Roofline Analysis and Performance Characterization for Intel Integrated GPUs

Afonso Rodrigues de Carvalho
afonso.carvalho@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

Abstract

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

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

I. Introduction

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

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

A thorough investigation of these architectures is critical so that the applications achieve the maximum potential a GPU has to offer in GPGPU. The identification of these characteristics is feasible via benchmarking tests fully exploiting the capability of highly parallel computation units and the memory access bandwidth and latency. As means to keep developing computer architectures, performance models and profiling methods that delineate the performance of applications are fundamental. In the scope of this work, insightful models such as the ORM [7] and CARM [8][9] are used to characterize the behavior and performance of applications in GPUs. These are valuable tools to guide programmers through optimizations to their code and help to portray the nature of the applications' boundaries. Tools such as Intel Advisor [10] have integrated CARM onto its software. Additionally, Intel Advisor also can enable a deeper glance into developed microbenchmarks and the underlying architecture for their execution, providing crucial data for optimization practices and analysis methods.

A. *Motivation*

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

II. Background and Related Work

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

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

In Gen 9.5 products, the L3 data cache shares the coherence domain with the CPU, meaning that if the CPU changes the data in the memory subsystem, the L3 cache refreshes the value of that data, to keep it coherent with CPU. The cache has a capacity of 512 KB or 768 KB per slice (depending on the product), with 64 B cache lines. It has a read/write bandwidth to the GTI of 64 B/cycle [5].

Each subslice contains multiple EUs, a thread dispatcher, a sampler and a data port. Gen9 and Gen9.5 products have from six to eight EUs per subslice. The data port is the memory unit that is in charge of load and store operations. It has a 64 B/cycle write and read bandwidth to the L3 cache. To maximize memory bandwidth, it tries to coalesce scattered memory requests into one 64 B cache line. All memory load/store, SIMD scatter/gather and SLM accesses travel through the data port [5].

The EU is the basic computation unit in this iGPU architecture. It computes all operations for 3D, media, or GPGPU kernels. Figure 1 contains the diagram of a Gen9.5 EU. All EUs have components for hardware thread management and four computation units. Thread management starts on the fetch unit, which selects which instructions are allocated slots on specific EUs, up to a maximum of seven hardware threads at any given instant. From there, the thread arbiter picks a thread from the pool of threads that are ready to be executed, up to four at the same time, given there are only four computation units. The choice depends on the type of instructions and computation that the threads require from the EU.

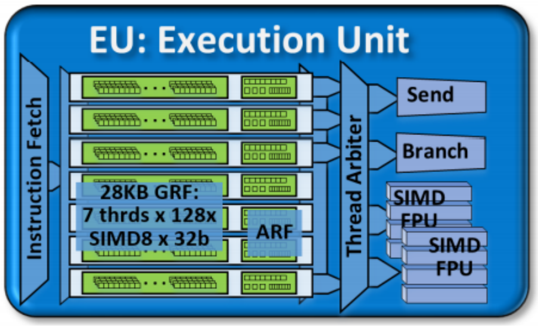


Fig. 1: Intel Gen9.5 iGPU Execution Unit Architecture [5]

The four computation units are responsible for executing all instructions present in the selected threads. The Send unit handles all load/store requests to memory or sampler operations, making use of the data port of the subslice and depending on the location of the L3 cache or the DRAM [5]. The branch unit handles SIMD the divergence and the convergence between kernels. [5].

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

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

OpenCL was adopted and supported by many vendors as their main GPU programming framework, being available in a wide range of target devices, such as GPUs, accelerators, and FPGAs. OpenCL introduces the concept of host and device, the latter being the platform where the kernel function is executed. The host is the platform that selects the target device, copies the data to it, selects one or more kernels, and queues them up for execution on the device. The kernel functions are coded by the programmer and can have any purpose. OpenCL defines the spawned instances as work items and work-groups as a group of work items. By tuning each parameter, the workload distribution can be adjusted to achieve higher performance.

