro. This required intelligent ways to mask data movement to achieve the desired speedups, however, there is no doubt that it impacted the performance of heterogeneous CPU+GPU systems. With integrated GPUs (iGPU), where the memory hierarchy is shared and proximity to the CPU avoids the necessary interconnect structure, this problem is less pronounced. It becomes therefore important to study the attainable performance of these systems in order to verify their effectiveness and energy efficiency.

1.2 Objectives

The objectives aimed for with this thesis fall within the scope of the following:

- Provide a tool that can read Intel GPU performance counters specifically for the Gen9.5 micro-architecture
- Evaluate the architecture's theoretical floating point performance roof and memory bandwidth roof using provided OpenCL kernels developed solely for the effect.
- Evaluate energy efficiency of different OpenCL provided data types and operations.

1.3 Main Contributions

The main contributions of this Thesis ride on this being the first work of its kind to exploit the Intel's Gen9.5 micro-architecture of integrated GPUs without relying on software architectural simulators or machine learning predictive models. The maximum reachable performance of the architecture is analyzed, together with memory bandwidth and energy efficiency. An open source tool to read GPU performance counters is provided for this end, which can be used to help later works.

Prior to the work on this thesis, a preliminary work based on the CPU was performed that contributed the following works communicated at the HPCS 2017 international conference:

- Diogo Marques, Helder Duarte, Aleksandar Ilic, Leonel Sousa, Roman Belenov, Philippe Thierry and Zakhar A. Matveev. "*Performance Analysis with Cache-Aware Roofline Model in Intel Advisor*", In Proceedings of the International Conference on High Performance Computing & Simulation (HPCS'17), Genoa, Italy, July 2017. (*paper in collaboration with Intel Corporation*);
- Diogo Marques, Helder Duarte, Leonel Sousa, and Aleksandar Ilic. "*Analyzing Performance of Multi-cores and Applications with Cache-aware Roofline Model*", In Special Session on High Performance Computing for Application Benchmarking and Optimization (HPBench'17), collocated with International Conference on High Performance Computing & Simulation (HPCS'17), Genoa, Italy, July 2017. (*extended abstract*)

1.4 Thesis Outline

The remainder of this document is organized as follows. Chapter 2 lays the foundation of the underlying architecture of the target GPU. Furthermore, it introduces the performance counters supplied by the architecture. It ends by reviewing state of the art works regarding GPUs and their extended use in the recent High Performance Computing (HPC) landscape. Chapter 3 describes the developed tool in detail, giving special attention to several design decisions that impacted the development life cycle of the tool, as well as the performance counters configuration process and the various caveats one needs to take into account, for instance the interaction with the graphics driver. Chapter 4 starts by debunking the Intel GEN ISA and the generated assembly by the OpenCL JIT compiler, exposing several sources of overheads in the compiled kernels. In this chapter the developed benchmarks are discriminated and the architecture's attained roofs are presented, not only for floating point performance and memory bandwidth but for power consumption as well. Lastly, Chapter 5 presents the conclusions obtained from the work done in this thesis, and offers insights into what future works could dive upon.

Chapter 2

Intel Integrated GPU Architecture and State of the Art Approaches

Discrete GPUs are praised for their parallel performance unmatched by CPUs. Not only can they handle complex 3D graphical workloads, but they also have the ability to provide great performance boosts when used as accelerators in the GPGPU computing spectrum. This allows the offloading of certain workloads, freeing up the CPU for even greater performance benefits. GPUs shine when executing heavy parallel applications that can make use of their hundreds of cores and complex memory hierarchy, which is fine-tuned to provide high throughput to get data to the cores as soon as possible. This is in direct contrast to a CPU's memory hierarchy, where the focus is on low latency.

This advantage over parallel workloads comes at a loss of single core performance, not only lacking the powerful branch prediction commonly seen in CPUs and out of order execution, one of the key features that allow CPUs to reach high instruction level parallelism, but supporting lower core frequencies as well. However, the main bottleneck discrete GPUs find comes not from their poor performance on sequential workloads, but from data transfers to and from the CPU domain. This can somewhat be suppressed by using latency hiding techniques, but such methods are highly dependent on the workload and are not always easily implemented. Integrated GPUs have the advantage over discrete ones by sharing the memory space between the CPU, avoiding the data transfers that hinder reaching maximum performance, with the caveat of having to share the same die as the CPU, forcing a lower number of parallel units which in turn lower peak performance.

Due to these various differences between discrete and integrated GPUs, integrated GPUs are still widely underused for GPGPU. One of the main focus of this thesis is to prove that integrated GPUs have a place in the GPGPU spectrum, be it from a performance standpoint or from an energy efficiency one.

Intel's Gen architecture comes mid way between a GPU and a CPU, supporting a number of cores that may seem low when compared to other GPU architectures, but with these same cores offering functionalities that are more akin to a CPU's vectorization unit than the less advanced common GPU cores, as will be covered later in this chapter. This chapter will therefore focus on describing Intel's latest GPU architecture, the Gen9.5, while also giving an introduction to the performance counter unit present

in the GPU.

2.1 Intel GPU Architecture

The Intel Gen9.5 graphics architecture follows closely after Gen9 with a myriad of light improvements, the most notable being the switch to 14 nm process fabrication technology, providing lower power consumption than its predecessor. The architecture is divided into 3 main components, the Slice, the Unslice and the Display components (subdomains).

Figure 2.1 depicts a streamed down System on Chip (SoC) architecture overview where the CPU cores and the GPU are connected. The Graphics Technology Interface (GTI) is the bridge between the GPU and the SoC Ring Interconnect, and it represents a ring network topology that connects all agents of the die. The Ring is a bi-directional 32-byte wide data bus with discrete channels for request, snoop and acknowledge. The agents connected to the Ring are the GPU, several CPU cores, the Last Level Cache (LLC) and the System Agent. The System Agent incorporates various components, of note are the Dynamic Random Access (DRAM) Memory Management Unit and various I/O controllers such as PCIe. The GTI is also home to global memory atomics hardware shared by both the GPU and the CPU cores. Last but not least, the GTI carries out power management for the GPU and harbors interfaces for the different clock domains between the GPU and the rest of the SoC. The bus that connects the GTI to the GPU supports 64 B/cycle read and write operations, although an additional configuration is also available that offers the reduction of the write throughput to 32 B/cycles (mainly for low power constraints).

Focusing on the GPU, Figure 2.2 provides a general overview of the 3 main components of the GPU and how they interact with each other. Given the scope of this thesis, a predominant focus will be paid on exploring the capabilities of the Slice components, since the Slice is where all GPGPU computations take place and the Unslice holds the 3D and the Media pipelines, which do not fall on the scope of this thesis. Nonetheless, the Unslice and its two major pipelines will be discussed with brevity due to their interactions between the Slice as well as their relevance in the counter architecture.

The Unslice contains fixed function media and geometry hardware, and it also provides access to the LLC and, in turn, to the DRAM, through the GTI. The Unslice contains the 3D and Media Pipelines, which provide fixed functions for the computation of 3D shaders and media kernels respectively. In the Unslice, the flow of execution of the 3D and media pipelines are controlled by the Command Streamer (CS). This unit manages the context switch between the two pipelines and forwards command streams received from the CPU to the different stages of the pipelines. It also manages the Unified Return Buffer (URB). The URB is a general purpose buffer located in the L3 cache that has the purpose of transferring data between different threads or between threads and fixed function units. It is an important part of the 3D pipeline, although it is not largely used for GPGPU computing.

For certain fixed function pipeline stages, it is common to request general purpose computation from the Execution Units (EU) in the Slice. To achieve this, they communicate with the Global Thread Dispatcher (GTD) and provide it with URB handles and other required types of data. The URB entries

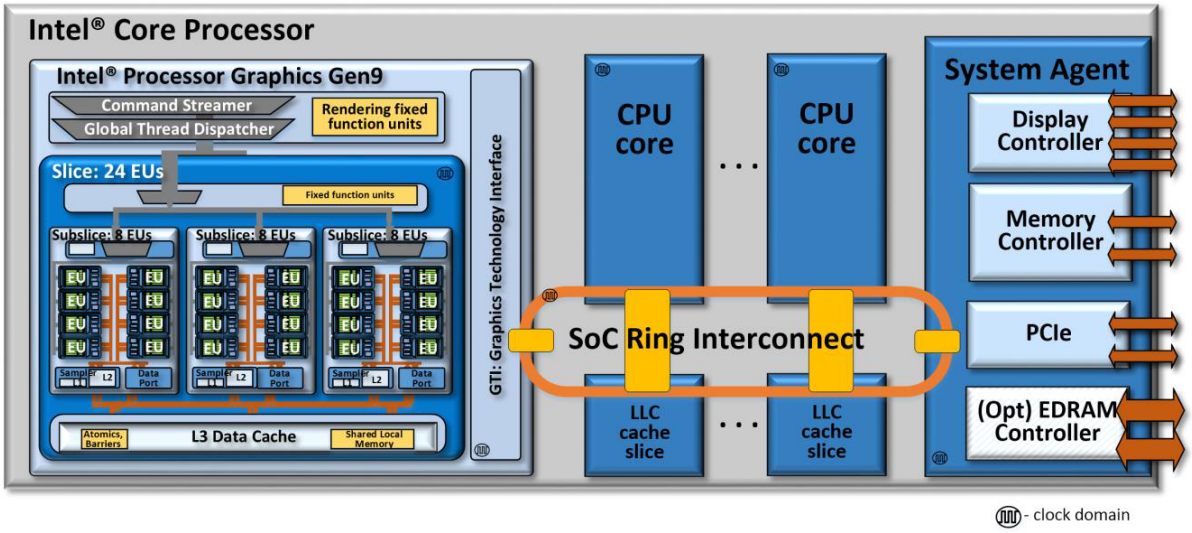


Figure 2.1: SoC Architecture [1].

are then pre-loaded in the General Register File (GRF) in the EUs of the Slice as per the thread launch.

The Command Streamer is responsible for the parsing of commands received from the CPU, mainly through the respective driver. Depending on the messages received, the Command Streamer distinguishes what kind of workload is being requested. It uses this information to route individual commands to the respective units in the GPU. This can be 3D workloads coming from a graphics library (such as DirectX or OpenGL), media workloads requesting the decoding of compressed video streams, or even GPGPU kernels originating from OpenCL runtime or other GPGPU heterogeneous execution environment. These are forwarded to the GTD where it can operate in two distinct modes. If the workloads do not involve barriers or Shared Local Memory (SLM) accesses, the workload is evenly distributed among the subslices to maximize occupancy (which in turn is expected to maximize throughput of instructions). If the kernels make use of SLM or barriers, the GTD attempts a better arrangement of thread groups to subslices in order to conform to restrictions in the access to barriers and to SLM.

The Media pipeline is responsible not only for media workloads, but for general purpose computing as well. The Thread Spawner resides in the Media pipeline, more specifically in the Video Front End (VFE), therefore, GPGPU APIs (like OpenCL) refer to the Media pipeline to request thread generation.

The 3D pipeline is composed of eight stages, each with its own specific function. The first stage after the Command Streamer, the Vertex Fetch (VF), acquires the vertex data from memory and formats them according to their attributes (*e.g.* position, color) while writing the results to Vertex URB Entries (VUE). References to the VUEs are passed down the pipeline. Next in the pipeline comes the Vertex Shader (VS) which sends requests to the GTD to compute the vertices in the Slice. Afterwards, the tesselation fixed functions. Tesselation is an optional part of the 3D pipeline. Normally, lighting occurs at a polygon level. However, with tesselation, polygons are divided and lighting is then performed in greater detail. This provides better lighting effects and is a technique that is being implemented more often in 3D workloads as of late. The Hull Shader processes primitives for the tesselation processes. At this point, it becomes important to understand the difference between vertices and primitives. A vertex is a structure made out of points in a 3D reference frame, containing attributes such as position and color.

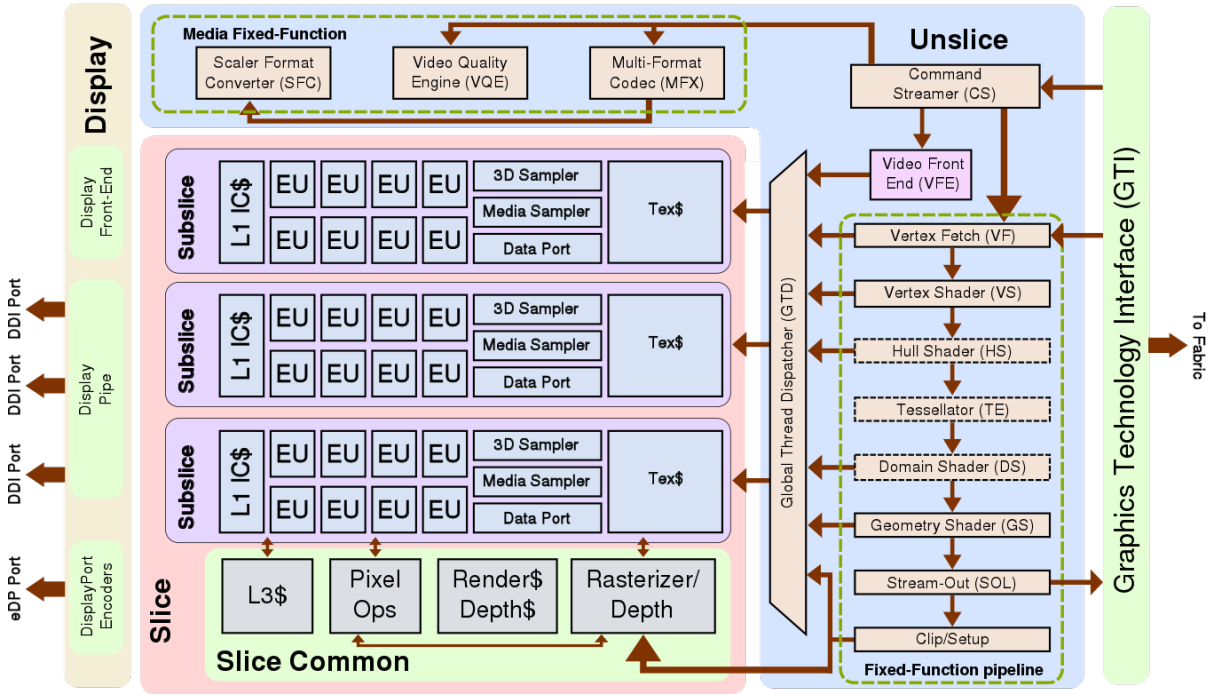


Figure 2.2: GPU general layout [2].

Primitives represent shapes, *e.g.* lines, and are constructed by vertices. The Tesselation Engine (TE) is evoked after the Hull Shader to complete the tesselation process and hands the Domain Shader (DS), the next stage in the 3D pipeline, the points that will be converted into vertices. The Geometry Shader, similarly to the Vertex Shader, spawns threads to handle the processing of the primitives coming from former stages of the pipeline, which in turn get to the Stream Output Logic (SOL) that is responsible for sending vertices out to memory buffers. The objects are then clipped by the Clipper (CLIP) according to the scene. This process determines which objects will be visible. Some objects will be positioned behind other objects, according to the point of view of the camera, or fall completely outside the screen. The objects will then pass through the Strip/Fan (SF) and the Windower/Masker (WM) where the rasterization takes place. Rasterization is the process by which pixels are mapped to primitives. Finally, the pixels are sent to the screen for display.

