UPPSALA
UNIVERSITET

UPTEC IT 13 017

Examensarbete 30 hp
November 2013

Integrated GPUs

how useful are they in HPC?

Jonatan Jansson

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Integrated GPUs - how useful are they in HPC?

Jonatan Jansson

Due to their potential computation power, GPUs are often used for high performance computing. However, discrete GPUs are connected to the CPU via the PCIe bus, which can cause bottlenecks due to high latency and low bandwidth to the CPU. Lately, integrated GPUs have become more common, and due to being integrated on the CPU-chip, the bottleneck of the PCIe bus is reduced. Integrated GPUs are however less powerful than discrete GPUs, and as they share resources such as power and memory bandwidth with the CPU, heavy CPU utilization can cause a loss in performance.

The aim of this thesis is to investigate the potential of integrated GPUs in high performance computing. This is done by comparing an integrated and a discrete GPU in various settings to see the impacts of the differences in latency and bandwidth, as well as the effects of the integrated GPU sharing resources such as power and memory bandwidth with the CPU.

The results show that the less powerful integrated GPU outperforms the discrete GPU for smaller problem sizes due to the lower latency, even in situations when the shared resources are utilized by the CPU. Integrated GPUs can therefore be very useful when performing high performance computations on smaller datasets. The discrete GPU is however in many cases faster for larger datasets, especially for more arithmetically intense algorithms, due to a larger number of computation cores and dedicated memory.

Handledare: Ian Wainwright
Ämnesgranskare: David Black-Schaffer
Examinator: Lars-Åke Nordén
ISSN: 1401-5749, UPTEC IT 13 017
Tryckt av: Reprocentralen ITC

Popular scientific summary in Swedish

Grafikkortet är en datorkomponent som är specialiserad på att hantera grafik, en process som innehåller en stor mängd beräkningar. I och med att grafikkorten har utvecklats så har även beräkningskraften och flexibiliteten ökat, vilket har gjort det möjligt att använda grafikkort till andra typer av beräkningar än enbart grafik. De senaste åren har det även blivit vanligare att inkludera ett mindre grafikkort integrerat på processor-chippet. Då grafikkortet måste dela både utrymme och resurser med processorn är det mycket mindre än ett separat grafikkort, och i regel inte alls lika kraftfullt. Däremot är ett integrerat grafikkort placerat mycket närmare processorn, vilket leder till att kommunikationen, både med processorn och med system-minnet, går snabbare.

Målet med detta arbete var att undersöka om integrerade grafikkort är användbara för högprestandaberäkningar. Högprestandaberäkningar handlar om att, på ett snabbt och effektivt sätt, utföra tunga beräkningar, och används idag inom flera olika områden, exempelvis signalbehandling, bildbehandling och simuleringar. En anledning till att integrerade grafikkort skulle kunna vara att föredra är den snabbare kommunikationen till processorn och system-minnet, men det finns även andra faktorer som spelar in som exempelvis pris och strömförbrukning. Nackdelen är att integrerade grafikkort inte är lika kraftfulla som separata grafikkort.

För att undersöka hur användbara integrerade grafikkort är sattes ett enkelt testprogram upp och kördes, både på ett integrerat och ett separat grafikkort. För att simulera olika typer av algoritmer kördes programmet i flera olika instanser där både mängden beräkningar och mängden data att göra beräkningarna på varierades. Genom att variera dessa två parametrar går det att få en bra uppfattning om hur skillnaderna i beräkningskraft och kommunikationstid påverkar respektive grafikkort. För att undersöka påverkan av andra olikheter, exempelvis att det integrerade grafikkortet delar flera resurser med processorn, kördes testerna upprepade gånger, där olika delade resurser utnyttjades till max av processorn under körningen.

Resultaten från testkörningarna visade tydligt att det integrerade grafikkortet presterade bättre än det separata grafikkortet för mindre mängder data, trots att det separata grafikkortet är betydligt kraftfullare. Detta beror till stor del på det lägre kommunikationstiderna till processor och system-minne, men även på att det separata grafikkortet kräver större mängder data för att kunna utnyttja hela sin kapacitet. För beräkningar på större mängder data var resultatet lite mer varierande, men då mängden beräkningar växte märktes det tydligt att det separata grafikkortet var kraftfullare. Däremot fanns det flera situationer då det integrerade grafikkortet ändå var snabbare för stora mängder data tack vare de låga kommunikationstiderna.

Utifrån resultaten kan man dra slutsatsen att integrerade grafikkort helt klart är intressanta att använda inom högprestandaberäkningar så länge man jobbar på mindre mängder data. De här testerna har däremot enbart fokuserat på programmets körtider, vilket inte alltid är det viktigaste. Ett exempel skulle kunna vara fall där dubbelt så lång körtid inte spelar så stor roll om strömförbrukningen är betydligt lägre, och integrerade grafikkort skulle därför kunna vara intressanta även i ett bredare spektrum av beräkningsproblem.

Contents

1 Introduction

1.1 Background

Due to the nature of computer graphics, GPUs (Graphics Processing Units) are built for highly parallel tasks. A high throughput is achieved by including a large number of computation cores working in parallel, resulting in a very high potential performance. As GPUs have evolved, the possibilities of using this computation power for other tasks, a concept known as GPU computing, opened up and today they are used for, among other things, HPC (High Performance Computing) [1].

Important factors to consider when using GPU computing are system-specific properties such as latency and bandwidth between the CPU and GPU. There are also algorithm-specific properties, such as the arithmetic intensity of the algorithm, and requirements for communication and synchronization, that can cause a huge impact on the performance [2].

In the past, the main way of getting access to a GPU was to add a discrete extension card. Lately it has become more and more common to include an integrated GPU onto the CPU-chip, but extension cards are still common as well, especially for more powerful GPUs.

Integrated GPUs are located on the same chip as the CPU, giving them the advantage of fast communication with the CPU, but also the disadvantage of sharing resources such as power and memory bandwidth. The memory bus has a rather low latency and high bandwidth (50 GB/s), allowing for relatively fast memory access [3, 4].

Discrete GPUs are generally connected to the system via a PCIe (Peripheral Component Interconnect Express) bus. As they are located on a separate extension card with a dedicated power supply, they are generally more powerful than integrated GPUs. Due to the high latency and low bandwidth (16 GB/s) of the PCIe bus, communication with the CPU is very slow in comparison to an integrated GPU. As memory operations to RAM are performed via the PCIe bus as well, a discrete GPU is much slower than an integrated GPU in terms of RAM access. A discrete GPU does however have its own memory with an extremely high bandwidth (288 GB/s) and rather low latency, allowing computations to be performed at a very high performance once the data resides in GPU memory [2].

In conclusion, discrete GPUs are faster than integrated GPUs once data resides in GPU memory. However, due to the low bandwidth and high latency of the PCIe bus, the transfer of data to and from system RAM and communication with the CPU is much slower for discrete GPUs than for integrated GPUs.

1.2 Related work

As GPUs have become easier and more flexible to program, there is a large, and increasing, number of publications on GPU computing and its uses. Many of those focus on algorithms mapped to be better suited for GPUs, or various areas where GPU computing is useful, including image processing, optimization, simulations, etc, underlining the possibilities and usefulness of GPU computing [5, 6, 7, 8].

As integrated GPUs are relatively new, especially from an HPC point of view, there is a limited number of studies. The existing publications are however positive to the possibilities of using integrated GPUs in certain situations, making this an interesting area [3, 9, 10].

1.3 Thesis description

The aim of the thesis is to analyse the differences of an integrated GPU compared to a discrete GPU with respect to HPC. While integrated GPUs are less powerful than discrete GPUs, they have the advantage of lower latency and higher bandwidth. As both latency, bandwidth, and computation power affects the total time to solve a problem it is interesting to find break even for different kinds of problems and problem sizes. The goal of the thesis is to find a practical situation were the integrated GPU performs better than the discrete GPU in terms of total execution time.