The computation abilities of Intel Processor Graphics Gen9.5 iGPUs can be exploited using the OpenCL programming model. In this architecture, a single work-group shares the same local memory space. For kernels that use SLM, runtimes map all instances of a work-group to EU threads in a single subslice. Thus, all kernel instances within a work-group share the same 64 KB [5]. Each work-group is mapped into a subslice, therefore several instances are required to reach a GPU's maximum potential performance [11]. The EU threads in one subslice run all work items from a specific work-group. If there is not enough capacity for the incoming work items, the execution is stalled, forcing the EU threads to wait, meaning parallelization was not possible. This event can occur when a subslice has full occupancy or when there was inefficient resource usage [11].

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

The ORM and the CARM define the metrics OI and AI, respectively, (in FLOPS/β_D) as the amount of executed operations (ϕ in FLOPS) per accessed byte of the memory traffic (B_D in β_D/s). These are similar but fundamentally different metrics since the memory traffic notion is distinct between them. The metric above defined is the horizontal axis of the roofline models. The vertical axis corresponds to the throughput of the application, labeled as Performance, in FLOPS. Considering F_p, in FLOPS, as the peak floating point throughput of the device, the maximum attainable performance ($F_a(OI)$ in FLOPS) is obtained according to equations 1 and 2, for the ORM and the CARM, respectively

$$F_a(I) = \min\{B_D * I, F_p\} \tag{1}$$

and

$$F_a(I) = \min\{B(\beta) * I, F_p\}. \tag{2}$$

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

In the CARM, a significant unexplored area of the roofline plot is shown, corresponding to the caches' bandwidth, the highest being the L1 cache. If the tested application sits in the middle of the plot, the results can be inconclusive regarding the essence of the boundary. The CARM can also profit from the inclusion of new ceilings for optimization purposes. MAD instructions, ADD/MUL instructions, or SIMD utilization are examples of new roofs to include in both models. The models retain a few key differences.

A. Related Work

In [12] and [13], the graphics portion of the Intel iGPU is put to the test through the benchmarking of games and other 3D applications. Both papers resort to an offline feature selection on which a regression technique was applied, and an online learning method. They use a lightweight recursive least-squares method to accurately predict the change in frame time and power, respectively.

The authors of [14] compared the execution of the FDTD algorithm in an iGPU to a discrete GPU. The FDTD is a method to model electromagnetic fields, requiring several iterations of FLOPS. The established procedure was to divide the chunks between the host (CPU) and device (iGPU) and exchange data after one chunk calculation and repeat these steps. The programming API used was DirectCompute. As for the final results, the iGPU managed to obtain almost twice the throughput of a low range discrete GPU. It should be noted that a small dataset was used, which allowed the iGPU to prevail over the discrete one.

In [15], a compiler/framework for heterogeneous systems was designed, to distribute the workload between the CPU and other devices. This work was developed using the OpenMP API. The compiler translates specific regions of the code and translates them to GPU kernels and generates a CPU parallelized version in case the communication with the GPU fails. The compiler collects data from the program the builds the performance models; however, the final evaluation is only known at runtime. A prediction is generated with the obtained data of the advantages of offloading the work to the GPU. According to the result, either of the generated code versions gets executed.

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

In [17], a significant performance characterization is made to an Intel iGPU. These tests were made in the Intel Skylake

[6] and Intel Kabylake [6] iGPU architectures. In this article, through the usage of OpenCL kernels, set up with work-groups composed of 32 work-items each, the number of work-groups was varied, for single-precision and double-precision for MAD operations. The authors concluded that the peak throughput is achieved with 32 work-items per work-group and for 96 and 192 work-groups for SKL and KBL architecture, respectively. For reduced work-group amounts, the throughput is low due to the lack of operations to saturate the EUs.

The next microbenchmark of [17] consists of a single-threaded random access test for varying set sizes, to obtain the access times to different memory blocks. For the smaller set sizes, the access time proved to be constant for the SKL iGPU, spiking at a size of 512 KB, which is the standard size of the L3 cache. Whenever the set size becomes larger than the L3 cache, the access time does not remain constant for the LLC accesses. Therefore, the authors believe that the iGPU does not take advantage of the full capacity of the LLC or that some space may be reserved for the CPU.

The work developed in [18] shares similarities with [17] by analyzing the effect of the LLC in computation. The gem5 simulator was used for this work, representing an integrated CPU-GPU system. The LLC proved to cause at least a slight speedup in the tested microbenchmarks, particularly in operations requiring fine-grained synchronization and atomic function usage. Data sharing between the devices also presented improvements, causing reduced memory access latency.