As previously referred, the Slice contains the general purpose computation units that take care of all the calculations requested by the other pipelines and by GPGPU kernels. It is divided into Subslices, which in turn, have multiple EUs, or Execution Units that are responsible for computations. The Unslice requests use of the EUs whenever the 3D or Media pipelines require general computations to be performed. For this end, the 3D and Media pipelines send messages to the Global Thread Dispatcher to request thread initiation. It is the job of the Global Thread Dispatcher to assign requested computations to hardware threads and to allocate register space in the EUs. Depending on the SKU configuration, an Intel iGPU can be equipped with a different number of Slices with varying numbers of subslices and EUs per subslice. In Intel KabyLake, the GPU micro-architecture Gen9.5 has 5 distinct SKU configurations: GT1 with 1 slice of 2 subslices and 6 EUs per subslice; GT1.5 with one slice of 3 subslices and 6 EUs per subslice; GT2, being the standard configuration, with 1 slice of 3 subslices and 8 EUs per subslice

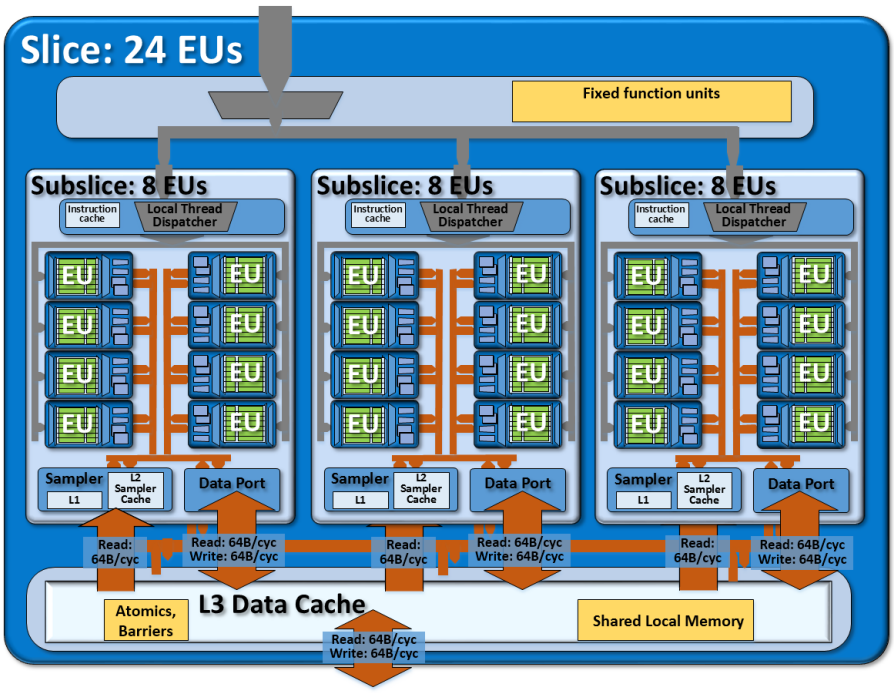


Figure 2.3: Architectural overview of the Slice [1].

and GT3 with 2 slices of 3 subslices and 8 EUs per subslice [3].

2.1.1 Integrated GPU Slice Architecture

Slices are arrangements of subslices. For the GT1 configuration, a slice contains 2 subslices, while for all other configurations, each slice is a group of 3 subslices. In addition to the subslices, a slice also contains a banked L3 cache and respective SLM. Fixed functions for handling atomic operations are contained in the slice, as well as various 3D and media fixed functions [1].

The L3 data cache present in the GPU, not to be confused with the CPU's L3 Cache, which is referred to as the LLC in regards to the GPU (functioning as an L4 cache, if counting with the Sampler caches), has a capacity of 768 KB per slice, with cachelines of 64 B. Each subslice has its own L3 partition, and these partitions are aggregated together and function as a single cache for the slice. For configurations with a higher number of slices, all the L3 partitions of each subslice in each slice are also aggregated together to form a unique, GPU wide L3 cache. All Samplers and Data Port units have their own individual interface to the L3 cache, capable of supporting read and write bandwidths of 64 B per cycle. Thus, this is linearly expanded by the amount of subslices in the GPU. For the GT2 configuration, for example, with a single slice, the total theoretical bandwidth is 64 B/cy $\times$ 3 Subslices = 192 B/cy. However, not all workloads will be able to reach this upper-bound, for which careful partitioning of the problem size throughout the slices is required. Data to and from the L3 can only be transferred in 64 B wide packets, *i.e.*, a cacheline. Therefore a read instruction from a thread in a EU of a single precision floating point value (4 B) always translates into a 64 B data traffic. The OpenCL JIT compiler is known to optimize this procedure, grouping data fetches to avoid wasted bandwidth [1].

The SLM is situated in the L3 cache and serves as a programmer managed scratchpad memory for

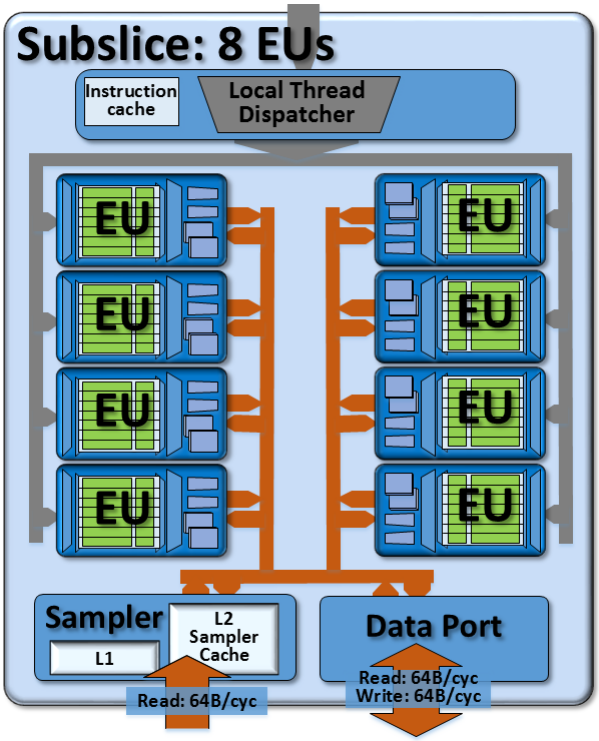


Figure 2.4: Architectural overview of the Subslice [1].

sharing data across different EUs located in the same subslice. There are many workloads that take advantage of sharing of data between threads, namely 2D image processing and FDTD computation (in essence any workload that is not massively parallel and exhibits constraints at the border). Each subslice has access to 64 KB of SLM. As SLM resides in the L3, access times do not benefit from lower latency. However, due to it being more highly banked, it allows full throughput even for access patterns that are not 64 B aligned or for data not contiguously adjacent in memory. Contrary to the L3 cache, SLM is not coherent with other memory structures.

2.1.2 Integrated GPU Subslice Architecture

A subslice aggregates a number of EUs which are responsible for computations. According to the product configuration, a subslice can have a different number of EUs. Subslices of GT1 and GT1.5 product configurations both contain 6 EUs. For the larger GT2, GT3 and GT4 configurations, each subslice comes with 8 EUs. Besides the EUs, a subslice is composed of a Local Thread Dispatcher (LTD), an Instruction Cache, the Data Port unit and a Sampler unit.

The LTD is responsible for sharing the workload throughout the EUs in the subslice. It works in tandem with the GTD located in the Unslice in order to provide optimal thread coverage. The LTD does not communicate with the GTD. As a result, the LTD attempts to assure a uniform distribution of the threads given by the GTD to the EUs.

The Sampler is composed of L1 and L2 read-only caches and it is used solely for texture and image surfaces. It also provides fixed function hardware for conversion between address and image coordinates, various clamping modes commonly seen in image processing algorithms, such as mirror, wrap,

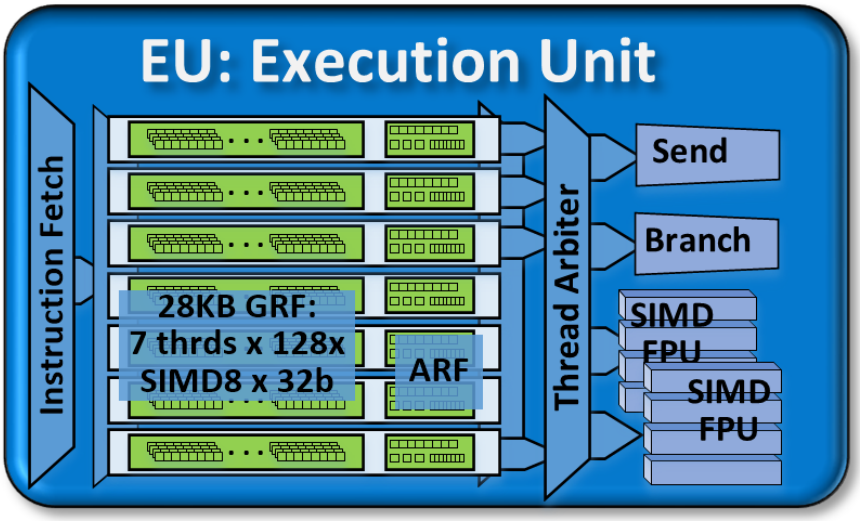


Figure 2.5: Architectural overview of the Execution Unit [1].

border and clamp, as well as sampling filtering modes such as point, bilinear, trilinear and anisotropic. There operations are not commonly used in GPGPU, thus they are reserved for 3D workloads. As previously mentioned, the Sampler supports a read throughput from the L3 cache of 64 B per cycle.

The Data Port unit is connected to the L3 fabric in the slice and it is responsible for serving memory load and store request. It supports read and write throughputs of 64 B per cycle. Hence, all memory requests from the EUs have to pass through the Data Port unit [1].

2.1.3 Integrated GPU Execution Unit Architecture

The EUs are composed of a register file of 128 general purpose 256 bit registers per thread. These compose the General Register File (GRF). A set of architecture-specific registers are also present in each EU, called the Architectural Register File (ARF). These registers mainly serve for resource management such as keeping track of the instruction pointer per thread and managing dependencies. They are mainly used by the hardware and driver. In Section 4.1 they will be further developed because of their role in the GEN assembly. Each EU contains four distinct functional units: the Send unit, the Branch unit and two Floating Point Units (FPU). These allow one EU to co-issue up to four instructions per cycle, albeit they must come from different threads. To assure that each EU has enough workload to maintain this throughput, each EU supports up to seven hardware threads. In total, the GRF amounts to 128 $\times$ 256 bits / 8 / 1024 * 7 threads = 28 KB for a single EU, with 4 KB per thread [1].

Memory operations are compiled into GEN Assembly ”*send*” instructions, which are dispatched to the Send unit. Each *send* instruction includes a pointer to the message payload that the Send unit uses to correctly format messages to send to the Data Port unit in the subslice. The Data Port unit, in turn, forwards the memory access requests and delivers the data back to the GRF (in case of a load).

Branch instructions are handled by the Branch unit. Branching takes up an important role in maximizing performance, with some aspects of the EU's Instruction Set Architecture (ISA) having mechanisms specifically for dealing with branch prediction and to avoid thread divergence. Thread divergence is a

common performance pitfall where a given workload has two possible paths that depend on the outcome of a comparison. If the threads being executed in the EU follow the same path, then no branching overhead is endured. However, if they diverge, then all threads affected are set to execute both paths to avoid staling the pipelines, introducing overheads in the computation. This development is common with conditional instructions and in loop control.

The two FPUs in each EU are capable of both single precision and half precision floating point and integer computations. One of the FPUs can also perform double precision floating point computation and transcendental math operations. Although the EU's ISA allows logical instructions of 1, 2, 4, 8, 16 or even 32 wide 32 bit data types, the FPUs are physically only 4-wide, allowing for four simultaneous executions of single precision floating point operations per FPU. However, for wider vector types, these units are fully pipelined, which allows reaching maximum throughout independent of specified SIMD width by reusing the same units of execution. Each FPU has the ability to perform one single precision MAD operation (add and multiply in a single instruction) in one cycle. The EU can therefore achieve a theoretical throughput of 16 single precision floating point operations per cycle: 2 (MAD) $\times$ 2 FPUs $\times$ SIMD-4 = 16 FLOP/cy. Given that only one of the FPUs support double precision and that double precision operations offer only half of the single precision throughput, double precision throughput drops four-fold, resting at 4 FLOP/cy for MAD operations.

2.2 Intel GPU Performance Counters

The available counters in Intel Gen GPUs are divided into three categories: the Aggregated or A counters, the Boolean or B couters and the C counters. The A counters contain 3D graphics metrics, such as the number of threads dispatched in each stage of the 3D pipeline, as well as some GPGPU metrics, for instance the number of total barrier messages or SLM reads and writes. The A counters also contain the programmable counters that offer more flexibility for an end-user and that are exposed by the tool developed in this thesis. The B counters are counters that count binary signals, apply programmable logic to them and propagate them to other B counters that allow for further programmable logic. This allows for complex boolean filtering of events. The B and C counters suffer from lack of documentation, and although they can be collected by the developed tool by applying the same mechanism that will be described in Chapter 3, they are not subjected to analysis.

The Intel GPU comes with its own performance counter architecture, which is referred as the Observation Architecture (OA) [3]. The OA exposes counters A0 to A35, B0 to B7 and C0 to C7. Counters A7 to A20 and B counters are especially noteworthy for their configurability. All A counters are 40 bit wide, with the exception of A32 to A35 and B and C counters that are 32 bit wide. The counters are by default disabled and they require a configuration step to be performed. Only counters A4, A6, A19 and A20 are free running, functioning continuously without requiring configuration. The full set of A counters is depicted in Apendix A.

The OA currently allows two different ways of accessing the performance counters. On one hand, it exposes all its counter registers through Memory Mapped Input/Output (MMIO). On the other hand,

by receiving a specific message, *i.e.*, MI_REPORT_PERF_COUNT, the GPU writes a snaptshot of the counters to a predefined memory position. The snapshot can take one of four different formats, and they are displayed in Tables 2.1, 2.2, 2.3 and 2.4. The tables not only display the different formats, but they also illustrate how they are accessed in memory. Each cell in Tables 2.1, 2.2, 2.3 and 2.4 corresponds to a 32 bit value and the first cell of each table shows what counter is accessed by the first double word of the pointer returned by the API used to read the counters. Counter format 000 mainly offers a small set of A counters, while counter format 111 on the other hand offers B and C counters. Counter format 010 allocates more memory than the previous two formats, but aggregates A, B and C counters. Nonetheless, it is counter format 101 that offers the full set of counters. It is also the one that offers the full width of the counter registers, aggregating the last bytes from the A counters into 32 bit values each containing 4 high bytes from 4 distinct counters. The formats are specified through the Counter Select bits, *i.e.*, a set of control bits in the OACONTROL register. For 40 bit wide counters, there exists one 32 bit register for the lower 32 bits and another 32 bit register for the remaining 8 bits.

All counter formats include four non-counter registers: GPU_TICKS, CTX_ID, TIME_STAMP and RPT_ID. The GPU_TICKS register acts as a clock counter, which content is incremented every GPU clock cycle. This property makes it dependent on frequency, so it should only be relied on if the GPU frequency has been previously set. TIME_STAMP refers to the TIME_STAMP counter that is configured in tandem with the performance counters. Both the configuration of the counters and this register will be scrutinized in Section 3.3. The CTX_ID refers to the context ID of the currently active context in the render engine. Finally, the RPT_ID offers a set of control bits that contain various information about the current performance monitoring in different fields (used in the developed tool ins the scope of this thesis). As such, the fields will be enumerated. Bits 31:26 of this 4-byte value hold the squashed slice clock frequency, which is an encoded value for the slice clock frequency decoded by the proposed tool. Bit 25 indicates if current context ID is valid, *i.e.*, if the context selected for performance counter monitoring is the same as the one actively being executed. Bits 24:19 indicate the reason of the triggering of the report, with the reasons being “Timer Triggered”, “Internal report trigger 1”, “Internal report trigger 2”, “Render Context Switch” and “GO Transition from ‘1’ to ‘0’”. Through analysis of the results presented in this thesis, it can be observed most reports are triggered by “Timer”, although this information is discarded by the tool. The rest of the bits are collected using functions from Intel GPU Tools but their data is ignored, due to most of them being undisclosed.

Table 2.1: Performance counters report format: Counter Select = 0b000.

A10 (low dword)	A9 (low dword)	A8 (low dword)	A7 (low dword)	GPU_TICKS	CTX_ID	TIME_STAMP	RPT_ID
A18 (low dword)	A17 (low dword)	A16 (low dword)	A15 (low dword)	A14 (low dword)	A13 (low dword)	A12 (low dword)	A11 (low dword)

Table 2.2: Performance counters report format: Counter Select = 0b010

A10 (low dword)	A9 (low dword)	A8 (low dword)	A7 (low dword)	GPU_TICKS	CTX_ID	TIME_STAMP	RPT_ID
A18 (low dword)	A17 (low dword)	A16 (low dword)	A15 (low dword)	A14 (low dword)	A13 (low dword)	A12 (low dword)	A11 (low dword)
B7	B6	B5	B4	B3	B2	B1	B0
C7	C6	C5	C4	C3	C2	C1	C0

Table 2.3: Performance counters report format: Counter Select = 0b111.

C3	C2	C1	C0	GPU_ TICKS	CTX_ID	TIME_ STAMP	RPT_ID
B7	B6	B5	B4	B3	B2	B1	B0

Table 2.4: Performance counters report format: Counter Select = 0b101.