Variables taken into consideration are the differences in latency and bandwidth, as well as usage of resources shared with the CPU. The tests are based on the hypothesis that an integrated GPU can perform better than a discrete GPU for problems with a lower arithmetic intensity, especially for smaller datasets, due to the higher bandwidth and the lower latency between GPU and CPU.

In this thesis the following questions will be answered:

- how does the usage of shared resources affect the performance of the integrated GPU?
- is there a practical situation where an integrated GPU performs better than a discrete GPU in terms of throughput?

All tests are based on a simple vector increment kernel, which takes a vector of a certain length, and performs a certain number of multiplications on each element of the vector. By changing the length of the vector, various sizes of datasets can be tested, while changing the number of multiplications simulates algorithms of various arithmetic intensity.

To test the effects of shared resources, the same test has been run again, once with the CPU working at 100%, and once when the CPU is using as much of the memory bandwidth as possible. To simulate a more realistic execution, the test has also been run by sequentially queueing the same kernel 10 times. This also shows the effects of synchronization on the GPU.

The results show that integrated GPUs can be very useful for computations, as they outperform the discrete GPU in many situation, even though they are less powerful in terms of computation speed. This is clearest for smaller datasets $\leq$ 16kB, as the integrated GPU beats the discrete GPU in all tests, but can also be seen for datasets up to $1MB$ for lower arithmetic intensities. It should however be noted that the CPU can outperform both the GPUs for arithmetically intensive calculations on small datasets due to the lack of parallelism.

2 GPU

Handling graphics on a computer is a rather expensive task in terms of computations, and to offload the CPU it is common to use a separate computation unit known as a GPU. The first GPUs were fixed-function special-built hardware only to complete a process known as the graphics pipeline. The pipeline consists of several stages of processing to eventually convert a data representation of a scene or an object to an image on the screen. Each of these stages are highly data-parallel as they do the same computation for all vertices or fragments depending on the stage.

As computer graphics evolved, the need to control the pipeline increased. The solution was to allow the programmer to write functions, known as shaders, to run on each object in a certain stage of the pipeline. This resulted in greater flexibility as previous GPUs only had built-in configurations. As the needs for flexibility eventually increased, GPUs were evolved to give the programmer control of the entire GPU, and not only each stage. This simplified the usage of GPUs for computations beyond graphics, known as GPU computing, and made GPUs much more popular and convenient for HPC (High Performance Computing) [11, 12].

2.1 GPU architecture

A GPU generally consists of a large shared memory, a number of compute units, and one or many thread dispatchers. Each compute unit contains an instruction cache, a number of processing cores, and local memory shared among the cores in the compute unit, as well as registers shared between all threads in the compute unit. Each core consists of a SIMT-unit (Same Instruction Multiple Thread) of a certain vendor-dependent vector width. A conceptual example of a GPU can be seen in figure 1.

A GPU also includes several graphics-specific components which are of less importance in GPU computing, and will therefore not be explained in detail [2, 9].

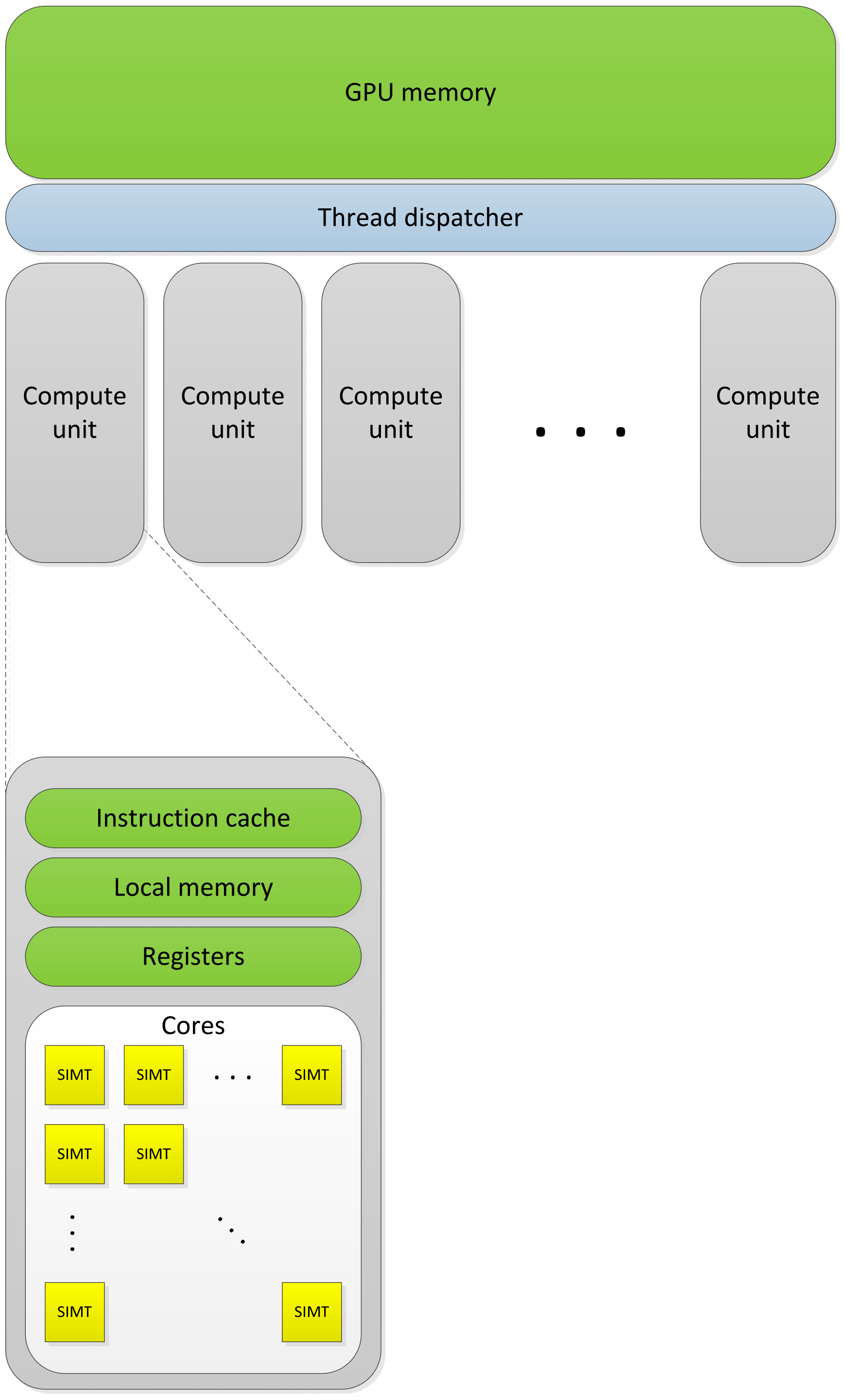


Figure 1: An illustration of a general GPU architecture. A GPU generally consists of GPU memory, one or many thread dispatchers, and a set of compute units. Each compute unit contains local memory, registers, instruction cache, and a set of SIMT-units.

The general work-flow when executing a program on a GPU is that the dataset is divided into smaller parts known as workgroups (AMD), or thread blocks (Nvidia), which are distributed among the compute units by the thread dispatcher. A compute unit can work on several workgroups at the same time but only one program for all workgroups. Each workgroup is divided into wavefronts (AMD), or warps (Nvidia), which, in turn, consists of a set of workitems (AMD), or threads (Nvidia). All workitems in a wavefront are run simultaneously with a shared program counter, which reduces the amount of control logic for each core. This does however cause branches in the code to become a potential problem as the wavefront must go through all directions that is required by at least one workitem. This is highly inefficient as all workitems in a wavefront will spend clock cycles in a compute unit even though only some of the workitems, the ones choosing the current direction of the branch, are progressing.