The papers [19] and [20] intend to explore the Intel iGPU microarchitecture and to test OpenCL kernel algorithms for work-group broadcast, and work-group reduce operations, analyzing the hardware behavior. The used Intel Runtime for GPUs was Beignet, which converts the OpenCL kernels into the iGPU GEN Assembly instruction set [21] through an LLVM compiler. The low-level assembly was thoroughly analyzed in both papers. The best throughput and lowest latency are achieved for 64 and 128 work-items per work-group. Note that for MAD operations, tested in [17], 32 work-items were enough to achieve maximum throughput, however, for the tested operations in [19] it falls short of the expectations. Similarly, in [20], 64 work-items per work-group achieve the best throughput of all the options.

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

III. Proposed Work

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

A. *OpenCL Benchmark Architecture*

The microbenchmark set was developed in C/C++, employing OpenCL to operate in the iGPU. OpenCL has a set of initialization functions used independently of the type of kernels chosen. Consequently, these functions are common to almost all OpenCL benchmarks. Their general purpose is to select the device to work on, whether CPU, GPU or other, to create a queue for all the commands to execute on that device, and allocate the needed device memory for the data objects. Afterward, the data has to be copied to the device so the latter can eventually execute the kernels. If needed, the output data is transferred to be read back on the host. Due to the common architecture in many benchmarks, the variations between the different proposed tests only require distinct kernels and redefined object data types and sizes, as the main data transfer and kernel execution processes do not change. Figure 2 presents a detailed flow chart of the general microbenchmark setup and behavior, describing all the established methods from the start to the end of the application.

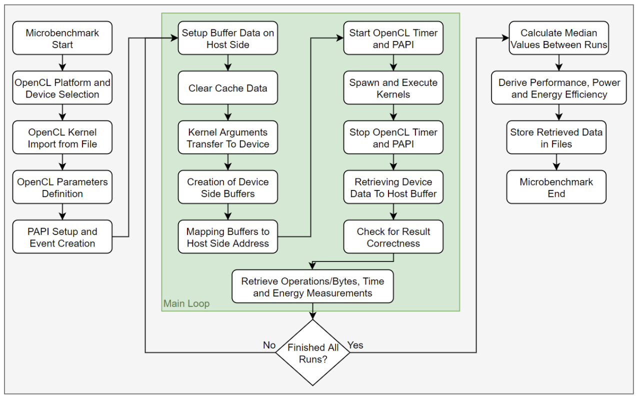

Microbenchmark Start
OpenCL Platform and Device Selection
OpenCL Kernel Import from File
OpenCL Parameters Definition
PAPI Setup and Event Creation
Setup Buffer Data on Host Side
Clear Cache Data
Kernel Arguments Transfer To Device
Creation of Device Side Buffers
Mapping Buffers to Host Side Address
Start OpenCL Timer and PAPI
Spawn and Execute Kernels
Stop OpenCL Timer and PAPI
Retrieving Device Data To Host Buffer
Check for Result Correctness
Retrieve Operations/Bytes, Time and Energy Measurements
Main Loop
No
Finished All Runs?
Yes
Calculate Median Values Between Runs
Derive Performance, Power and Energy Efficiency
Store Retrieved Data in Files
Microbenchmark End

Fig. 2: OpenCL Microbenchmark Architecture and Behavior Flowchart

OpenCL provides a list of all compatible hardware devices, from several vendors, that are accessible in the machine running the set of microbenchmarks. This includes multiple CPUs, GPUs and FPGAs. To achieve this, the functions clGetPlatformIDs() and clGetDeviceIDs() choose a device to be the target setup for the application. In this application, an Intel Gen 9.5 iGPU is selected through these commands. Following that, several initialization functions are required to set up the OpenCL kernels. These include the creation of a command queue and a context for the kernels on the selected device; The kernels selected for microbenchmarking are stored in .cl files and parsed to string types through the host code. Multiple kernels are present in one .cl file, corresponding to different vectorization levels for the same GPU operation. The intended vectorization data type is given as an argument of the application, so only the respective kernel gets copied. The given kernel input string is a required parameter for the clCreateProgramWithSource(), and consequently, the clCreateKernel() functions, allowing the compilation of the kernels to later start their execution. By this step, kernel compiler flags can be defined. The maximum achievable throughput in this architecture requires MAD operations and high amounts of parallelization. To do so, the flag '-cl-mad-enable' must be set. Other available flags involve compiler optimizations and math simplifications, both of which were not used in the Thesis, as any hidden optimization can give unpredictable results and compromise the validity of the run tests.