RPT_ID	TIME_STAMP	CTX_ID	GPU_TICKS	A0 (low dword)	A1 (low dword)	A2 (low dword)	A3 (low dword)
A4 (low dword)	A5 (low dword)	A6 (low dword)	A7 (low dword)	A8 (low dword)	A9 (low dword)	A10 (low dword)	A11 (low dword)
A12 (low dword)	A13 (low dword)	A14 (low dword)	A15 (low dword)	A16 (low dword)	A17 (low dword)	A18 (low dword)	A19 (low dword)
A20 (low dword)	A21 (low dword)	A22 (low dword)	A23 (low dword)	A24 (low dword)	A25 (low dword)	A26 (low dword)	A27 (low dword)
A28 (low dword)	A29 (low dword)	A30 (low dword)	A31 (low dword)	A32	A33	A34	A35
A3, A2, A1, A0 (high byte)	A7, A6, A5, A5 (high byte)	A11, A10, A9, A8 (high byte)	A15, A14, A13, A12 (high byte)	A19, A18, A17, A16 (high byte)	A23, A22, A21, A20 (high byte)	A27, A26, A25, A24 (high byte)	A31, A30, A29, A28 (high byte)
B7	B6	B5	B4	B3	B2	B1	B0
C7	C6	C5	C4	C3	C2	C1	C0

2.3 State of the Art Approaches for Integrated GPU Performance Characterization

Research works on integrated GPUs are very scarse, and they have been highly in favor of exploring the micro-architectures and developing models or simulators, as presented in [4]. Other studies, such as [5] and [6], adopt machine learning techniques to estimate performance and propose such models as a high level substitute for lengthy architectural simulators, laying claim to the speed of the prediction. Fast and accurate predictions are paramount during the design phase for architects, a fact that sparks interest in the research of such models.

There have been some efforts in parallelizing specific sorting algorithms thought to take advantage of the parallelization potential of GPUs. These tend to be harder to implement than their CPU counterparts due to constraints on data sharing in GPUs. However, there have been success cases in achieving better performance than the CPU algorithms. One study in particular, [7], makes use of the locality of the integrated GPU, *i.e.* being on the same die as the processor and not having to go through PICe interfaces

Table 2.5: Relevant state of the art works regarding iGPUs.

Paper	Year	Discrete / Integrated / Simulator	Objective
[11]	2010	Discrete	Proposal of a power and performance model
[12]	2011	Discrete & Integrated	Comparison of discrete and integrated GPU architectures
[13]	2012	Discrete & Integrated	Comparison of an algorithm implemented in a discrete GPU vs integrated GPU
[9]	2014	Simulator	Study on memory system heterogeneous behavior
[14]	2014	Integrated	Proposal of scheduling algorithms
[6]	2015	Discrete	Development of machine learning models for performance and power estimation
[10]	2016	Integrated	Performance analysis of workgroup broadcast
[15]	2016	Integrated	Performance analysis of workgroup reduce
[8]	2016	Simulator	Study on cache effects on heterogeneous benchmarks
[5]	2017	Integrated & Simualtor	Development of machine learning models for performance estimation
[7]	2017	Integrated	Implementation of a sort algorithm in GPU
[16]	2017	Integrated	Analysis of heterogeneous memory management algorithms
[4]	2018	Simulator	Modeling and simulation of Intel iGPUs

to move the data between CPU and GPU, to avoid the higher latencies involved when accelerating problems using discrete GPUs. This technique has major weight in the usefulness of integrated GPUs versus discrete GPUs, as is discussed in [8].

Another type of research interest lies in the CPU and GPU collaboration and the heterogeneity of such systems. In [9], the difference in memory access patterns between CPU and GPU is shown, proposing that top level caches (L1 and L2) have a lessened impact in maximizing performance in GPU workloads, contrasting with the CPU, where the presence of a cache hierarchy is critical for performance. This happens due to the highly bursty nature of GPU memory accesses and low data reuse, benefiting more from a highly banked memory system than a latency hiding cache hierarchy.

There are few works that directly target Intel integrated GPUs. Of mention is the work proposed in [10], that evaluates the performance of work-group broadcast in Intel's iGPUs. Being a very architecture dependent workload, the broadcast implementation suffers from slight variations on work-group size.

These works tend to rely in state of the art simulators to test new algorithms or to micro-benchmark a given architecture. Works that study integrated GPUs and use real hardware are surprisingly quite rare. Hence, the research work condicted in this thesis aims to bridge the gap between the existing literature in discrete GPUs and the lessened amount of research work in integrated GPUs and by performing extensive performance evaluation and benchmark real hardware (as opposed to using simulators). Table 2.5 summarizes the most relevant research works in the spectrum of performance analysis in GPUs, some of which have been previously addressed. In the remaining of this section, a more detailed description of the most relevant research works is provided, which partially tackle the topics in this thesis.

The authors in [4] perform a thorough characterization of the Intel Gen9 micro-architecture. Several insights of the architecture are taken from this work, making it relevant to this thesis due to the similarities between Gen9.5 and Gen9 architectures. The work focuses on building micro-benchmarks in OpenCL to exploit the architecture, namely, benchmarks to achieve peak floating point performance as

well as to measure the latency in memory access patterns. The latter was performed by employing a pointer chasing algorithm. The results show that a work group size of 32 threads is enough to reach peak throughput, and that GPU latency is close to a hundred times higher than the CPU, which is to be expected given the GPU's focus on high throughput rather than low latency. Another remarkable study of this work is cache sharing effects between the CPU and GPU. For this they employ four different scenarios of memory access: 1) both CPU (one thread) and GPU (one thread) employ a pointer chasing workload, resulting in 20 CPU accesses in between every GPU access due to the different latencies of the two parties, average access time measured in the CPU side, 2) same scenario but access time is measured in the GPU side, 3) same scenario but CPU now runs with 4 threads increasing the interference 4-fold, which resulted in an average of 100 CPU accesses in between GPU accesses, average access time measured in the CPU side, 4) same scenario but CPU only runs on one thread and GPU spans multiple threads, access time measured on GPU side. Results from Scenario 4 show 10 GPU accesses in between every CPU access. From the GPU side, the results indicate that whenever the working set fits in the LLC, and given the GPU's access times seem unaffected by the CPU, the GPU access time is independent of the CPU and that the lower access times observed when increasing the working set come from limitations in the GPU only. A lack of bias towards the CPU from the LLC is also observed, given the GPU does not suffer in scenario 3, where a higher concentration of CPU accesses are occurring. Only DRAM contention is observed when the working set is increased to spill to DRAM. From the CPU side of things, for one threaded GPU activity, the CPU is hardly affected, which is to be expected given the higher ratio of CPU accesses relative to GPU accesses. In scenario 3, DRAM contention is again prominent, with low indication of GPU interference. For scenario 4 however, where the CPU to GPU access ratio is inverted (1 CPU access every 10 GPU accesses), the CPU access rates are visibly affected when the working set fits in the LLC. This leads to suggest eviction of CPU's data via GPU requests.

The work also studies memory coalescing when the GPU accesses the DRAM. To do this, they divide the memory region between work groups and each work item in a work group reads a constant size of bytes. The stride is varied across experiments, starting with a stride of 1. This ensures consecutive work items in a work group read consecutive bytes. This guarantees the access pattern gets coalesced into a single request. For a stride of 2, every second batch of bytes is read. For a stride of 16, each read operation of a work item targets a different cache line. Across the three access patterns, the total amount of data read remains the same. Bandwidth is measured with respect to cache lines read. Figure 2.6 shows the results obtained. For a stride of 1, as the number of work items and work groups increase, an increase in the bandwidth is observed. For larger strides, this effect happens sooner. With the increase of work items and work groups, a higher number of threads spawn, which will lead to more data requests and to more MLP (Memory Level Parallelism), which explains the increase in bandwidth. As the stride increases, for the same number of threads, more cache lines are being targeted, resulting in a higher number of requests. With this, the peak bandwidth is reached with a lower number of threads than with smaller strides.

The last of the benchmarks done by this work intends to measure MLP. For a given number of

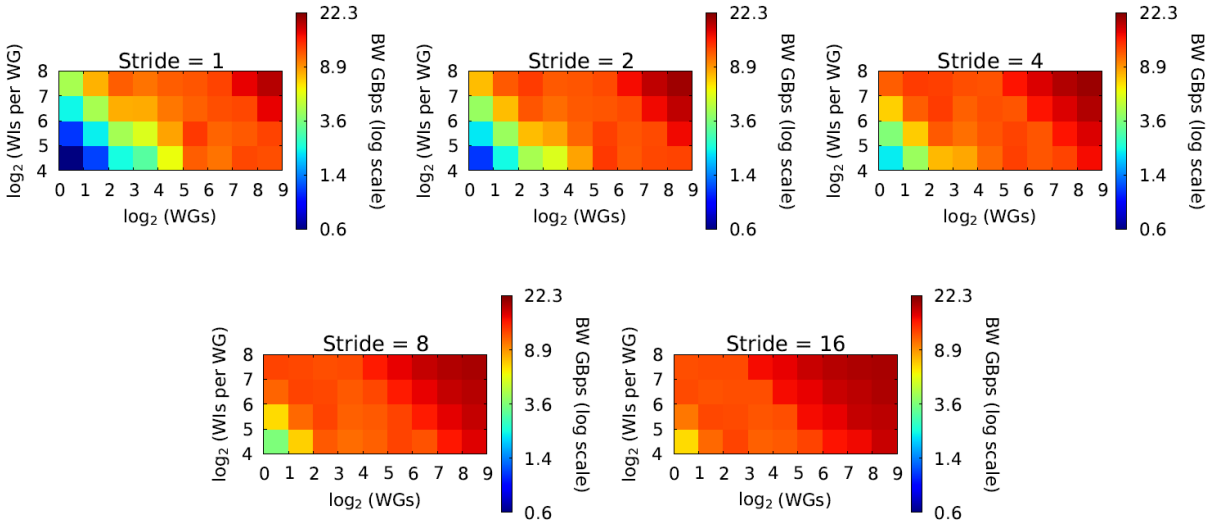


Figure 2.6: Raw memory bandwidth for independent memory accesses at different strides [4].

serviceable simultaneous memory requests supported by the system, a workload can hide latency if the number of requests does not exceed this value. Otherwise, it would suffer from higher execution time. In order to measure this, work groups with only one work item are spawned. The experiment then varies the number of work groups and the size of the data set worked on by each work group. The number of work groups varies until the maximum number of threads supported by the architecture is reached, while the size of the data set varies according to what level of the memory hierarchy the current test is aiming at. The following conclusions are derived from Figure 2.7. For a data set size of 64 B and 2 KB, while the data set fits in L3, there is no notable difference in performance. For the 1 MB data set, it starts out in the LLC, but spills into DRAM at the 8 work groups mark. Performance drops at this point and coincides with the 8 MB data set. For the 8 MB data set, that starts off in DRAM, performance remains constant until around the 100 work groups mark. This marks the MLP limit for the architecture, at 100 simultaneous requests.

The rest of the paper details the work of building a simulator for the Gen9 micro-architecture, designed to reach the hardware limits exposed by the previous benchmarks. This section falls behind the scope of this thesis, so it shall be skipped. The important conclusions made by this paper that are relevant to this thesis are the following architectural discoveries: EUs are capable of reaching peak floating point performance with just 4 threads; the GPU suffers higher latency than the CPU even when accessing shared resources such as the LLC; the GPU is unable to use the LLC's full capacity; the GPU is largely unaffected by CPU interference when accessing the LLC, while the CPU is affected by the GPU's accesses to LLC; finally, the GPU supports 100 memory accesses of MLP across the entire memory hierarchy. These developments, be they for Gen9, can be also used for Gen9.5 due to the similarities between the architectures.

In [6], a machine learning model is constructed to predict performance and power of a given workload in different GPUs. This work shares much with [5], however it is more geared towards GPGPU,while the latter has more focus on graphics processing. The model is constructed by running various kernels on different hardware configurations. A hardware configuration, in the scope of this paper, is a triplet of

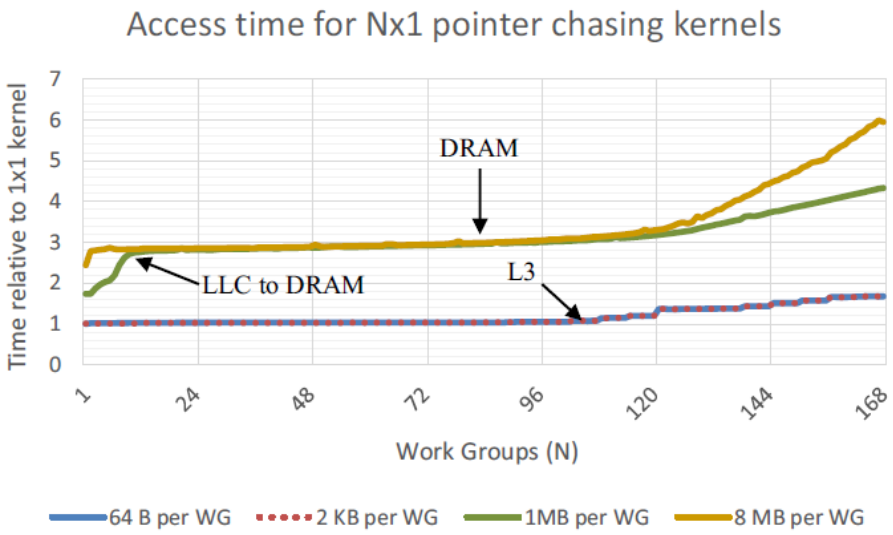
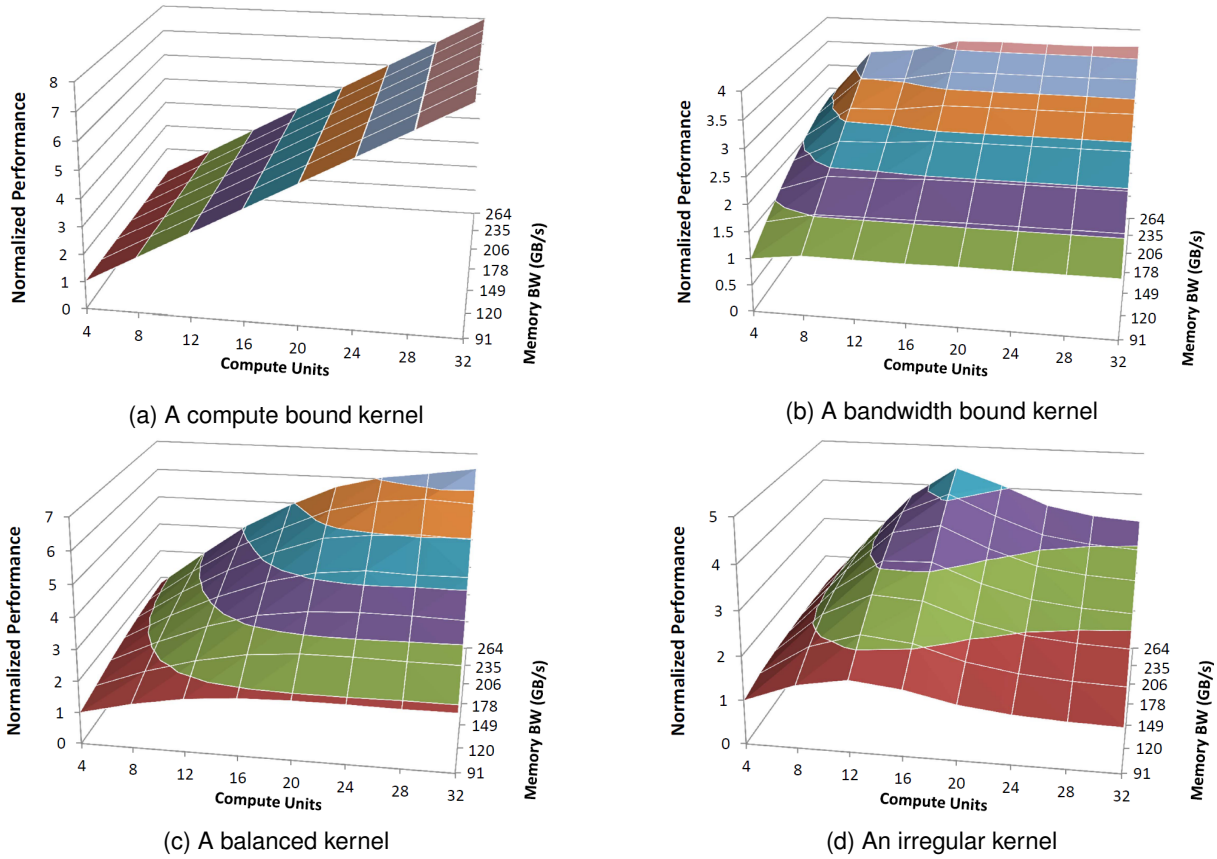


Figure 2.7: Relative access time for Nx1 kernels chasing pointers in different working sets [4].

values containing Compute Units (CU) count (a more general term for Intel's EUs), core frequency and memory frequency. For each kernel and a set of hardware configurations, execution times or power values are gathered, in regards to what model one is attempting to construct (performance or power, respectively). This, in addition to performance counter values gathered at what is called the base hardware configuration, constitute the training set. The training set serves as features to identify the kernel's scaling surface. A scaling surface can be of the types represented in Figure 2.8. They are used to predict how a given kernel will scale by varying any of the three underlying variables, CU count, core frequency or memory frequency. The first phase of the model training process involves the clustering of kernels according to their scaling surfaces. For this, a K-means algorithm is used. The second phase requires construction of a classifier with the aim to predict which of the clusters best describes the scaling behavior of a given kernel based on its performance counter values taken at the base hardware configuration. In sum, the classifier chooses a scaling surface for the kernel and the surface is used to predict how the kernel will perform with other hardware configurations. The classifier is constructed via multi-layered perceptrons.