To utilize the compute units a concept known as latency hiding is used. As soon as a wavefront needs to access memory a load-request is sent. The wavefront is then put to sleep until the load is completed. While the wavefront is asleep, other wavefronts can work. By interleaving wavefronts and workgroups, the latency of memory accesses can be hidden, as the SIMT-units can be utilized all the time without stalling during loads.

An important trait of the compute units is that each unit is completely independent of the other compute units. This simplifies the hardware as memory coherence does not have to be accounted for. The downside is that, since all compute units are independent, they cannot communicate or synchronize with each other. This also prevents communication between workgroups within a compute unit as it is impossible to know beforehand how the thread dispatcher distributes the workgroups among the compute units [2, 4].

2.2 Latency and bandwidth

Latency and bandwidth are two important terms when considering GPUs. These terms are constants for each GPU depending on the bus, drivers, and if the GPU is integrated or discrete, and will be used frequently throughout the thesis to evaluate and compare GPUs. The following is an explanation of each term and its potential issues:

- **Latency** is defined as the time it takes from the moment the CPU starts sending the first bit of data to the moment when the GPU receives it. As the data needs to reach the GPU for the GPU to process it, latency cannot be avoided. This is also an issue for CPUs as data must be fetched from memory [4, 13].

- **Bandwidth** is defined as the amount of data that can be transferred each second. This can become an issue for algorithms with a low arithmetic intensity as it might cause the GPU to stall while waiting for data. As in the case of latency, a similar issue exist when the CPU fetches data from memory [2, 13].

2.3 Discrete GPUs

Discrete GPUs are generally added as extension cards, most commonly over the PCIe bus. As extension cards generally have a fair amount of space, the GPU card can be quite large and they often have a separate power supply to increase the power over the 75 Watts provided by the PCIe bus. The larger area and the extra power makes it possible to build extremely powerful GPUs.

The downside of using an extension card is the speed of the bus used. Even though the PCIe bus theoretically allows for communication at a speed of up to 16 GB/s, it is rather slow compared to the bus between CPU and memory, which in theory can reach 50 GB/s, and the memory bus within the GPU, with a theoretical max speed of 288 GB/s. The latency is also rather high due to the physical distance of the card. This causes problems in GPU computing, as smaller computations, and computations requiring a large amount of communication with the CPU, might suffer from the high latency, while computations with larger amounts of data might suffer from the lower bandwidth [3, 9]. Figure 2 illustrates the bottleneck by showing the relative bandwidth sizes.

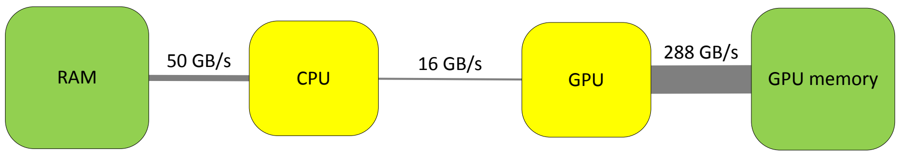


Figure 2: An illustration of a system with a discrete GPU showing the differences in bandwidth. The bandwidth between GPU and GPU memory is much higher than the bandwidth between the CPU and RAM. The GPU does however require data to be sent to GPU memory via the PCIe bus, which has a much lower bandwidth.

2.4 Integrated GPUs

Integrated GPUs are built-in on the same chip as the CPU, a combination also known as APU (Accelerated Processing Unit) [14]. As the need for lighter dedicated graphics hardware increased, integrated GPUs have become more and more common, and though they do not have the same computing power as most discrete GPUs, they are enough for many users.

The main performance gain of an integrated GPU is the fact that it is located on the same chip as the CPU implying that the latency of the communication between CPU and GPU will be decreased drastically. As the GPU is more tightly integrated with the CPU it uses system memory directly over the memory bus at a theoretical speed of 50 GB/s, compared to using the PCIe bus for discrete GPUs at a theoretical speed of 16 GB/s. This should improve the performance for latency- and bandwidth-bound problems.

Figure 3 illustrates the bandwidth relevant for an integrated GPU. Note how we get rid of the low-bandwidth bottleneck between CPU and GPU, but that we also lose the high-bandwidth memory.

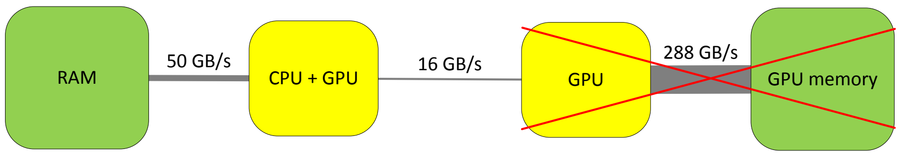


Figure 3: An illustration of a system with an integrated GPU. By using an integrated GPU, a bottleneck is removed, but we also lose the high-bandwidth memory of the dicrete GPU.

Being located on the same chip as the CPU does however bring up a few drawbacks. The GPU has to be much smaller to fit on the chip while a discrete GPU can use much more space, causing integrated GPUs to be less powerful than discrete GPUs [3, 9]. As it shares chip with the CPU it must also share resources such as power and memory. There are various systems for directing the power to the component, CPU or GPU, that the user currently needs, which potentially might cause the GPU performance to decrease if the CPU is highly utilized at the same time. As the memory is also shared, the GPU may perform worse if the CPU is working on a memory-intensive task [14].

3 GPU computing

As the graphics card vendors made GPUs more flexible and easier to program, the idea of GPU computing, also known as GPGPU (General Purpose computations using Graphical Processing Units), became increasingly popular. Due to the large number of computation cores, a GPU can achieve extremely high performance if fully utilized, and the idea of GPU computing is to use this potential computation power for computations beyond graphics.

It is however not always possible to utilize the GPU fully, or even get a speed-up compared to running an algorithm on the CPU. Due to the very limited support for synchronization and thread communication on a GPU, some algorithms are hard to implement efficiently.

As the data needs to be transferred to the GPU, factors such as latency and bandwidth between CPU and GPU are important to consider, as well as different ways of managing the data transfer and the arithmetic intensity of the algorithm, when determining if an algorithm is suitable to run on a GPU [1, 4].

3.1 GPU Kernel

A function designed to run on a GPU is known as a kernel. A kernel consists of a function, but differs from an ordinary function in that a kernel describes the task to perform for each workitem, rather than the entire task. An example for this is a program that adds one to all elements of a vector. While a normal function would consist of a loop that adds one to the current cell and then moves to the next cell, a kernel would be described as add one to a cell. The idea is that this kernel is run once for each cell in parallel, which is exactly how shaders are designed.

3.2 Synchronization on a GPU

The lack of synchronization between blocks on a GPU can cause problems in many algorithms, and there are mainly four ways to work around this.

The first solution is to let the CPU deal with the synchronization by sending data to the CPU between kernel executions, the worst case of all data just getting sent back and forth. This requires a lot of communication, which raises the total computation time, especially for devices with a high latency. This can however be useful for algorithms with extensive sequential parts that cannot be parallelized, instead of running the serial part on the GPU, which is possible, but inefficient due to the parallel nature of GPUs.

The second solution is to queue all tasks to the GPU as separate kernels, with a directive to do them in sequence. By doing this, one task is completed before the second task starts which results in a global synchronization of the data in GPU memory. This is generally much more efficient than the first method as the latency to the CPU only occurs once instead of once for each task.

The third solution is to implement synchronization in the algorithm. Studies have shown that a speed-up can be gained this way. The downside is that the algorithm gets more complex [4, 15, 16, 17].

The fourth solution is to use other algorithms that require less synchronization, and thereby are more suited for a GPU. The downside of this solution is that writing a new, more parallel algorithm, generally is expensive, both in terms of time and money, and might even be impossible, especially for problems that are of a sequential nature.