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

To microbenchmark and measure statistics from the hardware's behavior, a precise profiling tool is required. The chosen option is also integrated into the OpenCL programming model in the form of events, which ensure reliability and reduced overheads. The function parameters ensure the measurement only from the start of an event to its end. The selected event was the clEnqueueNDRangeKernel() function, which provides the sole calculation of the kernel time execution, as only the work done on the GPU, provided by the developed kernels, is relevant. This profiling method is used consistently and with the same precision among all developed kernels and benchmark tests in this Thesis to ensure coherency between results.

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

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

B. OpenCL Kernels for iGPU Microbenchmarking

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

TABLE I: Developed OpenCL Kernels for iGPU Architecture Microbenchmarking

Operations	Data Type	Vectorization
Addition	Int, Float, Double	1, 2, 4, 8, 16
MAD	Float, Double	1, 2, 4, 8, 16
Load/Store	Float	1, 2, 4, 8, 16
Load	Float	1

For throughput kernels, three distinct data types are tested, floating-point single-precision, floating-point double-precision, and integer. Buffers consist of different vector types for the developed kernels that support them. They range from scalar to vector2, vector4, vector8, and vector16. The change in the size of each element due to vector types was also taken into account and adjusted so that every iteration had the same amount of executed operations, to observe any compilation changes given different buffer vectorizations and any performance differences among them. These data types are tested for both addition and MAD operations, with the latter having '-cl-mad-enable' included in the kernel compiler flags. Only MAD is expected to achieve maximum throughput, as the operation implies both multiplication and addition are done in one single clock cycle, effectively doubling the throughput of an ADD instruction. Figures 3 and 4 depict examples of the developed kernels to attempt to reach the maximum throughput possible in the architecture, in ADD and MAD operations, respectively.

```
i = get_global_id(0)
if i < DATASIZE
    for j < ITERATIONS
        a[i] = a[i] + b[i]
```

Fig. 3: ADD Operation OpenCL Kernel

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

```
i = get_global_id(0)
if i < DATASIZE
    for j < ITERATIONS
        a[i] = a[i] * b[i] + b[i]
```

Fig. 4: MAD Operation OpenCL Kernel

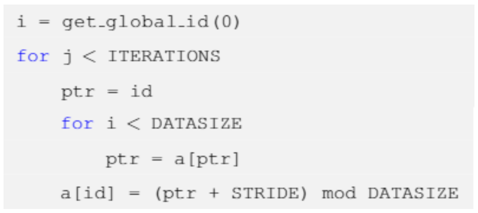
Through testing, the global prefix on both buffers would yield much lower throughput, implying that the 'b' buffer was not being cached, requesting the element loads from main memory. As a result, the buffer 'b' is defined in the constant memory level since it does not need to be written to in any of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

To achieve the maximum possible bandwidth, the L3 cache, present inside a slice of the iGPU has to be exploited to the maximum. This requires significant reuse of the values stored in the cache, to avoid, as much as possible, cache misses and travel times to the DRAM, which have much lower bandwidth. This is achieved by forcing repeated loads from the data stored in the cache. Bandwidth tests were executed for floating-point single-precision and double-precision data types. The exclusion of integers is due to similar size and instruction execution compared to the float data type, originating naturally equal results. The first presented kernel includes both load and store instructions, for all vectorization ranges, while the second one only runs a scalar pointer chasing algorithm, attempting to minimize the number of store instructions and deriving the memory bandwidth measurement from load instructions. A single-threaded version of the second kernel is also used to measure memory latency to all caches and main memory. Figures 5 and 6 illustrate the differences between the memory kernels.

```
i = get_global_id(0)
for j < ITERATIONS
    a[i] = b[i]
```

Fig. 5: Load/Store Operation OpenCL Kernel

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



```
i = get_global_id(0)
for j < ITERATIONS
    ptr = id
    for i < DATASIZE
        ptr = a[ptr]
    a[id] = (ptr + STRIDE) mod DATASIZE
```

Fig. 6: Pointer-Chasing OpenCL Kernel

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

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

IV. Experimental Evaluation

The Intel Gen 9.5 iGPU architecture was the target of an extensive performance characterization, in terms of computation throughput, memory bandwidth and latency, power consumption, and energy efficiency. Further analysis was conducted through the usage of CARM, through the Intel Advisor software. All tests were executed on the same setup, present in Table II.