The model can predict performance up to an average error of 15%, at a speed well above those of architectural simulators. Its worth is therefore situated on design space exploration, where the extreme cost time-wise of using cycle-accurate simulators makes them prohibitive.

In [15], after a brief overview of the Gen overall architecture and of the Gen ISA, two implementations of reduction in work groups are analyzed: variant MSG and variant SLM, in Intel's OpenCL open source implementation, Beignet. Figure 2.9 presents both variants of a reduce ADD operation. Both variants start by reading data from memory (INIT stage), with each thread holding a portion of the full array. Common to both variants is the first stage (S1), where each thread computes the local sum of its slice of the array. Focusing on variant MSG, in stage 2 (S2) thread ID 0 sends its local reduction to thread ID 1, which in turn will concatenate the value received with its own local reduction. Thread ID 1 is left with a local array of 2 values, where a second reduction is taken place. This leaves thread ID 1 with only one



(a) A compute bound kernel

(b) A bandwidth bound kernel

(c) A balanced kernel

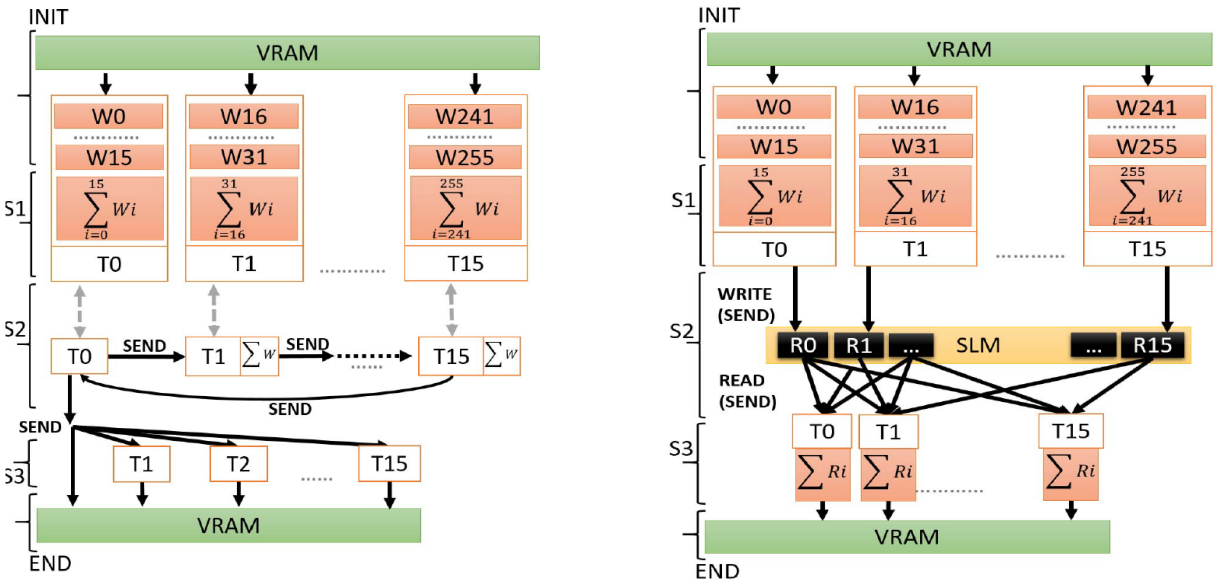
(d) An irregular kernel

Figure 2.8: Example of four distinct GPGPU performance scaling surfaces. Frequency is fixed while CUs and bandwidth vary [6].

value again, which it proceeds to send to the next thread. The process repeats itself until the last thread (thread ID 15 in Figure 2.9a) sends the last result of the reduction back to thread 0, which had been waiting with a *wait* instruction. Thread ID 0 then broadcasts the final value to all other threads (stage 3) and they proceed to write the result to memory (END). In regards to variant SLM, in stage 1 all threads write their intermediary result to SLM, followed by a fence to provide synchronism. This is necessary for in the next stage (S2) every thread will read the array from SLM and compute the final result individually (stage 3), as seen in Figure 2.9b. The threads then write their result to memory (END).

Results show that variant SLM achieves better performance. Tests were performed in two different systems, machine 1 and machine 2. The most notable difference between the two is the variation in Gen architecture, one being Gen7 and the other Gen8. The advantage of testing in different architectures lies in debunking the notion that a specific architecture benefits more from a particular variant. While the architecture certainly influences performance, given that one is the evolution of the other, they share enough characteristics so as not to suffer from entirely different results.

The performance advantage for the SLM variant can be explained by the fact that in the MSG variant, in stage 3, the threads are in an idle state except for one, all waiting to receive the previous thread's results, serializing the computation. Variant SLM on the other hand, makes better use of the parallelization by executing the partial reductions in parallel. The extra amount of work done by each thread in the SLM variety is amortized by not having to wait for the result to go through all threads in the work group.

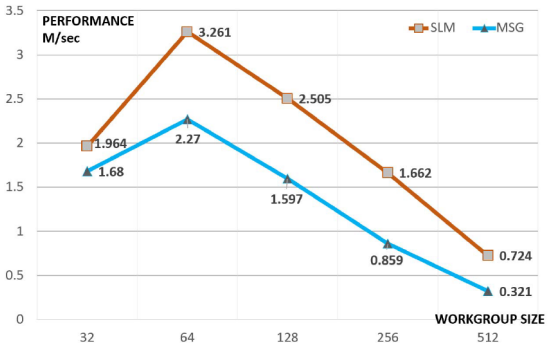


(a) Variant MSG, thread to thread message passing, work-group reduce ad.d

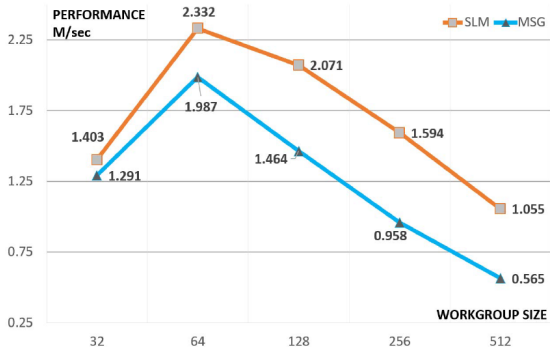
(b) Variant SLM, thread to shared local memory read/write, work-group reduce add.

Figure 2.9: OpenCL work group reduce algorithm. Two possible variants [15].

In [10], after an obligatory introduction to the Grn architecture, a comprehensive analysis of a broadcast at the GEN assembly level is given, followed by a performance analysis on the implementation of the work group broadcast by Intel's OpenCL open source implementation, Beignet. The broadcast implementation is described in four stages. The first stage deals with memory allocations. The work group size, allowed to be either one, two or three dimensional according the OpenCL specification [17], is internally flattened to a one dimensional allocation of the threads. This implies performance is independent of the work group dimension, given the same size. In each thread's GRF, there must be allocated space for the input set, temporary storage for intermediate operations and the output set. The Beignet implementation of broadcast uses SLM, so this must also be allocated in the GRF. The JIT compiler sets up the SLM address offset even before the broadcast function. The second step is to ensure data consistency. Before reaching the broadcast function, every hardware thread must have a proper SRC and DST register ready for the broadcast, and because of thread switching behavior, there is no guarantee that if a context switch were to happen, the broadcast would only start after all the data is safely stored in the registers. For this reason, a BARRIER is issued before broadcast execution. This ensures all registers are ready. The third step is to effectively write the value intended for broadcast into the SLM for posterior reads by the other work items in the work group. However, this is not a straight forward implementation. The OpenCL specification states all work items are to call the broadcast with the same set of local IDs. This requires a method in place to tell apart which of the work items is to write to SLM. This is done via the Gen ISA, by disabling all other channels except the one to write. The paper goes into the assembly and describes that a *cmp* instruction is used to modify the predicate in order to mask the channels other than the one to write. The ensuing *send* instruction will now only have effect on the unmasked channel. The final phase handles the reading of the SLM block by the other work items. A second BARRIER is issued to make sure SLM is read only after the previous write operation has ensued. The work items

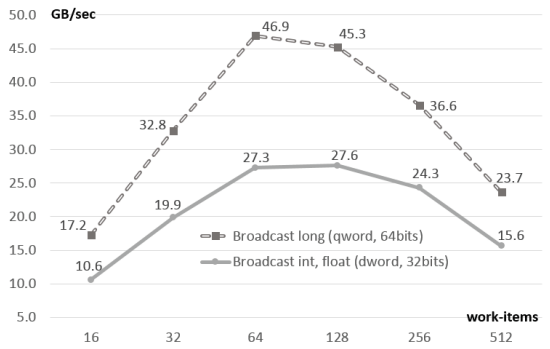


(a) Machine 1: Core i7 5600U @2.6ghz, GPU Intel HD 5500, Gen8 GPU, Broadwell GT2, with 8 GB DDR3 RAM.

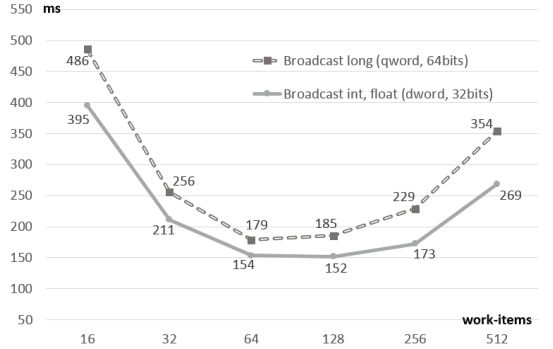


(b) Machine 2: Core i7 3770 @3.4ghz, GPU Intel HD, Gen7 GPU, Ivy Bridge GT2, with 8 GB DDR3 RAM.

Figure 2.10: Work-group reduce min performance, variants MSG vs SLM. Results are in Millions of reductions per second [M/sec] [15].



(a) Throughput of work group broadcast for various work group sizes, double word vs quad word.



(b) Latency of work group broadcast for various work group sizes, double word vs quad word.

Figure 2.11: Throughput and latency of work group broadcast for various work group sizes [10].

will have their DST register populated with the value read from SLM and this terminates the broadcast operation.

The paper ends with an experimental analysis of the broadcast implementation of Beignet. The test was run on an Intel HD 5500, which corresponds to a Gen8 Broadwell GT2 (8 EUs, 3 Subslices for a total of 24 EUs). The throughput can be seen stabilizing along the 1000 iterations mark, tending to 45 GB/s. A test run demonstrated that in order to hide the kernel launch latency overhead, the broadcast would need to be called in a loop with 1000 iterations. This treatment is replicated for all subsequent tests. Figure 2.11a shows the throughput difference between a double word value (32 bit) and a quad word value (64 bit) across various work group sizes. The peak is visible between 64 and 128 work items. Figure 2.11 shows the latency. Although using a bigger payload yields higher throughput, latency wise it is worse. This is due to the EUs instruction set, requiring 2 *send* instructions for the quad word broadcast. The interval with th elowest latency shares the work group dimensions observed for the throughput, between 64 and 128 work items. The paper goes to show that by eliminating the barriers of the broadcast implementation, latency drops substantially (50% to 75%), proving that barriers severely impact performance, but are in some algorithms necessary to maintain data consistency.

In [13], an FDTD algorithm that makes use of the integrated GPU for acceleration is proposed and

compared to an implementation that uses a discrete GPU, as well as a multi-threaded CPU one. The implementations make use of the DirectCompute API, a general purpose computing API provided by the DirectX framework. The paper starts by giving a run down of the FDTD calculation and of the architecture of the platform used to run the tests, an Intel i5-2500K processor equipped with the iGPU Intel HD 3000 and a discrete Nvidia GTS240. Results show that running the FDTD computation on the iGPU achieves roughly twice the performance of the discrete implementation. The speedup of the FDTD CPU implementation using the four cores available as well as the AVX instruction set provided a speedup over a naive CPU implementation similar to the one provided by the iGPU.

2.4 Summary

In this chapter the architecture of the Intel Gen9.5 GPU was laid bare. The architecture follows a hierarchy of Slice $\rightarrow$ Subslice $\rightarrow$ EU, where each level is an arrangement of units of the level below it. The Slice contains the L3 cache, and subslices are connected to the L3 cache via Data Port units. The Data Port units exchange data between the L3 cache and the EUs in each subslice. The EUs are effectively the computational units and are comprised of two FPUs for processing and a GRF for local storage. Each EU supports seven concurrent threads, with co-issue of up to four distinct threads.

The performance counters are introduced by discussing the Observation Architecture, the Performance Monitor Unit (PMU) equivalent for the GPU. The two available modes of performance counter reporting are introduced and the four available report formats are shown. Important non-counter registers that are also reported by the OA and the four report formats are the GPU_TICKS and TIME_STAMP registers, which will be used to time the OpenCL kernels, an important step for the power analysis later on.

Finally, the state of the art in integrated GPU research is addressed, stating several works that hinge on this topic. This revealed a lack of focus in integrated GPUs when comparing to the amount of research available for discrete ones. Furthermore, it was uncovered that a big slice of research works make use of simulators, not testing their findings in real hardware. The work done in this thesis comes therefore to fill in this gap, by providing results run on real hardware without relying on simulators, and providing a comprehensive performance and power analysis for the Gen9.5 architecture of Intel's integrated GPUs.

Chapter 3

Tool for Fine-grain Benchmarking of the Intel GPU Architecture

The need to configure and access the available counters to perform application characterization and experimentally evaluate the upper-bound capabilities of an Intel integrated GPU architecture, led to the development of a tool with the intent of aggregating the launching of a set of specifically developed OpenCL benchmarks and the configuration and reading of the counters. The developed tool relies on a set of API structures inherited from Intel GPU Tools [18] in order to provide the low-level access to the necessary registers to configure the counters through MMIO and sending commands to the GPU using the Message Interface. The tool also makes use of PAPI to acquire energy consumption (and consequently power) through the RAPL interface.

Although the tool is developed mainly for the Gen9.5 micro-architecture, given the similarities across different Intel Gen GPU architectures, in particular Gen9.5 and Gen9, the tool exposes counter acquisition framework, which is general enough to be used even for different Intel GPU micro-architectures. Furthermore, future micro-architectures can be supported if no significant changes to the counter configuration are made. A common trope of subsequent micro-architectures is the changing of register addresses, which can be solved with patches that update the hard coded register addresses to make the tool compatible.

3.1 Top-Down Architectural Description of the Developed Tool

A set of OpenCL kernels come integrated with the tool. These kernels are divided in groups according to what micro-architectural aspect of the GPU they aim to exploit. They can be used for evaluation of floating point performance, memory bandwidth or power/energy consumption upper-bounds for a given Intel GPU architecture. These will be better described in Chapter 4. The tool's general layout is presented in Figure 3.1. It starts by parsing the arguments and decoding the type of kernel to run. The type of kernel is important for memory initialization and correct argument setup for the launching of the OpenCL kernel. Subsequently, as is the case for all OpenCL applications, device and platform

identification takes place, as well as the compilation of the kernel itself.

Kernel parameters such as SIMD length and floating point precision are taken directly from the kernel's name. In order to correctly identify kernel parameters, memory allocated variable names are given a high degree of verbosity, as can be seen in Algorithm 1, which indicates the process of memory initialization. The greater verbosity facilitates debugging. The same treatment is done to GPU buffer creation, as seen in Algorithm 2. For a given memory region to be available to the GPU using OpenCL without using shared virtual memory space, one has to create and write buffer objects using the API calls *clCreateBuffer()* and *clEnqueueWriteBuffer()*, respectively. These API calls are common in every OpenCL application, as is *clSetKernelArg()* to specify the kernel arguments.