3.3 Memory management

There are mainly two ways to pass data to the GPU, and both solutions have their pros and cons.

The first solution is to copy all the data to the GPU before the kernel launches. On a discrete GPU this implies sending data to GPU memory via the PCIe bus, while a system with an integrated GPU will perform a memcopy within RAM. This might cause a bottleneck as all the data will have to be copied before the kernel can start, especially for discrete GPUs due to the bandwidth and latency of the PCIe bus. However, due to the high bandwidth between GPU and GPU memory, this can make the computations faster once the copy is complete. This method will be mentioned as copied memory in the rest of the thesis.

The second solution is to map the memory location of the data in RAM and send a pointer to the GPU. This allows the kernel to start straight away as no data needs to be copied. For an integrated GPU the memory latency and bandwidth should not be affected as the same memory is used. For a discrete GPU, using mapped memory drastically increases the latency for memory accesses, as each memory request is made from RAM instead of GPU memory. This does however allow the GPU to hide some of the latency as it can perform computations on parts of the data, while the rest is being transferred [2, 18]. Due to the lower bandwidth, this is not as efficient as latency hiding between GPU and GPU memory, but can still improve the performance compared to copied memory, especially when each memory location is only used a few times. This method will be mentioned as mapped memory in the rest of the thesis.

3.4 A simple model

To get a basic idea of the performance of different algorithms on different GPUs, using different memory management setups, a simple model can be created. To be able to do this we need to introduce arithmetic intensity as a factor.

Arithmetic intensity is the ratio of computations compared to the amount of data required for an algorithm. This is an important concept, especially when using fast devices with low bandwidth, as the low bandwidth might cause the overall performance to decrease for algorithms with low arithmetic intensity, even if the device can do the computations fast [13].

The first part of the model illustrates copied memory. When using copied memory, the data needs to be transferred to the device before any computations can take place. This is dependent on the latency, as well as the bandwidth and the amount of data. Once the data resides in GPU memory, the computation speed depends on the number of computations to perform, the amount of data to perform on, the arithmetic intensity of the algorithm, and the computational throughput of the GPU. The model can be seen in equation (1), where n is the number of kernels to execute in sequence, data is the amount of data to run the algorithm on, and computation speed is a measurement of the performance of the device.

$$\text{run time} = \text{latency} + \frac{\text{data}}{\text{bandwidth}} + n \times \frac{\text{data} \times \text{arithmetic intensity}}{\text{computation speed}} \tag{1}$$

The second part of the model illustrates mapped memory. When using mapped memory, the data transfer needs to be done for every computation instead of only once. However, all but the initial latency will be hidden by the computation time, given that the computation takes more time than the data transfer. Depending on the bandwidth and the arithmetic intensity, the computation time will either hide the transfer time, or be inaccurate as the GPU has to wait for data to transfer between the computations. The model can be seen in equation (2), where n is the number of kernels to execute in sequence, data is the amount of data to run the algorithm on, and computation speed is a measurement of the performance of the device.

$$\text{run time} = n \times \left(\text{latency} + \text{MAX}\left(\frac{\text{data}}{\text{bandwidth}}, \frac{\text{data} \times \text{arithmetic intensity}}{\text{computation speed}}\right)\right) \tag{2}$$

This simple model is enough to quickly gain hypothetical information about the performance of an algorithm on various devices, using the different memory management methods.

3.5 Frameworks for GPU computing

To be able to use GPU computing a framework suited for the GPU must be used. Many GPU vendors have their own framework for the programmer to get access to the GPU, Nvidia's CUDA (formerly Compute Unified Device Architecture) being the most popular.

OpenCL (Open Computing Language) is another popular framework, developed by the Khronos Group as an open cross-platform standard. The main advantage of OpenCL is its cross-platform capabilities as it includes many major vendors including Intel, Nvidia, and AMD. The downside is that OpenCL is much less hardware specific than for example CUDA, as it has to support a larger variety of hardware. However, it has been shown that OpenCL still is a fair competitor when it comes to HPC compared to vendor-specific models [19, 20].

4 Integrated vs discrete GPU in theory

Using the model in section 3.4, combined with the theory about integrated and discrete GPUs, some conclusions can be drawn.

Due to the high latency between CPU and discrete GPU, the size of a problem is important as the initial latency cannot be removed. This is an even greater issue when using copied memory, as the low bandwidth increases the initial time. Problems with a low arithmetic intensity might suffer from the low bandwidth as the computations might not be enough to hide the transfer for mapped memory, or make up for the initial transfer when using copied memory.

However, due to the extremely high bandwidth between GPU and GPU memory, and the possibility to hide latency by interleaving work-groups, memory accesses on a discrete GPU are very cheap as soon as data resides in GPU memory. Integrated GPUs also have the concept of latency hiding, but with a lower bandwidth. They are also less powerful in general, making copied memory on a discrete GPU the fastest for problems with a large amount of memory accesses, as long as the computation time is relatively long compared to the memory transfer time

For smaller problems, or problems that have to communicate with the CPU frequently, the high latency between discrete GPU and CPU becomes an issue as the time to send the data to the GPU might be relatively large, or even exceeds the time saved by doing the computations on the GPU instead of using the CPU.

In contrast, integrated GPUs have lower latency and higher bandwidth to the CPU, being on the same chip. Theoretically this should improve the performance, compared to running on a discrete GPU, for small problems and problems that need to communicate with the CPU frequently. Algorithms with low arithmetic intensity should also be more efficient on an integrated GPU due to the higher bandwidth.

Though the integrated GPU have several advantages due to the lower latency and higher bandwidth, discrete GPUs are generally more powerful. For that reason, an algorithm must benefit greatly from the advantages for an integrated GPU to beat a discrete GPU.

5 Experiments

In this section we explore the possibilities of using integrated GPUs for HPC. The benchmarks will focus on properties that differ between integrated and discrete GPUs. The experiments will start with a simple vector increment kernel to investigate the effects of latency and bandwidth for both discrete and integrated GPUs. The purpose of this benchmark is to determine the impact of the communication overhead and arithmetic intensity, as well as to create a reference for the following benchmarks.

Due to the potential drawback of the integrated GPU sharing resources with the CPU, the benchmark will be run again, once when the CPU is working on a CPU-intensive task, and once when the CPU is working on a memory-intensive task. Finally a benchmark with multiple sequential kernels will be run. The purpose of this benchmark is to simulate real-life applications, as well as to find potential effects from synchronization.

The benchmarks will be run using both mapped and copied memory to find the effects of the memory management, especially when the CPU is working on a memory-intensive task, as the bandwidth to the memory is more likely to become a bottleneck.

The next section will describe the system used to run the benchmarks, followed by a section for each benchmark, containing a more detailed description of the benchmarks along with the results.

5.1 Test equipment

The test system consists of the following hardware:

- APU: AMD A10-5800K 3,8 GHz with Radeon HD7660D
- Discrete GPU: PowerColor Radeon HD6970, 2 GB GDDR5 PCIe 2.1 x16
- RAM: Kingstone DDR3 1866MHz, 8 GB
- Motherboard: Gigabyte GA-F2A85X-UP4
- Hard drive: Seagate Barracuda 1 TB
- PSU: Corsair CX750M 750 W

As not many other vendors offer both integrated and discrete GPUs, AMD was chosen for the benchmarks. It should be noted that, while the APU was the latest high-end desktop device available from AMD at the start of the project, the discrete GPU is almost a year older. The reason for this is that the integrated GPU, Radeon HD7660D, is directly based on an architecture under the codename Cayman, which is used in the Radeon HD6900-series of discrete GPUs [21, 22].