TABLE II: Experimental Setup

System	Description
CPU	Core i5-8300H
iGPU	UHD Graphics 630

A. *Intel iGPU Throughput Characterization*

To properly analyze the throughput capabilities, the kernels for addition and for MAD were developed. These tackle the regular one-cycle arithmetic operations and the special case of multiplication and addition in one operation, respectively. However, before diving into the achieved results, it is crucial to identify the theoretical peak throughput of the target iGPU. For 32-bit single-precision floating-point, since the FPUs in every EU are physically composed of 4 SIMD lanes, each FPU can produce 4 operations per clock cycle. Coupled with the second FPU, each EU can provide up to 8 instructions per cycle. In the target iGPU, specifically, the 23 EUs, coupled with a maximum frequency of 1000 MHz (or 1 GHz), support up to 4*2*23*1 GHz = 184 GFLOPS, for arithmetic operations such as addition or subtraction. Since a MAD instruction achieves both addition and multiplication in one clock cycle, technically the achieved throughput is exactly the of double the previous result, standing at 368 GFLOPS for MAD. However, if the provided data is now composed of double-precision floating-point (or double) elements, the required cycles to produce the same number of operations relative to the float data type are cut to half, due to its doubled size of 64 bits per element. Additionally, the compute architecture of Gen 9.5 iGPUs only designed one FPU to be capable of supporting double-precision operations and other advanced math functions. As such, the maximum throughput for the double data type given by flooding the iGPU with ADD kernels only reach up to 184/4 = 46 GFLOPS. For MAD operations, the peak is obtained under similar circumstances, 368/4 = 92 GFLOPS. Table III illustrates the maximum theoretical throughput, depending on the operation and chosen data type.

TABLE III: Theoretical Maximum Throughput of the Experimental Setup in GFLOPS

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

The GPU performance was evaluated for three different data types: 32-bit integer, 32-bit float, and 64-bit double, for scalar and vectorized types. Figures 7 and 8 present the obtained throughput with increasing work-groups, for the float addition operation and the float MAD operation, respectively.

The maximum throughput achieved in the Figure 7 setup was 172 GFLOPS out of 184 GFLOPS, approximately 93.5% of the iGPU's theoretical peak. The best results come from vectorized data types Float4 and Float8 and the worst from Scalar and Float16. The plot continuously displays a gradual increase and sudden decrease in performance throughout the entire test. The oscillating nature of the plot is tied to the workload imbalance among the EUs in different subslices. This occurrence is especially aggravated by the imbalance in the number of EUs in the target setup, which changes