Algorithm 1 Memory initialization

1: **if** $scalar \in kernel\_name \wedge sp \in kernel\_name$ **then**
2: $allocate\ scalar\_sp\_A$
3: $allocate\ scalar\_sp\_B$
4: **else if** $vect2 \in kernel\_name \wedge sp \in kernel\_name$ **then**
5: $allocate\ vect2\_sp\_A$
6: $allocate\ vect2\_sp\_B$
7: **else if** $vect4 \in kernel\_name \wedge sp \in kernel\_name$ **then**
8: $(...)$
9: **end if**

Algorithm 2 GPU buffer creation

1: **if** $scalar \in kernel\_name \wedge sp \in kernel\_name$ **then**
2: $a \leftarrow clCreateBuffer(..., sizeof(float), ...);$
3: $b \leftarrow clCreateBuffer(..., sizeof(float), ...);$
4: $clEnqueueWriteBuffer(..., a, ..., sizeof(float), scalar\_sp\_A, ...);$
5: $clEnqueueWriteBuffer(..., b, ..., sizeof(float), scalar\_sp\_B, ...);$
6: **else if** $vect2 \in kernel\_name \wedge sp \in kernel\_name$ **then**
7: $a \leftarrow clCreateBuffer(..., sizeof(cl\_float2), ...);$
8: $b \leftarrow clCreateBuffer(..., sizeof(cl\_float2), ...);$
9: $clEnqueueWriteBuffer(..., a, ..., sizeof(cl\_float2), vect2\_sp\_A, ...);$
10: $clEnqueueWriteBuffer(..., b, ..., sizeof(cl\_float2), vect2\_sp\_B, ...);$
11: **else if** $vect4 \in kernel\_name \wedge sp \in kernel\_name$ **then**
12: $(...)$
13: **end if**

In order to avoid long compilation times, the tool keeps a record of the kernel binaries in separate files and reads the corresponding file instead of redoing the whole compilation process every time. The OpenCL source code for the kernels themselves are aggregated in files according to their type, which allows for a significant reduction in the time taken to read the source files and recompile the kernels for each run. These files contain intermediate binary code that is later fully compiled by the OpenCL JIT compiler into the code for the targeted architecture of the platform.

The OpenCL preliminary code, *i.e.*, device and platform identification, context creation and kernel compilation, is done via the API calls *clGetPlatformIDs()*, *clGetDeviceIDs()*, *clCreateContext()* and either *clCreateProgramWithBinary()* or *clCreateProgramWithSource()*, depending on which is available, the source or the pre-compiled binaries.These are essential for every OpenCL application and denote the common workflow of building OpenCL programs.

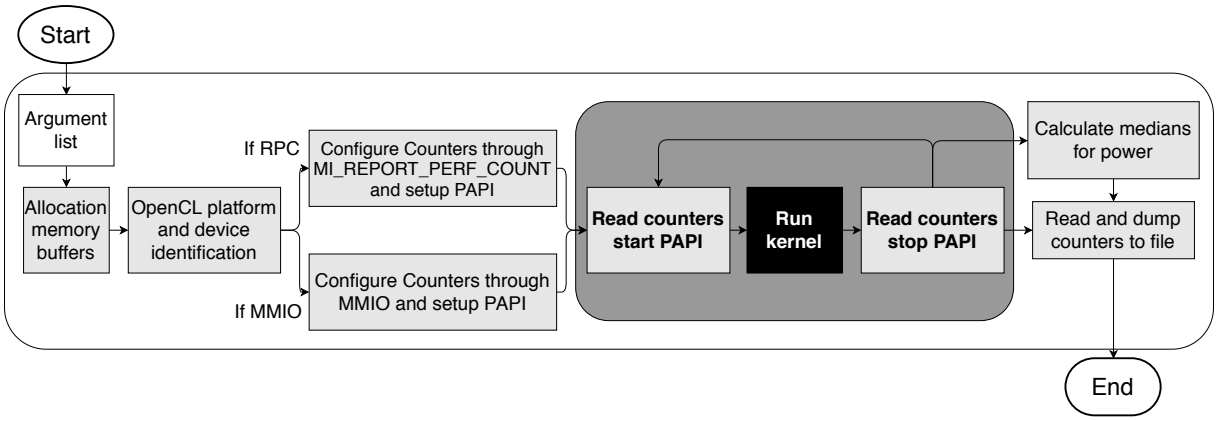


Figure 3.1: Developed tool general layout.

One of the arguments to the tool selects counter access procedure. There are two distinct access types, the *mmio* and the *rpc* type. The RPC type makes use of sending an MI_REPORT_PERF_COUNT message to the GPU to instruct it to write the counters in a previously allocated memory buffer accessible by both the GPU and the CPU. The other method employs the use of Intel GPU Tools' MMIO API to read the registers of the counters through MMIO. Regardless of the access type, the Intel GPU Tools' MMIO API is still needed to write to the specific registers pertaining to the counter configuration. This procedure will be more thoroughly described in Section 3.3.

The option to select the type of counter access mode delivers flexibility to the end-user. The *rpc* method is more direct in that it communicates with the graphics driver, while *mmio* interacts with an API before reaching the driver. No performance impacted overheads were detected with either access method.

The process of sending a message to the GPU is handled by the driver, and Intel GPU Tools provides an interface for easy message submission. The following are C macros provided by Intel GPU Tools to send messages to the GPU:

- BEGIN_BATCH
- OUT_BATCH
- OUT_RELOC
- ADVANCE_BATCH

BEGIN_BATCH sets up a batch (allocates a structure), while OUT_BATCH sends a double word into the batch. OUT_RELOC sends relocations into the batch, a necessary procedure for batches to be submitted to the validation list, which is a list of buffer objects kept by the driver. ADVANCE_BATCH issues the message created in the batch by the previous macros into the GPU, completing the command submission sequence. The issue of an MI_REPORT_PERF_COUNT message entails the code in Listing 3.1. Of note is the creation of a batch with three double words and one relocation. One of the double words required is the actual message ID of the message to be sent. Intel GPU Tools provides a list of macros with most of the available messages. There exist two different message IDs for MI_REPORT_PERF_COUNT,

the GEN6_MI_REPORT_PERF_COUNT and GEN8_MI_REPORT_PERF_COUNT. Caution is advised to issue the one corresponding to the architecture in play. For Gen8 and subsequent architectures, GEN8_MI_REPORT_PERF_COUNT is used, while GEN6_MI_REPORT_PERF_COUNT is used for previous architectures that support the OA unit. The first OUT_BATCH contains the message ID of an MI_REPORT_PERF_COUNT message in the form of the macro GEN8_MI_REPORT_PERF_COUNT. The OUT_RELLOC call specifies the target buffer and the read and write domains. These are information pertaining to the driver and go out of scope of this thesis. The last OUT_BATCH takes as argument a pointer to the memory location where the counters will be written to. Finally, ADVANCE_BATCH terminates the command submission and sends the batch with the message to the GPU.

Listing 3.1: Submission of MI_REPORT_PERF_COUNT.

```
BEGIN_BATCH(3, 1);
OUT_BATCH(GEN8_MI_REPORT_PERF_COUNT);
OUT_RELLOC(dst_bo, I915_GEM_DOMAIN_INSTRUCTION,
        I915_GEM_DOMAIN_INSTRUCTION, dst_offset);
OUT_BATCH(report_id);
ADVANCE_BATCH();
```

The MMIO API used by Intel GPU Tools exploits the aperture in the graphics card's onboard memory. This has been been mapped via a Base Address Register (BAR) to allow direct access from the CPU. The BAR can be retrieved via the PCI configuration space. The PCI configuration space is a set of registers that are mapped to memory locations, *i.e.*, a certain set of memory addresses are used to access these registers via the CPU. The GPU is regarded as a PCI device, so it has its own place in the PCI configuration space. PCI devices are addressed via Bus, Device and Function, known as the BDF or B/D/F. The Intel Integrated GPU is typically located in bus 0, device 2, function 0. By reading the PCI configuration space of BDF 0:2.0, one can retrieve the BAR that is to be added to the GPU's MMIO registers' addresses in order to access them.

The next step resides in the actual readings and kernel launch. The counters are read before and after the launching of the kernel. The difference between the two readings gives the correct values for the kernel. If the kernel is a power measurement kernel and the argument *"–power-smoothing"* is received, the tool will repeat this procedure a certain number of times to allow the power readings to stabilize. For kernels other than the power ones, this process would be necessary for the power readings to be trustworthy, for RAPL only updates every 50 ms [19]. This value, while not disclosed, was experimentally evaluated through repeated tests by varying the execution time of kernels. The aforementioned process repeats the kernel launch until the total time of kernel computation surpasses the minimum required time of 50 ms, in order to provide correct power values. However, this procedure is not necessary for power kernels since these have been coded specifically to take enough time to have several RAPL increments. The feasibility of this option relates to the fact that power readings are sensible and prone to noise, so by relaunching the kernels several times and calculating the median of all the readings eliminates noise and offers more repeatable results.

The final process rests with the reporting of the counter values, power, and various useful timing data. Timing data can be used to identify faulty kernel launches and errors in the power readings. Several timing sources are used due to the different clock domains where readings are being done. The counters themselves adheres to the GPU clock frequency while the power readings are done in the CPU side. The different timing sources are better discussed in Chapter 4.

3.2 User Interface of the Developed Tool

Interfacing with the tool is done via several command line arguments. Table 3.1 specifies all the most commonly used command line arguments exposed to the end-user. Of notable importance are $--$*blocks* and $--$*tpb*, choosing the number of work groups and work items, respectively. These should always be provided, for the default values are zero, which means no actual kernels will be launched. The $--$*build* option should be used whenever a modification to the provided kernels is performed, for these will only take effect if recompilation is redone. This is attributed to the fact that the tool uses precompiled binaries for the kernels to reduce the set up time required to read the kernel sources. This option will therefore recompile the kernel sources and update the binaries accordingly. The $--$*access* option specifies which access mode should be used with the tool when reading the counters. The possible modes are *rpc* and *mmio*, which correspond to reading the counters through the Memory Interface or via direct MMIO register reads, respectively. The difference between the two modes is described in detail in Section 3.1. The option $--$*enable*$-$*fma* indicates to tool to compile the kernels using the OpenCL flag $-$*cl*$-$*mad*$-$*enable*. This indicates the OpenCL compiler to use FMA instructions if the architecture supports them. This option should always be used in conjunction with $--$*build* when using the kernels with MAD operations. The option $--$*power*$-$*smoothing* should only be used with power kernels, however, since it only affects power kernels, it can be used for other kernels without prejudice. This option triggers a loop that repeats the kernel launch a set number of times, namely 10, and calculates the medians of the power consumption across the 10 launches. This value was experimentally acquired and results in a compromise between execution time and variance of the results. More runs imply less variance, which in turn corresponds to more reliable values, but longer execution times. The options $--$*custom*$-$*counter*$-$*increment*, $--$*custom*$-$*counter*$-$*coarse* and $--$*custom*$-$*counter*$-$*fine* allow the user to define a counter to read. The counters were programmed via a C macro in the code to allow more flexibility in the development stage. This option was therefore added to allow the user to change the counter without recompilation of the tool. The available counters are specified in Section 3.3. The option $--$*validate* was added to provide an insurance that the kernels were actually running to the end. With this option, the buffers allocated for the GPU are returned to the CPU and written to files to visually assess the calculations done. This mainly serves a debugging goal, and so is not very useful for the end user.

Table 3.1: Supported command line arguments for the tool.

Short	Long	Description
−d	−−dump	Write values to file
−r	−−repeat	Repeat runs
−c	−−choose	Choose kernel from list of available kernels
−b	−−build	Compile OpenCL kernels
−p	−−profile	Give timing for different sections of the code
−a	−−access	Choose counters access mode
	−−blocks	Number of Blocks (Work Groups)
	−−tpb	Number of Threads Per Block (Work Items)
	−−enable−fma	Enable FMA when compiling kernels
	−−power−smoothing	Repeat power kernels and calculate medians of results
−q	−−quiet	Disables writing to stdout
	−−external−app	Choose an external app instead of the provided kernels
−cc−i	−−custom−counter−increment	Supply own increment counter
−cc−c	−−custom−counter−coarse	Supply own coarse counter
−cc−f	−−custom−counter−fine	Supply own fine counter
	−−dump−kernel	Show kernels in stdout
−v	−−validate	Validate results using CPU

3.3 Counter Configuration

Counter configuration starts with writing to the OACONTROL register. Figure 3.2 shows the register layout with the discretized bits. In this register, bit 0, Performance Counter Enable, should be set to allow reporting of counters. On receiving an MI_REPORT_PERF_COUNT message with this bit cleared, the message will be dropped and behavior is undefined. The counters will not update except for the free running A counters A4, A6, A19 and A20. After setting this bit, three other registers must be configured to guarantee correct report triggers for context switch: OABUFFER, OAHEADPTR and OATAILPTR. The configuration for these registers will be addressed further on. Returning to OACONTROL, bit 1 is the Specific Context Enable. If this bit is cleared, then all contexts are eligible for counting, that is, the counting is done globally with no filtering. If the bit is set, only a single context will affect the performance counters. In previous architectures, namely Sandy Bridge, Haswell and their microfabrication process reduction counterparts, the context ID would have to be written to bits 31:12 of OACONTROL. In newer architectures, there is now a separate register for this end, the OACTXID. On context submission, software is expected to populate the OACTXID with the appropriate context ID. As is the register OACTXCONTROL, which offers additional functionality for the counters, such as the time stamp. The time stamp is a special counter that, as its name implies, can be used to count time. Its frequency is configurable and it can be paused and resumed. Bits 7:2 of the OACTXCONTROL register allow the setting of the TIMESTAMP counter's period according to the following equation:

$$StrobePeriod = MinimumTimeStampPeriod \times 2^{(TimerPeriod+\phi)}, \tag{3.1}$$

where ϕ corresponds to 4 for the Haswell architecture and to 1 for any other architecture. Bit 1 acts as Enable/Disable and bit 0 allows the pausing and resuming of the counting.

The next step in configuring the counters, would be to configure the remaining registers, OABUFFER,

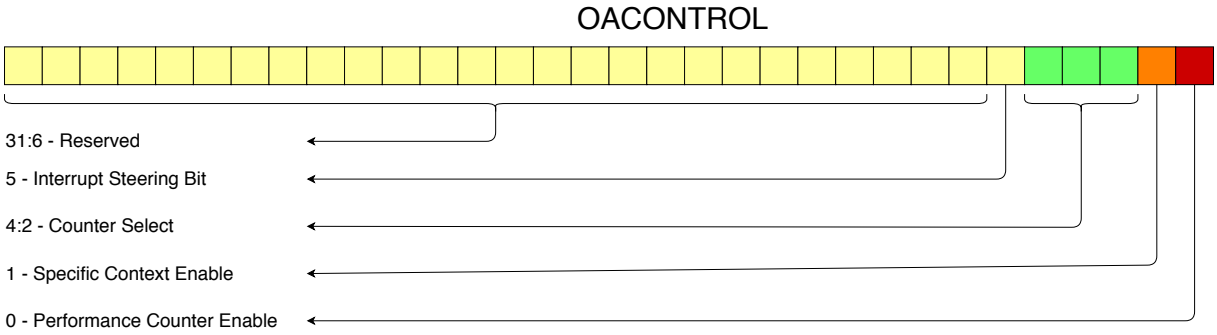


Figure 3.2: OACONTROL register bit layout.

Table 3.2: Translation table for Inter Report Buffer Size, bits 5:3 of OACONTROL register.

Values	Bytes
000	128 KB
001	256 KB
010	512 KB
011	1 MB
100	2 MB
101	4 MB
110	8 MB
111	16 MB

Table 3.3: Register EU_PERF_CNT_CTRi overview.

Bits	Description
31:24	Reserved
23:20	Fine Event Filter Select EU event i+1
19:16	Coarse Event Filter Select EU event i+1
15:12	Increment Event for EU event i+1
11:8	Fine Event Filter Select EU event i
7:4	Coarse Event Filter Select EU event i
3:1	Increment Event for EU event i

OAHEADPTR and OATAILPTR. There is a specific order to the configuration of these registers that needs to be followed. OAHEADPTR takes priority, and one must write into bits 31:6 the starting virtual address of the buffer that is to be used for the MI_REPORT_PERF_COUNT command. The second step is to configure OABUFFER by writing into bits 5:3 the Inter Report Buffer Size. This 3 bit value entails the size of the buffer in which the counters will be reported to, and complies with the mapping in Table 3.2. Bit 1 enables the overrun mode. When this mode is set, the counters get written at regular intervals defined by the *TimerPeriod* in OACTXCONTROL (bits 7:2). The last step in counter configuration would be to write the address of the end of the buffer into bits 31:6 of OATAILPTR. This process should conform to the following equation:

$$OATAILPTR[31:6] = OAHEADPTR[31:6] + convertBytes(OABUFFER[5:3]), \tag{3.2}$$

where *convertBytes()* should perform the translation specified in Table 3.2, *i.e.*, convert the bit encodings to the corresponding bytes.