By comparing GPUs by the same vendor, based on the same architecture, the impact of architecture-specific influences on the results are minimized. The drawback of this is that the discrete GPU is not up-to-date, with the main issue that an older PCIe bus standard is used, limiting the bandwidth between CPU and discrete GPU.

The APU uses AMD Turbo Core, which directs power to where it is needed. If the APU is idle, all cores, CPU and GPU, will clock down to save power. When a program is started, more power will be directed to the device that needs it, i.e. to the CPU for CPU-intensive programs, and to the GPU for GPU-intensive programs. However, when both the CPU and the GPU are being used, the power budget might not be enough to run both the GPU and all CPU-cores att full speed, making it interesting to test the GPU performance while the CPU is working [14, 23].

The test system uses the following software:

- Operating system: Windows 8 Pro, 64-bit
- GPU driver: AMD Catalyst 12.104
- Compiler: MinGW g++ 4.7.2

5.2 Vector increment benchmark

The vector increment benchmark is used as a first benchmark to measure the benefits of the lower latency and the higher bandwidth to the CPU for an integrated GPU. In the benchmark, we perform the operation $\vec{v} = \vec{v} \times 1.00001$ multiple times on vectors $\vec{v}$ of different sizes. This is a simple way to find the boundaries of when the integrated GPU beats the discrete GPU as changing the size of the vector will show the effect of the difference in latency, while the various number of increments will show the effect of the difference in bandwidth and computing power as it changes the arithmetic intensity.

The benchmark is run on the integrated and the discrete GPU to allow for comparisons, but also on the CPU. The reason for this is to make sure that problem sizes for which the CPU might be fastest can be found.

5.2.1 Implementation

The benchmark is implemented as a kernel consisting of a single loop, iterating a set number of times over a vector. The kernel did originally perform the operation $\vec{v} = \vec{v} + 1$, but this did not work as the compiler optimized the loop, which made the execution time constant. An optimization that has been made is to insert the number of iterations through a `#define`-statement instead of being passed as an argument. This allows the compiler to unroll the loop at compile-time, making the number of iterations correspond to the arithmetic intensity more accurately as there are fewer clock-cycles wasted on calculating the comparison in the loop.

It should also be noted that the vector element is only read from global memory once, and stored in private memory. The reason for this is not only to improve the performance, but also to make sure that the variation in number of iterations affects the arithmetic intensity, and not the amount of memory accesses.

The kernel code can be seen in listing 1.

```
#define ITERATIONS 128
kernel void vector_add(global float *v)
{
    int gid = get_global_id(0);
    float data = v[gid];

    for(int i = 0; i < ITERATIONS; i++)
    {
        data = data * 1.00001f;
    }
    v[gid] = data;
}
```

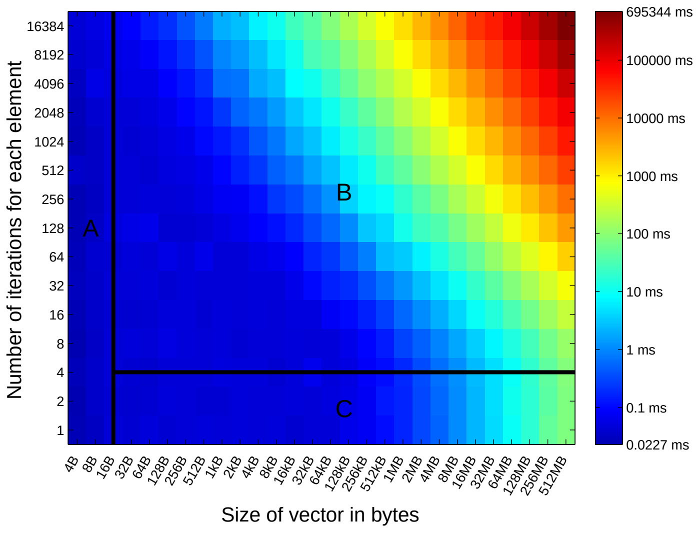
Listing 1: The OpenCL kernel used for the vector increment benchmark. The #define-statement is not included in the kernel source file, but inserted by the program. An example of the statement is added here to illustrate what the code looks like at compile-time.

The benchmark was run for vectors of length 2^n for $0 \leq n \leq 27$, and by doing 2^x iterations, $1 \leq x \leq 16$, for each element. The vector consists of 32-bit floats, making the size of the dataset range from 4 B to 512 MB. Each instance was run for mapped memory on the CPU and the integrated GPU, and on both mapped and copied memory on the discrete GPU. Copied memory is not used for the CPU or the integrated GPU as that would use RAM in the same way as mapped memory, but with the added overhead of two copy-operations. To get accurate results, the median value of ten runs for each instance is used.

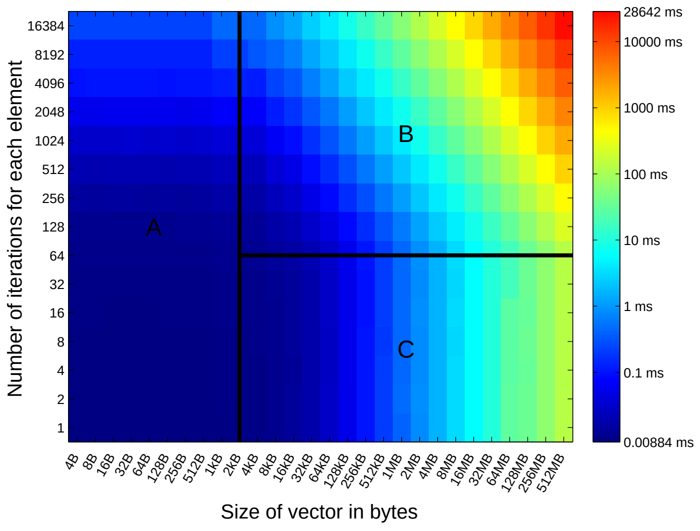
5.2.2 Results

Following is the results for the benchmark. All times are measured in milliseconds and all graphs show the amount of data in the vector, using single-precision floats, on the x-axis, and the number of iterations for each element on the y-axis.

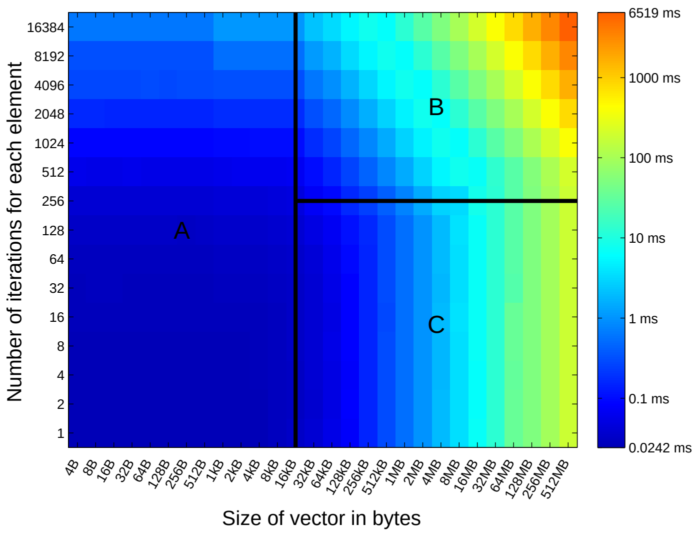
Figure 4 shows the results of the benchmark, where figure 4a shows the execution times on the CPU, figure 4b shows the execution times on the integrated GPU, figure 4c shows the execution times on the discrete GPU using mapped memory, and figure 4d shows the execution times on the discrete GPU using copied memory.