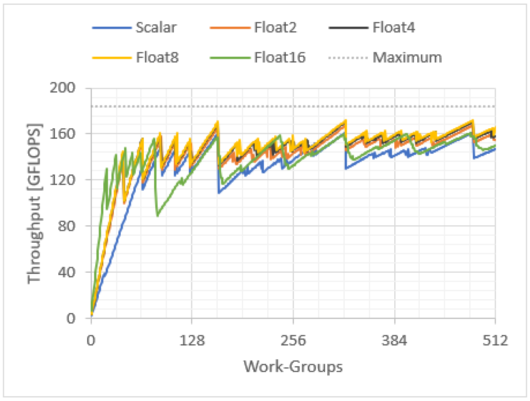


Fig. 7: Float ADD Operation Throughput with increasing total Work-Groups

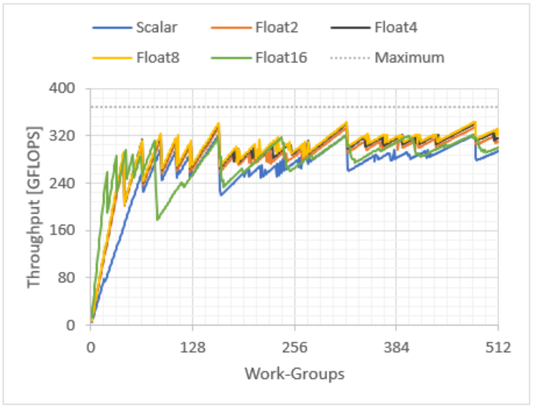


Fig. 8: Float MAD Operation Throughput with increasing total Work-Groups

how the best performance is achieved. The number of work-groups for which the ADD kernel was executed on the iGPU reached the peak value of 172 GFLOPS are 161, 322, and 483 work-groups. All of these are multiples of 161, which is, not coincidentally, the maximum amount of hardware threads that this iGPU can have at the same time stored in all combined EUs. In the Gen 9.5 iGPU microarchitecture, each EU can have up to 7 hardware threads with a kernel ready to execute, from which the thread arbiter selects the one to insert in the recently freed up FPU. This means that full occupation of the threads is achievable by deploying 7*23 = 161 work-groups in them, and the same logic applies to the multiples of 161 work-groups. The CARM analysis result for the microbenchmark containing the addition kernels illustrates achieved values from 169 to 172 GFLOPS which fall in line with the peak reached by the software. In Figure 8, all performance curves are equivalent to the ones in the addition kernel, but with double the obtained throughput, as expected. Float4 and Float8 are the best performers, and Scalar and Float16 are the worst ones.

B. Intel iGPU Memory Bandwidth and Latency Characterization

The memory bandwidth of the iGPU is a vital metric, as it can be in many cases, the major bottleneck of an application to be run on a GPU. This is especially the case in dedicated GPUs, but in the end, it is very dependent on the software and on the operations that require execution. The studied iGPU has a dedicated L3 cache, which contains a separated portion for the SLM, a shared LLC with the CPU, and the system main memory DRAM, also shared with the main processor. The latter two require data to traverse the Ring Interconnect of the architecture, to go from iGPU to LLC/DRAM, or vice-versa. Therefore, if these requests are a significant portion of the workload, the achieved bandwidth by the iGPU is distant from its potential maximum bandwidth. Supposing all required data is in the L3 Cache, located inside the iGPU, having each subslice with 64 B read/write bandwidth per cycle on the data port. Each EU has one Send unit, and each GRF register in an EU can contain up to 32 B of data. In theory, if multiple EUs have send instruction requests ready to be executed, to fetch values from the cache, during the whole execution time of the kernel, the maximum bandwidth would be achieved. The Load/Store kernel intends to achieve that, by having two large floating-point buffers, one to load an element from and one to send the same element to. Repeating this instruction for several iterations forces mass usage of the L3 cache, to try to mitigate the latency of any necessary DRAM accesses. Contrary to what was observed in the throughput performance characterization, changes in the vectorization level of the chosen data type have a much more drastic effect. This microbenchmark, similarly to the throughput performance characterization ones, varies the number of OpenCL work-groups, in Figure 9.

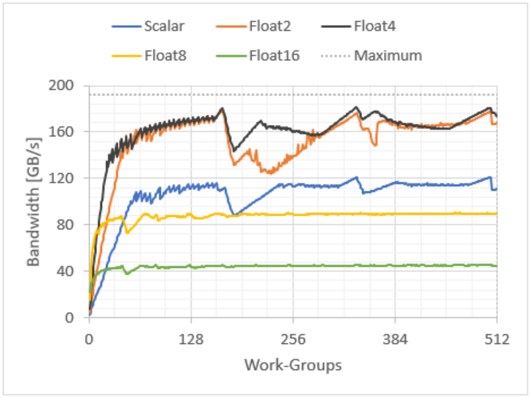


Fig. 9: Float Load/Store Operation Bandwidth with increasing total Work-Groups

The best bandwidth was achieved by the single-precision Float2 and Float4 kernels, at approximately 178 GB/s and 181 GB/s, for a relative bandwidth of 92.7% and 94.3%, respectively. The Scalar kernel sees a significant drop-off, to a maximum of 120 GB/s, meanwhile, Float8 and Float16 reach no more than 90 GB/s and 45 GB/s. From analyzing the

GEN Assembly code behind the generated kernels, Scalar, Float2, and Float4 are identical in instruction count and order, varying only the way the registers are addressed and which ones are addressed in each instruction. Each load in the kernel is compiled into two send instructions, executed in SIMD16 of 16-bit words, with the second one offset by 16 words, totaling up to the maximum of 64 B. However, in Float8 and Float16, each load is compiled into SIMD8 instructions of 32-bit floats, without offset, so each send instruction is a new memory request. Therefore, Float8 only makes use of half of the maximum bandwidth. Float16 requires double the number of elements, requiring four instead of two send instructions, of which each one is a new memory request, consequently making use of only a quarter of the available bandwidth. This falls in line with obtained results as 90 GB/s and 45 GB/s are, respectively, half and one-fourth of the attained maximum of 181 GB/s.