With the counters enabled, it will be necessary to configure counters A7 to A20, *i.e.*, the programmable counters. These counters are programmable via another set of registers, the EU_PERF_CNT_CTR. There are 7 of these register and each one is used to configure a pair of the programmable A counters. They all share the same bit mapping. For $i \in \{0, 7\}$, the register EU_PERF_CNT_CTRi has the mapping displayed in Table 3.3. By combining the available filters, fine, coarse and increment, there is suddenly a vast amount of different metrics available. Tables 3.4, 3.5 and 3.6 depict the available options

Table 3.4: Counter configuration bits - Fine Event Filters [1].

Bits	Counter
0b0000	None
0b0001	Cycles where hybrid instructions are being executed
0b0010	Cycles where ternary instructions are being executed
0b0011	Cycles where binary instructions are being executed
0b0100	Cycles where mov instructions are being executed
0b0101	Cycles where sends start being executed
0b0111	EU# = 0b00
0b1000	EU# = 0b01
0b1001	EU# = 0b10
0b1010	EU# = 0b11

Table 3.5: Counter configuration bits - Coarse Event Filters [1].

Bits	Counter
0b0000	No mask
0b0001	VS Thread Filter
0b0010	HS Thread Filter
0b0011	DS Thread Filter
0b0100	GS Thread Filter
0b0101	PS Thread Filter
0b0110	TS Thread Filter
0b0111	Row = 0
0b1000	Row = 1

Table 3.6: Counter configuration bits - Increment Event Filter [1].

Bits	Counter
0b0000	EU FPU0 Pipeline Active
0b0001	EU FPU1 Pipeline Active
0b0010	EU SEND Pipeline Active
0b0011	EU FPU0 & FPU1 Pipelines Concurrently Active
0b0100	Some EU Pipeline Active
0b0101	At Least 1 Thread Loaded But No EU Pipeline Active
0b1000	Threads loaded integrator == maxthreads for current HW SKU

for each filter. Using, for example, fine event 0b0111 and coarse event 0b0111, its is possible to isolate events to one EU, and extrapolate later to the whole GPU. Further use of the increment event 0b0000, which counts cycles when one FPU is active, *i.e.*, when it is executing a floating point instruction, a measure of floating point instructions and, eventually, FLOP can be achieved.

In essence, most combinations of the programmable counters only allow to count cycles when a specific event is taking place, in a particular locale (*e.g.*: FPU pipeline, 3D media pipeline). This is a limitation of the Observation Architecture.

3.4 Summary

In this chapter, the tool was introduced, along with the counter configuration workflow. The tool's most standing out feature is the availability of two distinct modes of reading the counters, one using the Intel GPU Tools' MMIO API to directly access the registers were counter values are stored internally, the other using messages / commands to the GPU to force the recording of the counters to a previously identified region of memory. The tool's mode of operation was also scrutinized, giving emphasis on how the reading is done, in between OpenCL kernel launches.

Unlike the reading of the counters, counter configuration is done exclusively via MMIO, for direct access to the registers is required. Namely, the register OACONTROL handles most of the configuration, with the rest of the control registers being handled by the driver without intervention from the user. However, the most useful counters, A7 to A20, are programmable, and this task must be handled by the user. To program these counters, the registers EU_PERF_CNT_CTR0 to EU_PERF_CNT_CTR7 are

used. Each register controls two programmable counter and offer a wide range of programmability. A programmable counter is basically controlled by three sets of bits from the respective EU_PERF_CNT_CTR register, and the user needs to set the correct bits to turn on a given counting.

Chapter 4

Micro-Benchmarks and Experimental Evaluation

The micro-benchmarks consist of simple OpenCL kernels designed to fully exploit the capabilities of the architecture. In order to do so, and taking into account its SIMD capabilities, each benchmark has five different versions explicitly specifying the SIMD data type being exercised. These versions are *scalar*, *vect2*, *vect4*, *vect8* and *vect16*. There are also versions for each operation available, *i.e.*, ADD, SUB, MUL, DIV and MAD. With a total of 5 SIMD types and 5 operations each kernel has 5 $\times$ 5 = 25 versions.

All benchmarks were run on the same platform, an Intel Core i7-7500U with 8 GB DDR4 DRAM. The CPU is made up of 2 cores and 4 threads through hyperthreading, and has a stock frequency of 2.7 GHz. To run the results the hardware prefetch, hyperthreading and turbo boost were turned off, and the GPU frequency set to 1050 MHz. These are summarized in Tables 4.1 and 4.2.

Table 4.1: iGPU hardware features.

Features	HD Graphics 620
EUs	24
Threads / EU	7
SIMD FPU / EU	2
Max Frequency	1050 MHz
Slices	1
Subslices	3
L3 Cache	768 KB
L3 Cacheline	64 B
SLM / Subslice	64 KB

Table 4.2: CPU hardware features.

Features	i7-7500U (KBL)
Cores	2
Threads	4
Max Frequency	3.50 GHz
L1 Data Cache	32 KB
L2 Cache	256 KB
L3 Cache	4 MB
Cacheline (all levels)	64 B
DRAM	8192 MB

4.1 Intel GEN Assembly

In order to fully grasp what was being executed on the GPU, it was necessary to decode the exact instructions produced by the OpenCL JIT compiler. The Eclipse plugin from Intel OpenCL SDK allowed the visualization of the OpenCL kernels' GEN Assembly.

The ISA of the Execution Units differ greatly from the CPU's x86 ISA. A GEN instruction follows the form

[(pred)] opcode (exec-size | exec-offset) dst src0 [src1] [src2]

where pred reports the Predicate Enable bits of the instruction; exec-size corresponds to the SIMD width of the instruction, *i.e.*, the number of data elements processed by the instruction, which are called channels; , exec-offset to the Channel Select bits of the instruction that, together with the Predicate Enable bits, possess key roles in lowering branch prediction overheads; dst to the destination operand and src{0-2} to the source operands. Sources 1 and 2 are optional. The channel select bits select which channels are active when reading the ARF, while the predicate bits enables or disables specific channels in a single instruction. Together they are used to disable channels functioning as a commit stage by not allowing specific channels to write their results to the GRF. This is mostly used when the effective data width is less than that of the functional units (available FPUs are 4-way SIMD) or as a branch control mechanism, disabling channels that diverge from the rest of the workflow.

Source and destination operands conform to a register region syntax that follows the form:

RegNum . SubRegNum $<$ VertStride; Width, HorzStride $>$:type

where RegNum correspond to the region number, SubRegNum to the subregion number, VertStride to the vertical stride, which is the distance between two rows, in units of the data type, Width to the width, that corresponds to the number of units of the data type to be fetched per row, HorzStride to the horizontal stride, which is the distance between units of the data type in the same row, and type to the data type. The region and subregion numbers together define the origin of the data, while data type will provide the size of each unit of data. Figure 4.1 contains the general layout of the GRF. In the top rigt corner of the image there are expressed the possible subregion distributions according to the data type. For a byte (:b) data type, a subregion is one byte wide, for a word data type (:w) , a subregion is 2 bytes wide, for a double word (:dw) or float data type (:f) a subregion is 4 bytes wide. This subregion dependency on the data type offers a high degree of flexibility in register addressing. The rest of the figure provides two full examples of register addressing via regions 1 and 2. Region 1 is obtained via the following addressing: r5.1$<$16;8,2$>$:w . The region number is 5, meaning addressing starts in register 5. The *type* being word means the addressing unit is 2 bytes, therefore, subregion 0 corresponds to the 0th and first bytes, subregion 1 to the second and third bytes, and so on. An horizontal stride of 2 means to target every second unit in the register. A width of 8 means 8 units in a row, and a vertical stride of 16 will make the addressing fetch the second row, reaching into r6, for the subregion that is 16 units apart from the first subregion will be in the next row. The second example, region 2, comes from the following register addressing: r8.0$<$1;8,2$>$:w . By following the same logic as in region 1, with the region number being 8 and subregion number being 0, addressing starts in register 8 in th every first unit. Units

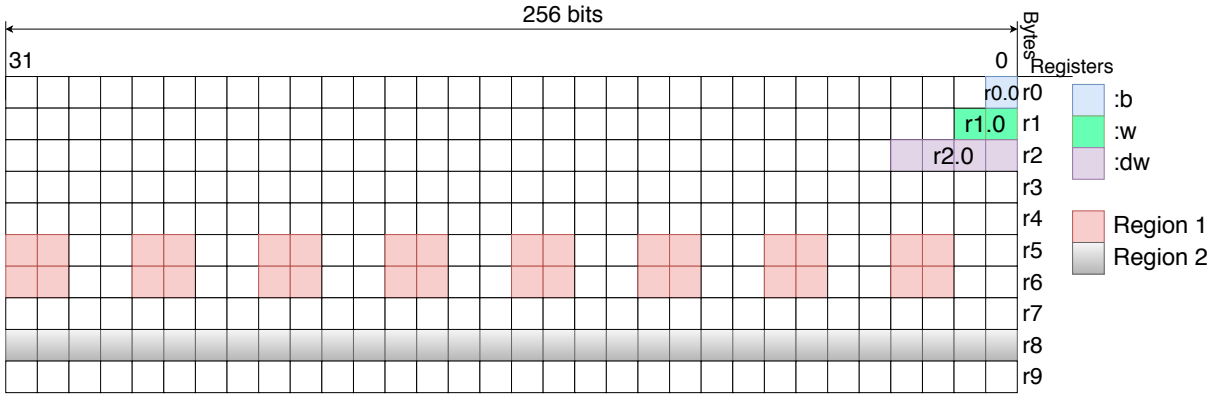


Figure 4.1: Register File layout.

Table 4.3: Available ARF registers [20].

Register Name	Register Count	Description
null	1	Null register
a0.#	1	Address register
acc#	10	Accumulator register
f#.#	2	Flag register
ce#	1	Channel Enable register
msg#	32	Message Control register
sp	1	Stack Pointer register
sr0.#	1	State register
cr0.#	1	Control register
n#	2	Notification Count register
ip	1	Instruction Pointer register
tdr	1	Thread Dependency register
tm0	2	TimeStamp register
fc#.#	39	Flow Control register

are words, therefore, the first two bytes of the register compose subregion 0. With the horizontal stride being 2, every second unit will be fetched, up until 8, the width. The vertical stride being 1, which is less that the horizontal stride, means that, unlike the first example, where the rows were spaced 16 units, so the fetch spilled into the next register, in this example the fetched units will be the interleaved units not yet touched in the same row, since the subregion 1 unit apart from the i-th subregion will be subregion (i+1)-th, which stands on the same row. The register addressing is, therefore, extremely flexible, allowing for complex data fetching patterns.

The kernel *scalar_sp_mad_kernel* shall be used as an example to demonstrate the compilation process into GEN Assembly. Figure 4.2 shows the OpenCL code and the generated assembly side by side. Color coded are the regions corresponding to each piece of code. A kernel using MAD instructions was chosen to better differentiate between the actual workload and the address calculation. The first three instructions are common to all kernels, and correspond to the argument fetching and the setting up of the Control Register (cr0.0). The Control Register is part of the ARF, the set of registers that keep track of the state of each thread. This register in particular handles floating point rounding modes and exceptions. The full list of ARF registers are displayed in Table 4.3.

The call to *get_global_id()* generates a *mul* instruction, which can be a source of unexpected over-

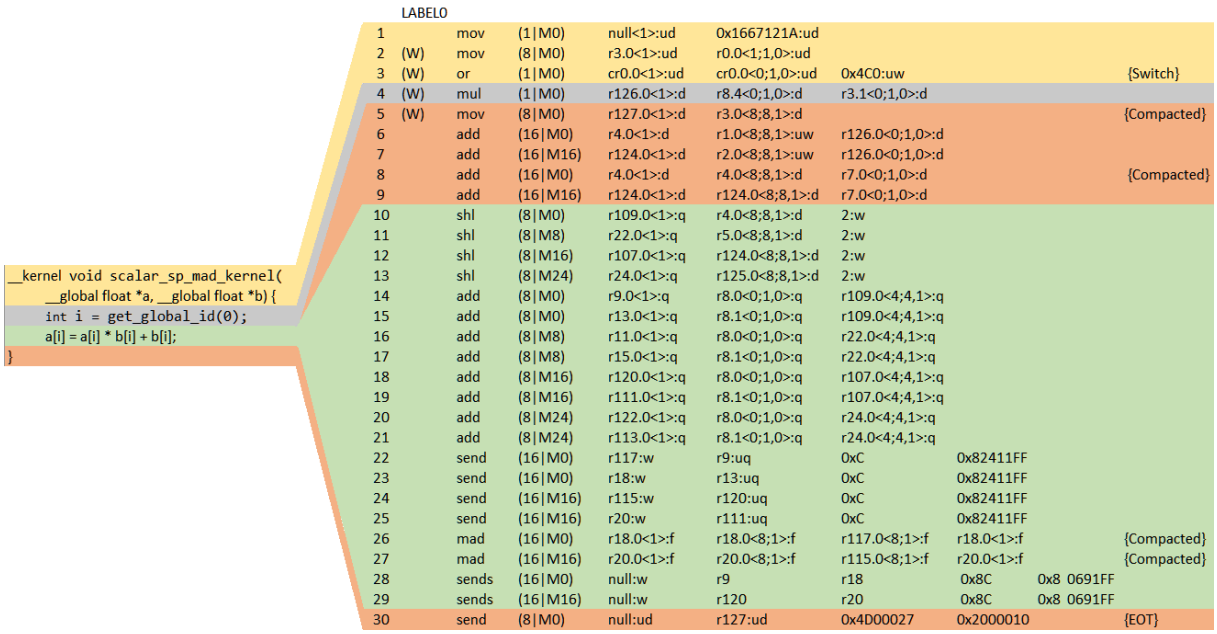


Figure 4.2: OpenCL kernel compilation to GEN Assembly.

head. However, the highest source of overhead comes from the address calculation as seen in lines 10-21. Following come the send instructions that can be seen as load instructions. Every memory operation needs to come from a *send* instruction, for the EU itself does not have the ability to access memory. The Send unit in the EU constructs messages to send to the Data Port unit in the subslice, and the Data Port unit is the one that effectively handles data traffic between memory and the EUs. The actual computation is done in lines 26 and 27, followed by the stores (*sends* instructions).

Lines 5 to 8 together with line 30 compose the piece of code necessary to terminate the thread. A thread always has to be terminated by a *send* instruction with the End-of-Thread bit set. This is made explicit in the code via the {EOT} annotation. As soon as the Thread Dispatcher receives the EOT message, the resources of the thread are freed and made available for other threads.

Some instructions are annotated with {Compacted}. This means the instruction, when compiled into machine language, will have a size of 64b instead of the usual 128b. This is useful to compress the binary file which will result in reduced instruction fetches. This in turn corresponds to less misses in the instruction cache, which leads to better performance and better cache data efficiency. On the other hand, the *or* instruction on line 3 comes with the {Switch} annotation. This means the current thread's instruction queue is flushed immediately after the annotated instruction, followed by a thread switch. This is used to prevent race conditions and is observed for all kernels. Not to be confused with compressed instructions. Compressed instructions are converted into two separate instructions by the Thread Dispatcher, thus doing double the work of a single regular instruction. They achieve the same end goal as compacted instructions, that of lowering code size.

4.2 Intel GPU Compute Performance Characterization

The peak floating point performance of the architecture, given its characteristics presented earlier in Table 4.1, can be calculated via the following equation,

$$FP_{max} = EU \times FPU/EU \times FLOP/cycle/FPU \times frequency, \tag{4.1}$$

where EU stands for the total number of EUs (24 for the current configuration), FPU/EU stands for the number of FPU units in each EU (2 in this case), FLOP/cycle/FPU stands for the floating point operations per cycle per FPU, which is 4 due to the four-way SIMD FPUs and the operating frequency in Hertz (1050 MHz), which amounts to 201.6 GFLOP/s for normal single precision floating point operations. For MAD single precision operations, this is two-fold given the architecture's ability to execute a *mad* operation in one cycle, corresponding to 403.2 GFLOP/s. For double precision, the throughput of the FPU is halved, and given the fact that only one of the FPUs are capable of executing double precision instructions, the peak floating point performance drops four fold from 201.6 GFLOP/s to 50.4 GFLOPS/s and from 403.2 GFLOP/s to 100.8 GFLOP/s for *mad* operations. For the remainder of this section, only results from operations *add* and *mad* will given, due to the similarities of the results between *add*, *sub* and *mul*.