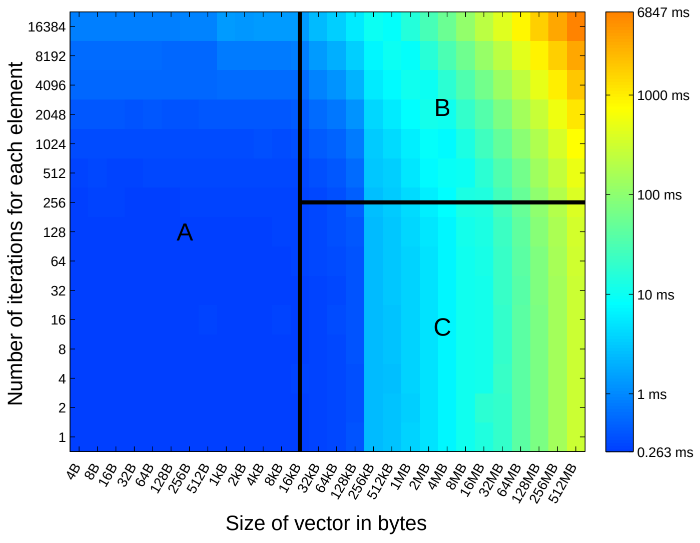
(a) Execution times on the CPU



(b) Execution times on the integrated GPU



(c) Execution times on the discrete GPU using mapped memory



(d) Execution times on the discrete GPU using copied memory

Figure 4: Execution times in milliseconds for the vector increment benchmark.

5.2.3 Discussion

From the results, using both the graphs in figure 4 and the raw results the graphs are based on, we can calculate the bandwidths and latencies of the hardware. The results of these calculations can be seen in figure 5.

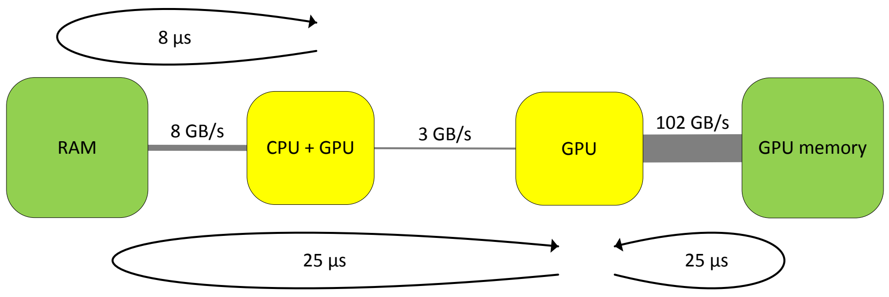


Figure 5: The measured bandwidths and latencies of the benchmark system.

By comparing this figure to figures 2 and 3, showing the specified bandwidths, we can see that there is a major difference. The main reason for the difference is that the theoretical images are based on the latest hardware, using quad-channel RAM and PCIe 3.0, while the benchmark system uses dual-channel RAM and an older discrete GPU using PCIe 2.1.

It might seem surprising that the discrete GPU has the same latency to its on-board memory as to system RAM via the PCIe-bus, even though it is located much closer. This is by design, as a discrete GPU usually hides most of the latencies. There is instead a focus on higher bandwidth, as that is more likely to become a bottleneck for larger amounts of data, which is what GPUs were originally designed for.

Throughput To see if the results are reasonable, the performance, in terms of throughput, can be calculated and compared to the theoretically highest possible throughput for each device. To calculate the throughput of a device we can use the number of operations performed, divided by the execution time.

For the GPUs, the highest throughput is achieved in the top right corner of the graphs in figure 4, which is the most compute intense instance of the benchmark, performing 16384 multiplications on each element in a 512MB array, or about 2199GFLOPs (billion FLoating-point OPerations). By dividing this number with the execution time for each device we can get the number of GFLOPS (billion FLoating-point OPerations per Second) the device can perform.

The CPU does however behave differently, achieving its highest throughput for a lower number of iterations. The reason for this is unknown, but could possibly be because of scheduling problems between the CPU cores and the FPU (Floating Point Unit), as this APU shares one FPU between two CPU cores.

Table 1 shows the measured throughput, together with the specified throughput and the theoretically highest throughput of the device.

Device	**Specified throughput (GFLOPS)**	**Theoretical throughput (GFLOPS)**	**Measured throughput (GFLOPS)**
CPU running the largest problem instance	121.6	7.6	3.16
CPU at its highest throughput	121.6	7.6	9.57
Integrated GPU	614.4	76.8	76.78
Discrete GPU using mapped memory	2703	337.9	337.33
Discrete GPU using copied memory	2703	337.9	321.17

Table 1: Specified, theoretical, and measured throughput for each device. The specified throughput is the highest possible throughput reachable for the device, the theoretical throughput is the highest possible throughput reachable for the device with the benchmark code, and the measured throughput is the throughput that was reached in the benchmark. All measured throughputs are based on the largest problem instance unless otherwise specified.

As can be seen in table 1, the measured throughput is not even close to the specified throughput. The reason for this is that the specified throughput is based on vector operations. The CPU uses AVX (Advanced Vector eXtensions) which performs vector-operations on vectors of 256 bits, allowing each FPU (Floating Point Unit) to perform an operation on eight single-precision floats at once if they are bundled together in a special data-structure. The GPUs use VLIW4 (Very Long Instruction Word 4), which performs vector-operations on vectors of 128 bits. Both devices also support the FMA-operation (Fused Multiply-Add), which performs a multiplication and an addition during the same cycle, doubling the potential throughput.

However, since the code does not use any special AVX/VLIW4-structures and only includes multiplications, a factor eight has to be accounted for when comparing the throughputs of the GPUs, and a factor 16 for the CPU. By comparing the specified throughput divided by 8, with the measured throughput, we get almost exactly the same number for the GPUs, indicating that the hardware is utilized as fully as possible for this code. The GPU using copied memory falls slightly behind as the measured times include the overhead of copying the data.

There is quite a difference when comparing the CPU results though, as it, for some unknown reason drops throughput as the arithmetic intensity increases. It also performs better than the theoretically possible maximum for some instances. This could be explained by AMD Turbo Core, as it allows temporary over-clocking of the CPU to up to 4.2 GHz.

Finding a pattern By comparing the graphs in figure 4, it can be seen that all graphs follow the same pattern, more or less. The pattern forms a line, which can be used to divide the graph into three areas, as shown in figure 6. By analyzing each area we can find the differences between the devices, and also get a clearer overview of the results.

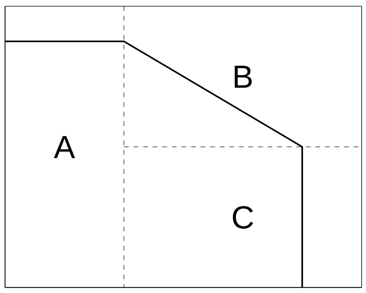


Figure 6: The area of test results is divided into three areas, given by the pattern in the graph.

Area A mainly consists of horizontal lines, implying that the execution time depends almost exclusively on the arithmetic intensity. The reason for this is that the amount of data is so small that the bandwidth barely affects the execution time. The only transfer time to include in this case is the latency, which is constant. It also shows that the device is underutilized as increasing the amount of data does not increase the execution time. This area is almost non-existent on the CPU, as it can be fully utilized using only a few threads, while a GPU, due to its parallel nature, requires more threads to run efficiently.

Area B consists of lines decreasing as the size of the vector increases. This indicates that the execution time depends on both the amount of data and the arithmetic intensity. As both the amount of data and the arithmetic intensity affects the execution time, the problem instances in this area are limited only by computation power. This region is much larger for the CPU than for the GPUs, again because of the low number of cores.

Area C consists of vertical lines, indicating that the execution time depends almost exclusively on the size of the vector, and not on the arithmetic intensity. This occurs when the execution time is limited by the bandwidth as there are not enough computations to hide the transfer of the rest of the data. This can be seen both for the CPU and the GPUs, and is clearest for the discrete GPU due to the lower bandwidth to system memory.