Memory latency was measured through a pointer-chasing kernel. Contrarily to the bandwidth measurement, it was single-threaded, meaning only one work-group with one work-item would execute the kernel. The stride 16 on the array would force fetching new cachelines in every cycle, and the average access time was measured by dividing the OpenCL event profiling timer by the number of accesses made. Figure 10 displays the attained data.

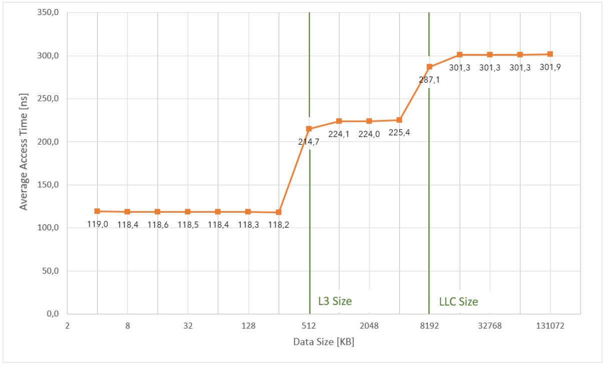


Fig. 10: Float Pointer-Chasing Algorithm for Latency with increasing Data-Size

The obtained plot shows a staircase shape, which is to be expected in a latency microbenchmark such as this one since all three memory blocks' are highlighted by constant access times. Additionally, a steep increase in access times happens when the array data size increases to a point where it no longer fits in the previous memory block, thus having to order requests from the following one, taking a progressively longer time to fetch the data. These results also match the ones obtained from paper [17].

C. Intel iGPU Power and Energy Efficiency Characterization

Both measurements of power consumption and energy efficiency were derived from energy expenditure metrics, collected through PAPI which provides data collections of the RAPL interface. Power requirements were calculated through the division of the energy expenditure the kernel execution time. On the other hand, energy efficiency was calculated by measuring the amount of executed operations or transferred bytes, divided by the amount of energy spent during the kernel execution. The power kernel for the MAD operation appears to be comparable between different vectorization levels, all inserted in the range from 8 W to 10 W readings, apart from oscillations when the workload saturation is low. The requirement in terms of power is slightly higher for less vectorized kernels, with Scalar requiring the most and Float16 requiring the least. This occurrence proved to be constant among all throughput kernels tested for power requirements. The energy efficiency readings reflect parts of both the throughput characterization plot and the power readings plot, as it assimilates itself to the first one. However, it distances the performance curves of the different vectorization levels due to their energy expenditure. Scalar and Float2 were the kernels that required the most power, and as such, decrease in energy efficiency relative to Float4, Float8, and Float16, all of which are more energy efficient.

D. Results Analysis

To conclude the display of the characterization done on the Gen 9.5 iGPU architecture, a more general review is taken. In the throughput computation department, for the addition operation kernel, Float4 and Float8, accompanied by Int4 and Int8, achieved the peak of 172 GFLOPS out of a possible 184 GFLOPS. For MAD operations, the same applies, but instead at absolute values of 345 GFLOPS out of 368 GFLOPS. Using the 64-bit double-precision floating-point data type, the obtained values are located at 43 out of 46 GFLOPS for addition kernels and 80 out of 92 GFLOPS for MAD kernels. Overall, different vectorization types only slightly impact the throughput of the iGPU.