The FPU benchmark kernels are divided into 4 types: single instruction, loop, loop with unroll and loop with private memory. The single instruction kernels are mostly used to observe overheads and to debug OpenCL related events. Their structure can be seen in Algorithm 3, where OP stands for a MAD operation or one of the standard operations: addition, subtraction or multiplication. Since the kernel is constituted by a single operation, each thread lacks the necessary workload to reach peak performance. The loop (Algorithm 4) and loop with unroll (Algorithm 5) were the first attempts at achieving high performance. The loop with unroll variant differs from the loop by having an explicit unrolling of 64 instructions. The automatic unrolling employed by the OpenCL JIT compiler varies in depth across SIMD lengths, raging from 32, to 64 or 128. Although these kernels achieve higher performance than the single instruction kernel, they still fall short of the peak performance. Furthermore, given the explicit vectorization of the OpenCL JIT compiler, the two versions do not exhibit a notable difference in performance. The loop with private memory, Algorithm 6, achieves the best performance out of all FLOP type kernels. Contrary to the other types, the use of private memory, *i.e.*, GRF, minimizes not only the memory operations but also the data packing/unpacking and address calculation, allowing for higher throughput of compute instructions and reducing the impact of memory latency. The per-thread value of the input vector (vector a) is loaded preemptively into the register file, where the computations will take place. In order to further reduce memory operations, contrary to the previous kernels where two vectors were used, this version relies on only one vector and computations are done with a constant term. Some care went into choosing the constant term. Given that the memory is initialized with the value 2.0 for vector a and 3.0 for vector b, the first iteration of the algorithm, with 2.0 and 1.0 as the constant terms for the *mad* operation, the per-thread value $a[i]$ would reach the value Inf within a low iteration count. This creates a floating point exception and causes a bypass in the computation, leading to low

Algorithm 3 Simple kernel algorithm

1: $i \leftarrow get\_global\_id(0)$
2: $a[i] \leftarrow a[i]\ OP\ b[i]$

Algorithm 4 Loop kernel algorithm

1: $i \leftarrow get\_global\_id(0)$
2: **for** $j < N$ **do**
3: $\quad a[i] \leftarrow a[i]\ OP\ b[i]$
4: **end for**

Algorithm 5 Loop-unroll kernel algorithm

1: $i \leftarrow get\_global\_id(0)$
2: **for** $j < N/unroll$ **do**
3: $\quad a[i] \leftarrow a[i]\ OP\ b[i]$
4: $\quad (...)$
5: $\quad a[i] \leftarrow a[i]\ OP\ b[i]$
6: **end for**

Algorithm 6 Floating point kernel using private memory

1: $i \leftarrow get\_global\_id(0)$
2: $temp \leftarrow a[i]$
3: **for** $j < N$ **do**
4: $\quad temp \leftarrow temp\ OP\ Constant$
5: **end for**
6: $a[i] \leftarrow temp$

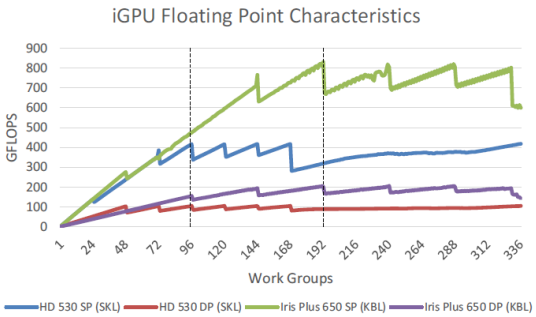


Figure 4.3: Performance comparison of HD Graphics 530 with 24 EUs integrated in a SkyLake processor and HD Graphics 620 with 48 EUs integrated with a KabyLake processor. Work Groups are comprised of 32 Work Items each [4].

execution times due to the reduced floating point operations. To circumvent this issue, the constant term was changed to 0.5 and 1.0 for the *mad* operation. Using *mad* operations with these values results in a convergent series independent of the value of *temp*, as shown by the following equations,

$$\begin{aligned} f^{(n)} &= f^{(n-1)} \times 0.5 + 1.0 \\ f^{(1)} &= temp \times 0.5 + 1.0 \\ \lim_{n\to\infty} f^{(n)} &= 2.0 \end{aligned} \tag{4.2}$$

where $temp \in \mathbb{R}_0^+$ and n stands for the iteration increment index (corresponds to j in Algorithm 6). This combination of constant values ensures computations are performed by all iterations of the loop.

An iteration count of 1024 ($N = 1024$) is not enough to replicate the serrated performance curve shown in Figure 4.3 from research work [4]. Even though 1024 iterations is enough to reach close to peak performance, it is only possible with a considerable number of work groups. By raising the number of iterations to 8192, the serrated effect becomes visible, as seen in Figure 4.4. This is explained through the process described in Figure 4.5, where the number of work groups needs to be a multiple of the number of subslices in order to achieve peak performance. This is also assuming the number of work items in a work group are enough such that the EUs in a subslice are given enough workload

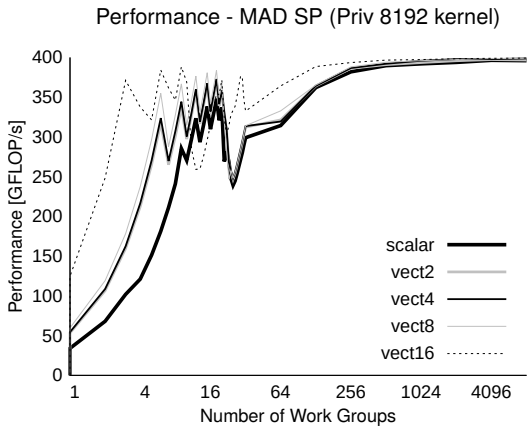
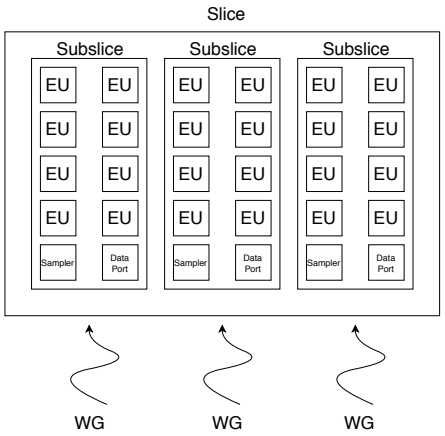
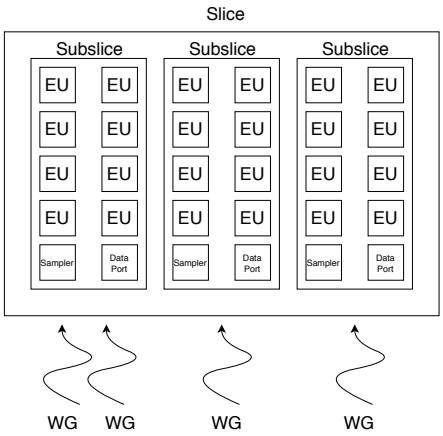


Figure 4.4: Performance comparison between the different SIMD lengths for the FLOP kernels using private memory and with 8192 iterations.



(a) Work group to subslice matching with the number of work groups being a multiple of 3.



(b) Work group to subslice matching with the number of work groups not being a multiple of 3.

Figure 4.5: Work group to subslice matching. When the number of work group is a multiple of the number of subslices, maximum performance is achieved. When this is not the case, one of the subslices will have a work imbalance compared to the others, which results in a higher execution time and less overall performance.

to achieve peak throughput. The OpenCL runtime reports 32 as the preferred work group size, and [4] achieves peak performance using this number. However, any work group size multiple of 32 is expected to deliver enough of a workload to the EUs so that peak performance for the EUs is achieved while simultaneously avoiding a workload imbalance such as the one evidenced by Figure 4.5 for the subslice. Figure 4.4 shows reaching peak performance with a work group size of 256, the maximum supported by the platform's OpenCL runtime. This performance curve is also shared by the power kernels, which their only difference with the flop kernels is the addition of an outer loop as a coarse way to control execution time. Their performance curves can be seen in Figure 4.6. In both performance and power kernels, as the SIMD length rises, high performance is reached with a lower number of work groups. This is expected since with the raising of the SIMD length, the data set size also rises, giving the GPU more work with a lower work group count. The serrated effect is clearly visible, with peaks at configurations of work groups multiple of the subslice count (3, 6, 9, and so on) and valleys for work group numbers in

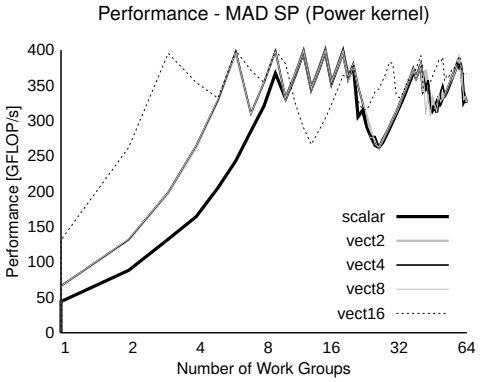
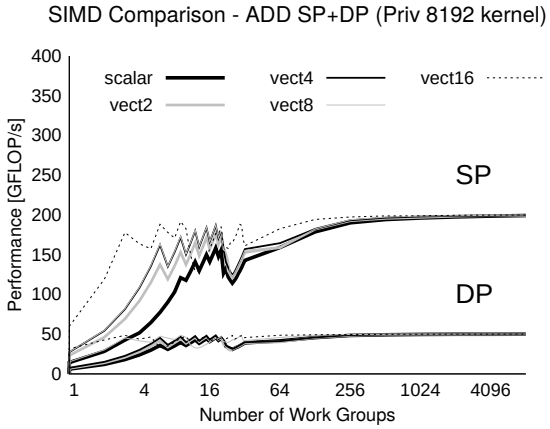
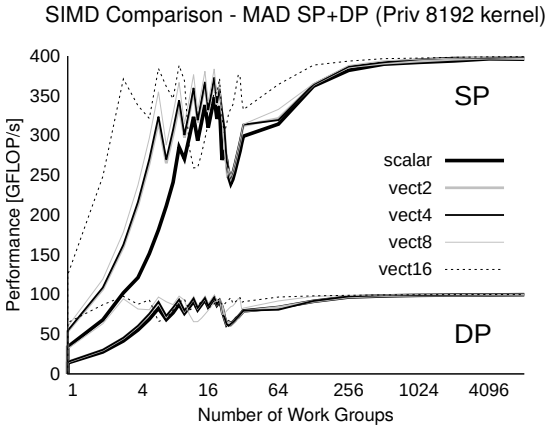


Figure 4.6: Performance comparison between the different SIMD lengths for the power kernels.



(a) Performance comparison of the *add* operation across different SIMD data types types for single and double precision.



(b) Performance comparison of the *mad* operation across different SIMD data types types for single and double precision.

Figure 4.7: Effects of different SIMD data types for the *add* and *mad* operations.

between said multiples.

Figure 4.7 shows the performance curves attained for the *add* and *mad* operations for different SIMD data types and for both precisions. As is expected, double precision performance rests at a quarter of the single precision performance for both types of operations.

By configuring register EU_PERF_CNT_CTR0 with the value 1904, which relates to Select Fine Event Counter 0b0111, Coarse Event Counter 0b0111 and Increment Event Counter 0b0000, counter A7 is programmed with counting cycles whenever the FPU0 pipeline is actively performing an instruction. Using this value, the FLOP/cycle metric can be calculated. Figure 4.8 shows the FLOP/cycle for the *add* and *mad* operations for both precisions. The performance kernels reach 7.9 and 15.9 FLOP/cycle for *add* and *mad* operations respectively, coming close to the theoretical 8 and 16 peak performance. The noteworthy bump in performance occurring between the 256 and 768 threads mark is due to the activation of the remaining subslices. For up to 256 work items, only one work group was being used. The rise in performance happens when two work groups of 256 work items are used, and again when the third work group starts being used, by now exercising all the subslices in the GPU. The SIMD data types's curves are collapsed as one given they share the same FLOP/cycle.

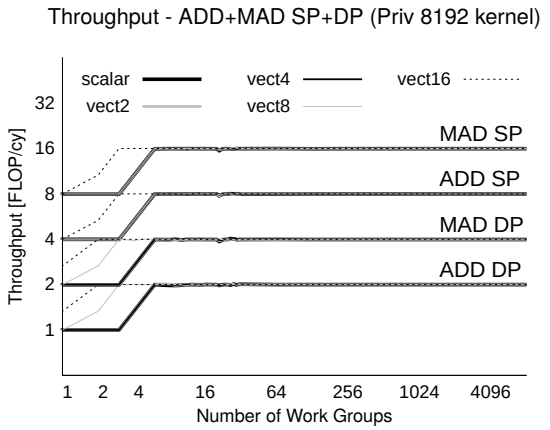


Figure 4.8: Comparison of FLOP/cycle between SIMD data for the *add* and *mad* operations for single and double precision.

4.3 Intel GPU Memory Bandwidth Characterization

Memory benchmarks are constructed much the same way as the FLOP benchmarks, as can be seen by Algorithm 7. Contrary to what is observed in the FLOP benchmarks, the SIMD data types heavily influence the attainable bandwidth for the memory kernels. Figure 4.9 shows the bandwidth obtained with the various SIMD data types. Due to overheads in the address calculation and data packing/unpacking, higher SIMD data types such as *vect16* and *vect8* fail to deliver acceptable performance, with *vect16* falling behind *scalar*. The best performant of the SIMD data types is *vect4* reaching the closest to the theoretical maximum bandwidth.

The same serrated effect perceived in the flop kernels are visible in the bandwidth curves. The peaks occur at work group sizes multiples of the number of subslices.

In order to reach peak bandwidth, SLM should be used. Although latency-wise, SLM and the L3 cache are equivalent due to the limit on the Data Port unit, the SLM, being highly banked, allows for greater diversity in access patterns. Access patterns on the L3 cache have a great influence on the attainable bandwidth.

Algorithm 7 Memory benchmark kernel algorithm

1: $i \leftarrow get\_global\_id(0)$
2: **for** $j < N$ **do**
3: $\quad dst[i] \leftarrow src[i]$
4: **end for**

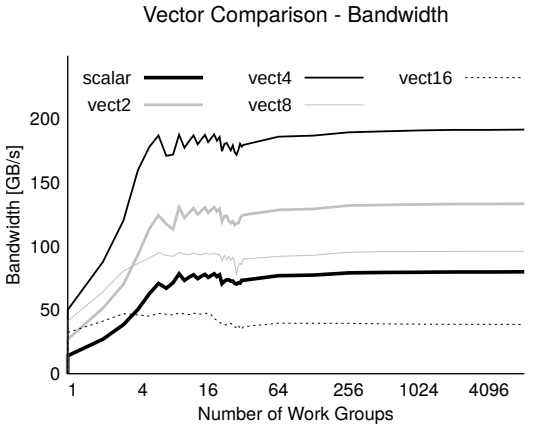


Figure 4.9: Attainable bandwidth with the standard memory kernel for different SIMD lengths.

Algorithm 8 Power kernel

1: $i \leftarrow get\_global\_id(0)$
2: $temp \leftarrow a[i]$
3: **for** $k < M$ **do**
4: **for** $j < N$ **do**
5: $temp \leftarrow temp\ OP\ Constant$
6: **end for**
7: **end for**
8: $a[i] \leftarrow temp$

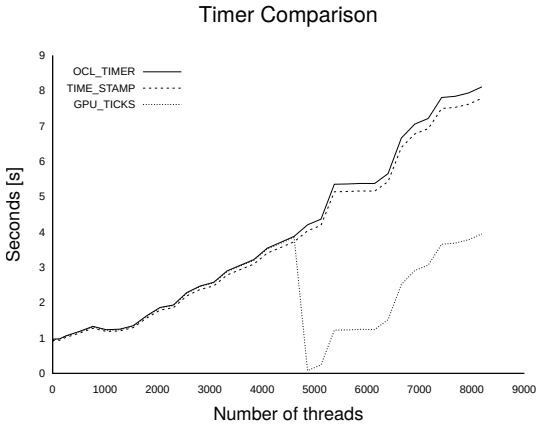


Figure 4.10: Comparison of the different timers available.

4.4 Intel GPU Power Benchmarks and Analysis

Power measuring in the tool is supported through PAPI, the Performance API. PAPI is used to read the RAPL system registers. RAPL is a system implemented in recent Intel processor architectures to provide energy readings. RAPL does not directly measure energy consumption, instead it supports a predictive model that outputs energy consumption based on other activity metrics collected internally [21].

Since RAPL only outputs energy values, in order to correctly measure power in the GPU a stable source of time measurement tied to GPU cycles was needed. Three sources were considered appropriate for this: through the OpenCL API, reading the GPU's TIMESTAMP register or via the GPU's GPU_CYCLES register. Using the power benchmarks, a slew of tests were carried out to establish which of the GPU clock sources to use for the power measurements. Figure 4.10 demonstrates the differences of the three time sources tested. While they appear closely in sync, around the 5000 threads mark, the GPU_CYCLES time source drops abruptly. This effect has been confirmed in software to be caused by an overflow of the register. Regarding the other two sources, a slight difference between them is noticeable. This can largely be attributed to the OpenCL profiler's overhead, although its contribution to the results is negligible.