Mapped vs copied memory By analyzing figures 4c and 4d, it can be seen that mapped memory seems to perform better than copied memory. To make this clearer, a speed-up graph can be created, showing the speed-up of using copied memory, compared to mapped memory for the discrete GPU. The graph can be seen in figure 7.

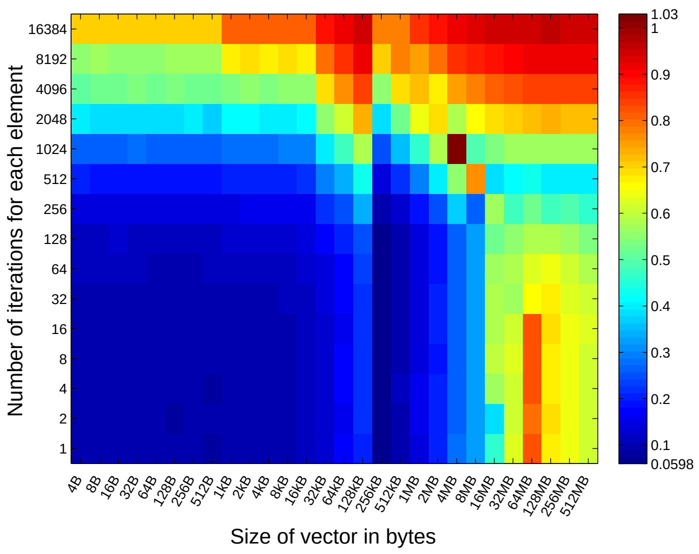


Figure 7: Speed-up for copied memory compared to mapped memory when running the vector increment benchmark on the discrete GPU.

Figure 7 clearly shows that mapped memory is to prefer for discrete GPUs as the speed-up of copied memory rarely exceeds one. This might seem surprising as the discrete GPU has its own memory with a higher bandwidth. This can be explained by the memory accesses of the algorithm. In this benchmark, each memory location is only accessed twice, once for reading and once for writing, which makes hiding the latency more efficient than the overhead of copying all the data.

Except that almost all values in the graph are less than one, the graph also shows a few other interesting features. When running on 256 kB of data there is a sudden drop in performance for copied memory compared to mapped memory, and with 64 MB there is a sudden improvement for low arithmetic intensities. This difference is hard to explain, but could be the effects of different memory management methods or hardware optimizations.

Is the integrated GPU faster? To find out if the integrated GPU is faster for some problems, we have to investigate the speed-ups of the integrated GPU, both compared to the discrete GPU to make sure that we have a speed-up, but also to the CPU to make sure that the problem instance is worth running on a GPU at all.

Figure 8 shows the speed-up of an integrated GPU compared to a discrete GPU. The graph is based on the execution times for mapped memory only, as that was the fastest method for the discrete GPU. The graph clearly shows the potential of the integrated GPU for smaller problem sizes as the speed-up exceeds 2.5 for most vectors of size $\leq$ 4 kB. This was expected as smaller amounts of data should benefit from the lower latency. The graph also indicates that we have a slight speed-up for larger vectors for lower arithmetic intensities. However, for larger vectors and a higher arithmetic intensity, the discrete GPU is almost 5 times as fast as the integrated GPU. The reason for this is that the latency is no longer a problem as it can be hidden during the computation. This also shows the advantage in computation power of the discrete GPU.

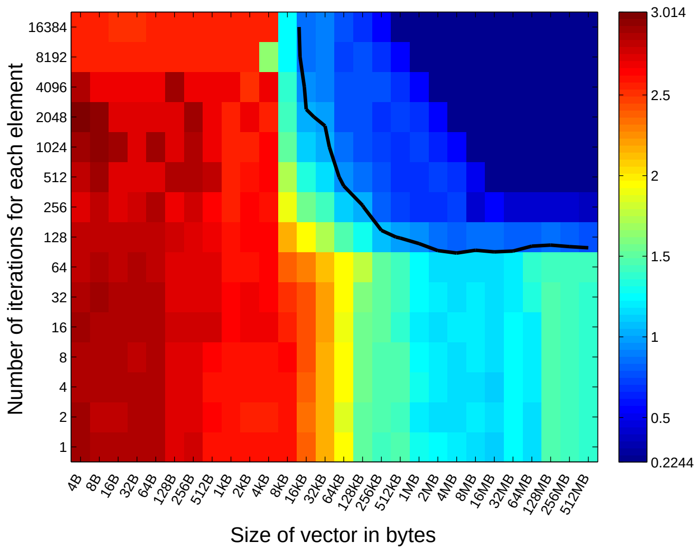


Figure 8: Speed-up of the integrated GPU compared to the discrete GPU. The black line denotes a speed-up of 1, i.e. where the devices are equally fast.

Figure 9 shows the speed-up of the integrated GPU compared to the CPU. The graph shows a major speed-up for larger vectors as long as the arithmetic intensity is rather high. The CPU is however faster for small vectors with a high arithmetic intensity. This is expected as the CPU works at a higher clock frequency, allowing applications with only a few threads to run faster. It is also clear that the CPU performs better for larger vectors, as long as the arithmetic intensity is low. As this part of the graph is the bandwidth-bound area, the CPU seems to have an advantage in memory access compared to the integrated GPU, even though they share memory bus. An explanation for this could be a hardware pre-fetcher, predicting the memory accesses and loading memory in advance.

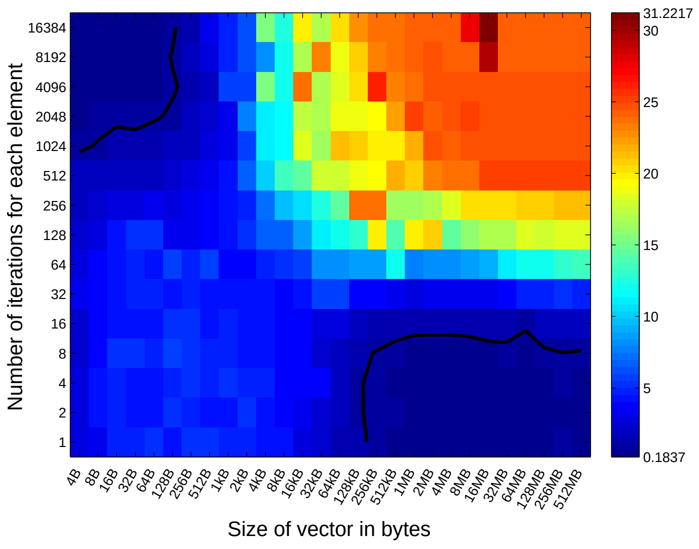


Figure 9: Speed-up of the integrated GPU compared to the CPU. The black line denotes a speed-up of 1, i.e. where the devices are equally fast.

By merging the two previous graphs, and by only considering if a value is greater than 1 or not, we get a graph showing all problem instances for which the integrated GPU is faster than both the CPU and the discrete GPU. This graph can be seen in figure 10. The graph clearly shows that the integrated GPU performs better than both the CPU and the discrete GPU for a large range of problem instances, mainly for vectors of sizes 128 B – 16 kB, where it outperforms the other devices for all arithmetic intensities.

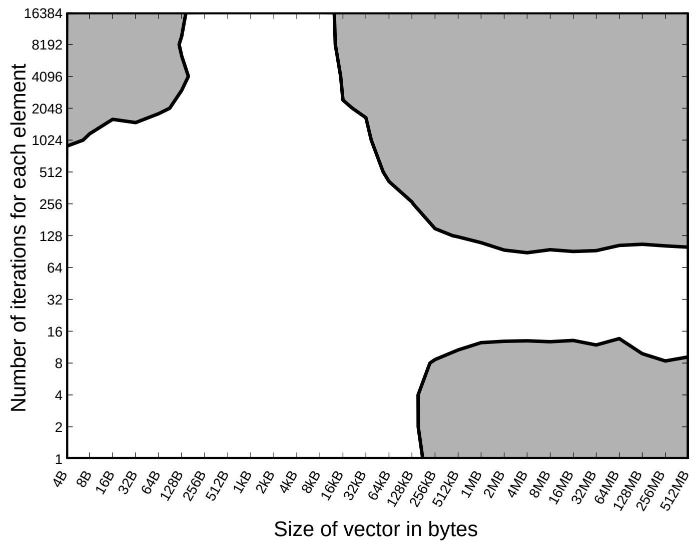


Figure 10: All problem instances where the integrated GPU is fastest, denoted by the white area in the graph.