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

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

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

V. Conclusions

Modern iGPUs are significantly under explored relative to their counterparts, dedicated GPUs. The size and location restrictions are serious drawbacks that make it inconceivable for an iGPU to have the same computing power as a discrete one. However, their virtues are originated from their restraints. This work took as motivation the concept of usage of an iGPU for general-purpose computation, GPGPU, which, due to which its affiliation with the computer's main processor, is conceivable. The Thesis aimed to search for attainable ways to reach the documented boundaries of the iGPU's potential, to attempt to prove its viability as a GPGPU, through its high parallelization capabilities and shared memory hierarchy system. Special attention was paid to performance characterizing models, in particular to Roofline Models, the ORM and the CARM, introduced as a tool to further analyze any defining results. The approach taken involved developing a microbenchmark set in OpenCL, intending to provide a collection of kernels for iGPU execution. These were built to search for the upper compute-bound and memory bounds that the architecture of an Intel Gen 9.5 iGPU supports. Coupled with additional external tools, energy expenditure, power requirements, and energy-efficiency readings were also taken and analyzed. Future works for these topics involve deeper characterization of useful metrics, such as the application of power requirements and energy efficiency for a CARM-like model for this iGPU microarchitecture. Furthermore, works relating both CPU and iGPU for heterogeneous systems purposes can be developed, either with a focus on computation performance-wise or exploration of the effects of the shared memory system LLC or DRAM, contributing to an approach to using GPU for general purposes alongside a CPU.

References

[1] A. Peleg and B. Ashbaugh and D. Helmly. *MICRO48-Tutorial on Intel® Processor Graphics: Architecture and Programming*.

[2] J. Peddie. *Is it Time to Rename the GPU?*

[3] Khronos. *The open standard for parallel programming of heterogeneous systems*.

[4] Khronos. *The OpenCL™ C 2.0 Specification*.

[5] S. Junkins. *The Compute Architecture of Intel® Processor Graphics Gen9*. 2015.

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

[7] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52 (Apr. 2009), pp. 65–76.

[8] A. Ilic, F. Pratas, and L. Sousa. "Cache-aware Roofline model: Upgrading the loft". In: *IEEE Computer Architecture Letters* 13.1 (2014), pp. 21–24.

[9] D. Marques et al. "Performance Analysis with Cache-Aware Roofline Model in Intel Advisor". In: *2017 International Conference on High Performance Computing Simulation (HPCS)*. 2017, pp. 898–907.

[10] Intel. *Intel Advisor*.

[11] A. Peleg and B. Ashbaugh and D. Helmly. *Maximize Application Performance On the Go and In the Cloud with OpenCL on Intel Architecture*.

[12] Ujjwal Gupta et al. "Adaptive Performance Prediction for Integrated GPUs". In: *Proceedings of the 35th International Conference on Computer-Aided Design*. ICCAD '16. Austin, Texas: ACM, 2016, 61:1–61:8.

[13] Francesco Paterna et al. "Adaptive Performance Sensitivity Model to Support GPU Power Management". In: *Proceedings of the 1st Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems*. ANDARE '17. Portland, OR, USA: Association for Computing Machinery, 2017.

[14] R. G. Ilgner and David B. Davidson. "A comparison of the FDTD algorithm implemented on an integrated GPU versus a GPU configured as a co-processor". In: *2012 International Conference on Electromagnetics in Advanced Applications* (2012), pp. 1046–1049.

[15] A. Chikin et al. "Toward an Analytical Performance Model to Select between GPU and CPU Execution". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2019, pp. 353–362.

[16] Naila Farooqui et al. "Affinity-Aware Work-Stealing for Integrated CPU-GPU Processors". In: *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '16. Barcelona, Spain: Association for Computing Machinery, 2016.

[17] P. Gera et al. "Performance Characterisation and Simulation of Intel's Integrated GPU Architecture". In: *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2018, pp. 139–148.

[18] V. Garcıa et al. "Evaluating the effect of last-level cache sharing on integrated GPU-CPU systems with heterogeneous applications". In: *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 2016, pp. 1–10.

[19] G. Lupescu, E. Slusanschi, and N. Tapus. "Analysis of thread workgroup broadcast for Intel GPUs". In: *2016 International Conference on High Performance Computing Simulation (HPCS)*. 2016, pp. 1019–1024.

[20] G. Lupescu, E. Slusanschi, and N. Tapus. "Analysis of OpenCL Work-Group Reduce for Intel GPUs". In: *2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 2016, pp. 417–423.

[21] R. Ioffe. *Introduction to GEN Assembly*.

[22] G. Lupescu, E. Sluşanschi, and N. Ţăpuş. "Using the Integrated GPU to Improve CPU Sort Performance". In: *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*. 2017, pp. 39–44.