RAPL allows reading energy consumption from three planes: Package, Power Plane 0 (PP0) and DRAM. The Package plane corresponds to the entire processor and includes PP0 and DRAM. PP0 corresponds to the cores. Measures for the iGPU were calculated from the other planes using the following equation:

$$P_{iGPU} = P_{PKG} - P_{PP0} - P_{DRAM}. \tag{4.3}$$

The power kernels mainly consist of modified FLOP kernels, as seen in Algorithm 8. The inner loop makes up the workload, while the outer loop is a coarse way of maintaining similar execution times between SIMD lengths (M varies with the SIMD length). The outer loop is required to provide enough execution time for an update of the RAPL counters. In order to better control the duration of the kernels, the number of iterations of the outer loop is dependent on the SIMD length. With the rise of the SIMD

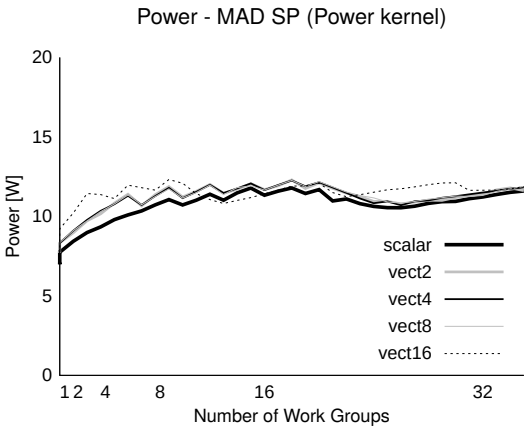


Figure 4.11: Power comparison of the *mad* operation across different SIMD data types for single and double precision.

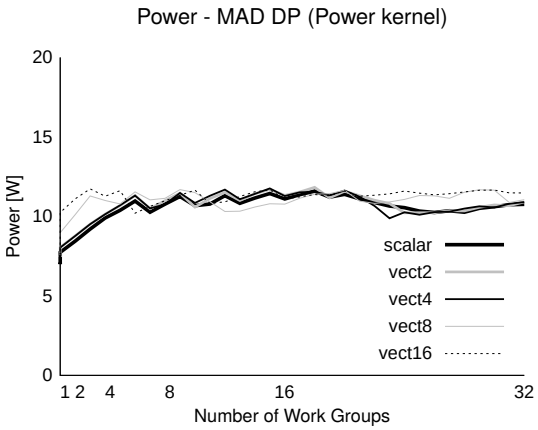


Figure 4.12: Power comparison of the *mad* operation across different SIMD data types for single and double precision.

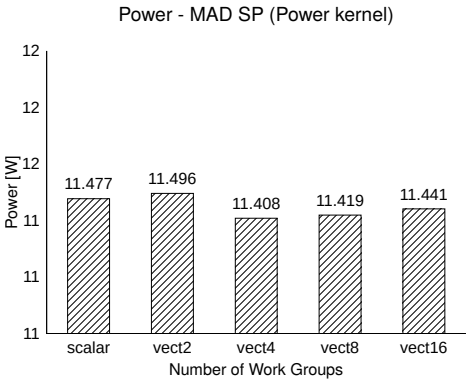


Figure 4.13: Maximum power values achieved by the different SIMD lengths for single precision.

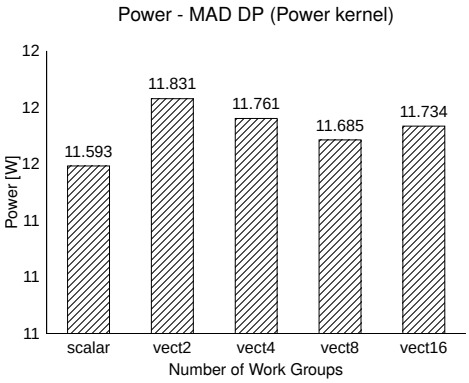


Figure 4.14: Maximum power values achieved by the different SIMD lengths for double precision.

length, there comes associated a rise of the data set size, which corresponds to a rise in the execution time for the same number of total threads. By lowering the outer loop iterations in accordance to the SIMD length, the execution time remains roughly independent of the SIMD length.

As mentioned in Section 4.2, the power benchmarks have a performance curve similar to the FPU benchmarks, due to the similarities between their algorithms. This same trend is noticed in the power curves, with maximum power being displayed for work groups of a size multiple of the number of subslices. Figures 4.11 and 4.12 illustrate this phenomenon. According to Figures 4.13 and 4.14, *vect2* is the most power hungry SIMD length, with *vect4* drawing the least amount of power. For single precision, the difference in power across the different SIMD lengths is less pronounced than for double precision, even though they share the same tendencies.

4.5 Intel GPU Energy Efficiency Analysis

Proper use of the GPU as an accelerator rely not only on the performance gains of offloading computation to the GPU, but on energy efficiency as well. This becomes increasingly important on battery

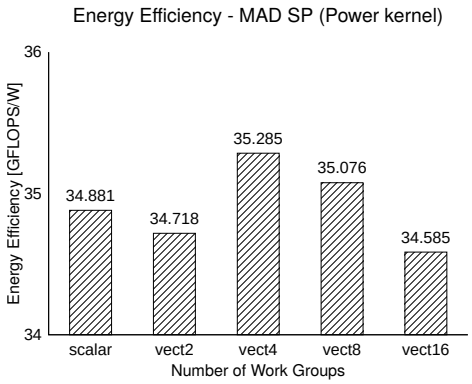


Figure 4.15: Energy efficiency of the different SIMD data types under the operation *mad* for single precision.

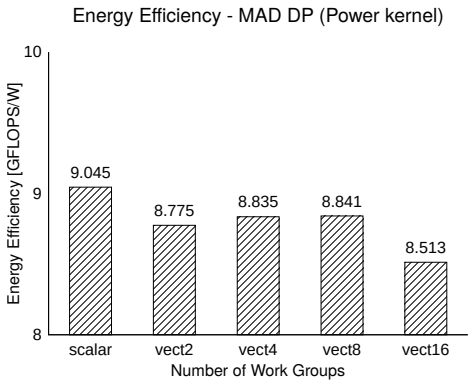


Figure 4.16: Energy efficiency of the different SIMD data types under the operation *mad* for double precision.

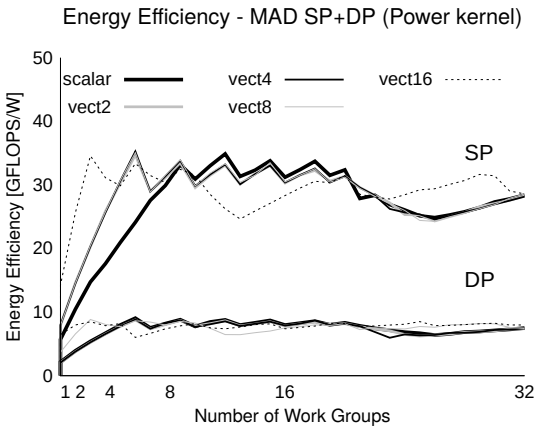


Figure 4.17: Energy efficiency of the different SIMD data types under the operation *mad*.

powered platforms such as smartphones. Thus, if a workload does not benefit of a reduction in computation time when offloaded to the GPU, but of a power savings instead, then the offloading is viable for the purpose of preserving battery power. In this regard, a study on the different SIMD data types' energy efficiency was conducted.

Folowing after the performance and power analyses, the energy efficiency curves retain the serrated curve. For low work group usage, *vect16* demonstrates the highest energy efficiency only due to the higher number of computations performed. When the GPU is fully exercised, Figure 4.17 shows the other SIMD lengths reaching higher efficiency than *vect16*. As with power and performance, the highest efficiency occurs at work group numbers multiple of the number of subslices.

Figures 4.15 and 4.16 shows the maximum energy efficiency attained by each SIMD length for single and double precision, respectively. *vect4* achieves the highest energy efficiency, which is expected due to the FPUs physically being 4-wide. *vect16*, on the other hand, possesses the least energy efficiency of the SIMD data types, followed by *vect2*.

When compared to CPU, the GPU achieves far higher energy efficiency values.

4.6 Summary

In this chapter, the GEN assembly and the register region syntax were addressed and explained. In addition, the algorithms designed for the kernels were presented, as well as the results acquired for performance, memory bandwidth, power and energy efficiency.

It was revealed that there is a dependency between the number of issued work groups and the number of subslices of the platform. Peak performance is achieved if the number of work groups is a multiple of the number of subslices, given a work group size is also a multiple of the number of EUs in a subslice. This was shown to be replicated across performance, bandwidth, power and energy efficiency, while occurring for all SIMD data types. The results show the best energy efficiency is achieved using *vect4* SIMD type, while *vect16* achieves the worst energy efficiency. An impactful difference is shown in regards to energy efficiency between single precision and double precision. The negligible difference in power consumption between the two, coupled with single precision having a fourth of the double precision's performance explains this result. Another insight from the benchmark results show that SIMD length has a non insignificant impact on memory bandwidth.

Chapter 5

Conclusions

In order to evaluate the benefits of integrated GPUs alongside the better performant discrete GPUs, a micro-architectural characterization was performed on the Intel Gen GPU architecture in this thesis. Furthermore, the development of a software tool capable of exposing the GPU's performance counters was conducted. This work laid the foundation for an in-depth benchmarking aiming to exploit the architecture's upper-bounds. In turn, a study that hinged on performance and energy efficiency gleamed several insights in how to employ the GPU in energy efficient domains. From the other side of the spectrum, ways to efficiently program performance critical workloads on integrated GPUs arose.

The main contributions of this thesis are the tool that can be used to aid further research and the insights brought by the performance and energy efficiency analysis. The 35 GFLOPS/W of the Intel GPU surpasses that of the CPU and rivals current discrete GPUs. Through the experimental results, a peak of 399,02 GFLOP/s for single precision and 99,99 GFLOP/s for double precision was achieved for MAD operations, and 199,96 GFLOP/s for single precision and 49,98 GFLOP/s for double precision for ADD operations, in regards to the floating point benchmark. For the memory benchmarks, *vect4* achieved a bandwidth of 191,99 GB/s, with all other vectorization types falling behind. The power benchmarks average 11,45 W for single precision and 11,72 W for double precision. These results open the door to further research in better workload distribution in heterogeneous systems in order to maximize energy efficiency. Of note, is the condition necessary to achieve peak performance. For this end, be it floating point performance, memory bandwidth or energy efficiency, it is required that the number of work groups be a multiple of the number of subslices. This avoids work imbalance in the GPU, reducing subslice idle time.

Future work falls on performing energy efficiency analyses for other integrated GPUs and of energy aware scheduling in heterogeneous systems. By knowing beforehand the most energy efficient SIMD data type, compilers can gain a new dimension besides performance and executable space in which to explore. New models can be proposed that offer a bigger focus in energy aware computing.

Bibliography

[1] S. Junkins. *The Compute Architecture of Intel Processor Graphics Gen9*. Intel, August 2015.

[2] Gen9.5 - Microarchitectures - Intel - WikiChip, 2018. URL `https://en.wikichip.org/wiki/intel/microarchitectures/gen9.5`.

[3] *Intel Open Source HD Graphics and Intel Iris Plus Graphics Programmer's Reference Manual*. Intel, January 2017.

[4] P. Gera, H. Kim, H. Kim, S. Hong, V. George, and C.-k. C. K. Luk. Performance Characterisation and Simulation of Intel's Integrated GPU Architecture. pages 139–148, 2018.

[5] G. Lupescu, E. I. Slusanschi, and N. Tapus. HALWPE: Hardware-Assisted Light Weight Performance Estimation for GPUs. *Proceedings of the International Conference on Parallel Processing Workshops*, pages 39–44, 2017.

[6] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. GPGPU performance and power estimation using machine learning. *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 564–576, 2015.

[7] G. Lupescu, E. I. Slusanschi, and N. Tapus. Using the Integrated GPU to Improve CPU Sort Performance. *Proceedings of the International Conference on Parallel Processing Workshops*, pages 39–44, 2017.

[8] V. García, J. Gómez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Peña. Evaluating the Effect of Last-Level Cache Sharing on Integrated GPU-CPU Systems with Heterogeneous Applications. *Proceedings of the 2016 IEEE International Symposium on Workload Characterization, IISWC 2016*, pages 168–177, 2016.

[9] J. Hestness, S. W. Keckler, and D. A. Wood. A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior. *IISWC 2014 - IEEE International Symposium on Workload Characterization*, pages 150–160, 2014.

[10] G. Lupescu, E. I. Slusanschi, and N. Tapus. Analysis of thread workgroup broadcast for Intel GPUs. *2016 International Conference on High Performance Computing & Simulation (HPCS)*, pages 1019–1024, 2016.

[11] S. Hong and H. Kim. An integrated GPU power and performance model. *ACM SIGARCH Computer Architecture News*, 38(3):280, 2010.

[12] M. Daga, A. M. Aji, and W. C. Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. *Proceedings - 2011 Symposium on Application Accelerators in High-Performance Computing, SAAHPC 2011*, pages 141–149, 2011.

[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. *Proceedings of the 2012 International Conference on Electromagnetics in Advanced Applications, ICEAA'12*, 27(0):1046–1049, 2012.

[14] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali. Adaptive heterogeneous scheduling for integrated GPUs. *Proceedings of the 23rd international conference on Parallel architectures and compilation - PACT '14*, pages 151–162, 2014.

[15] G. Lupescu, E. I. Slusanschi, and N. Tapus. Analysis of OpenCL work-group reduce for Intel GPUs. 2016.

[16] M. Dashti and A. Fedorova. Analyzing Memory Management Methods on Integrated CPU-GPU Systems. *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management - ISMM 2017*, pages 59–69, 2017.

[17] A. Munshi and L. Howes. *The OpenCL specification*. Khronos, July 2015.

[18] Intel GPU Tools. URL `https://cgit.freedesktop.org/xorg/app/intel-gpu-tools/`.

[19] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel, October 2017.

[20] R. I. Introduction to GEN Assembly, 2016. URL `https://software.intel.com/en-us/articles/introduction-to-gen-assembly`.

[21] S. Pandruvada. Running Average Power Limit – RAPL — 01.org, 2014. URL `https://01.org/blogs/2014/running-average-power-limit-\OT1\textendash-rapl`.

Appendix A

A Counters

Table A.1: Available A Counters [3].

Counter	Event	Description
A0	Undisclosed	Undisclosed
A1	# of Vertex Shader Threads Dispatched	Count of VS threads dispatched to EUs
A2	# of Hull Shader Threads Dispatched	Count of HS threads dispatched to EUs
A3	# of Domain Shader Threads Dispatched	Count of DS threads dispatched to EUs
A4	# of GPGPU Threads Dispatched	Count of GPGPU threads dispatched to EUs
A5	# of Geometry Shader Threads Dispatched	Count of GS threads dispatched to EUs
A6	# of Pixel Shader Threads Dispatched	Count of PS threads dispatched to EUs
A7-A20	Aggregating EU counter 0-13	Programmable
A21	2x2s Rasterized	Count of the number of samples of 2x2 pixel blocks generated from the input geometry before any pixel-level tests have been applied
A22	2x2s Failing Fast pre-PS Tests	Count of the number of samples failing fast before pixel shader execution tests (counted at 2x2 granularity)
A23	Undisclosed	Undisclosed
A24	2x2s Killed in PS	Number of samples entirely killed in the pixel shader as a result of explicit instructions in the kernel (counted at 2x2 granularity)
A25	2x2s Failing post-PS Tests	Number of samples that entirely fail tests that can only be performed after pixel shader execution (counted at 2x2 granularity)
A26	2x2s Written To Render Target	Number of samples that are written to render target (counted at 2x2 granularity)
A27	Blended 2x2s Written to Render Target	Number of samples of blendable that are written to render target (counted at 2x2 granularity)
A28	2x2s Requested from Sampler	Aggregated total 2x2 texel blocks requested from all EUs to all instances of sampler logic
A29	Sampler L1 Misses	Aggregated misses from all sampler L1 caches
A30	SLM Reads	Total read requests from an EU to SLM (including reads generated by atomic operations)
A31	SLM Writes	Total write requests from an EU to SLM (including writes generated by atomic operations)
A32	Other Shader Memory Accesses	Aggregated total requests from all EUs to memory surfaces other than render target or texture surfaces (e.g. shader constants)
A33	Other Shader Memory Accesses That Miss First-Level Cache	Aggregated total requests from all EUs to memory surfaces other than render target or texture surfaces (e.g. shader constants) that miss first-level cache
A34	Atomic Accesses	Aggregated total atomic accesses from all EUs. This counter increments on atomic accesses to both SLM and URB
A35	Barrier Messages	Aggregated total kernel barrier messages from all Eus (one per thread in barrier)