5.3 Vector increment benchmark with CPU load

The vector increment benchmark with CPU workload is a way of investigating how shared resources affects the performance of a GPU. As the integrated GPU shares power with the CPU, and the power is distributed using AMD Turbo Core, it is likely that the performance drops if the CPU is working at the same time.

The benchmark was run on the integrated GPU, and on the discrete GPU using mapped and copied memory. The CPU was not considered while running this benchmark as it was working on the test settings.

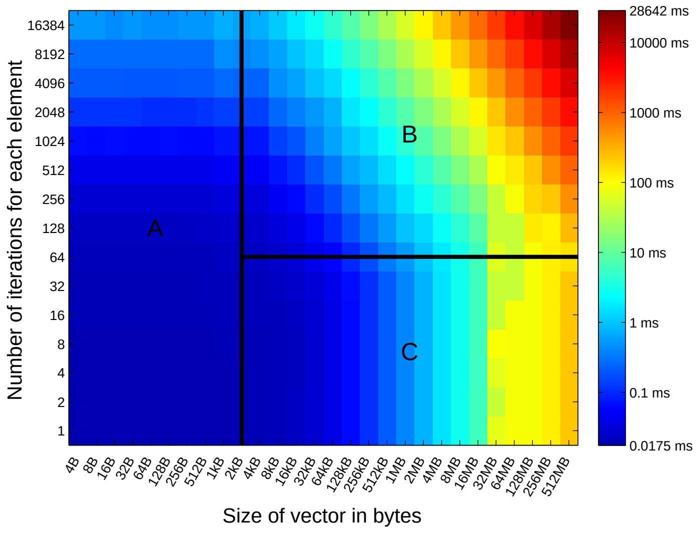
5.3.1 Implementation

The benchmark uses the same kernel as the vector increment benchmark. The only difference is that a second program is running at the same time, utilizing the CPU as much as possible. This program was designed to run an infinite loop, only doing a single multiplication on a float, in the background. The program was run in four instances to utilize all the CPU cores.

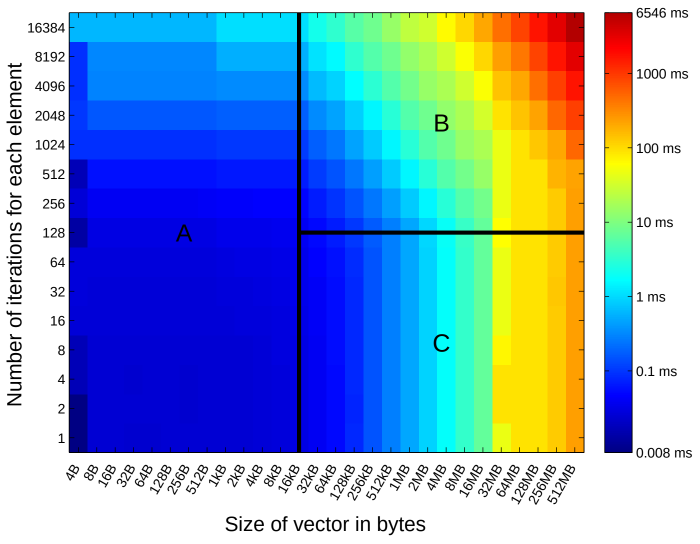
5.3.2 Results

Following is the results for the benchmark. All times are measured in milliseconds and all graphs show the amount of data in the vector, using single-precision floats, on the x-axis, and the number of iterations for each element on the y-axis.

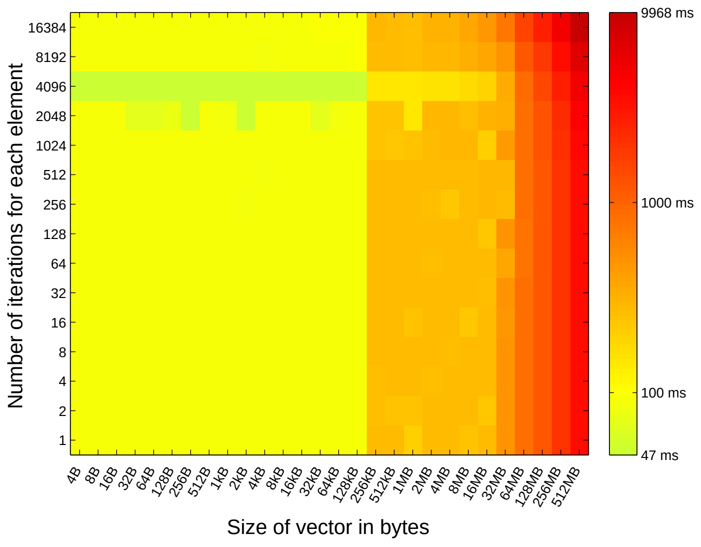
Figure 11 shows the results of the benchmark, where figure 11a shows the execution times for the integrated GPU, figure 11b shows the execution times for the discrete GPU using mapped memory, and figure 11c shows the execution times for the discrete GPU using copied memory.



(a) Execution times on the integrated GPU



(b) Execution times on the discrete GPU using mapped memory



(c) Execution times on the discrete GPU using copied memory

Figure 11: Execution times in milliseconds for the vector increment benchmark when the CPU is working at 100%.

5.3.3 Discussion

By looking at figures 11a and 11b we can see the typical ABC–pattern from figure 6. Figure 11c looks different and needs to be analyzed further. By looking at the numbers the graph is based on, it can be seen that almost all tests using a vector of size $\leq$ 128 kB takes approximately 93 ms, except for a few runs that take about 47 ms.

For more details, we can use a particular instance and look at all test runs, instead of only the median. Table 2 shows the result from all test runs using the discrete GPU with copied memory, for vector size of 2 kB, doing 2048 iterations for each element.

Copy to GPU	**Execution time**	**Copy from GPU**	**Total time**
0.129	0.180	46.698	47.007
0.129	0.178	46.707	47.015
45.828	0.179	46.706	92.713
0.128	0.178	46.696	47.003
0.134	0.178	46.673	46.985
0.146	0.178	46.692	47.015
45.812	0.179	46.637	92.628
0.128	0.178	46.707	47.013
45.850	0.178	46.689	92.718
45.789	0.178	46.665	92.632

Table 2: Copy- and execution times for ten runs of the vector increment benchmark using a discrete GPU and copied memory, for a vector size of 2kB, doing 2048 iterations for each element. All times are measured in milliseconds.

Table 2 clearly shows that the execution, and copying the data back to RAM always takes the same time, while the first copy operation changes between two values, 0.1 ms and 45.8 ms. This is presumed to be because of the CPU scheduling. Some times the copy can be completed in time, taking 0.1 ms. Some times the CPU makes a context switch and lets another process run for about 45.8 ms, before the CPU can switch back and perform the rest of the copy.

Due to this extra overhead, and the fact that the discrete GPU performed worse for copied memory than for mapped memory in the previous test, we have reason to believe that this is true, also when the CPU is working at 100%. Figure 12 shows the speed-up graph of the discrete GPU using copied memory compared to using mapped memory. The graph clearly shows that mapped memory does perform much better than copied memory in all cases, as all values are less than 1. Copied memory will therefore not be investigated further in this test.

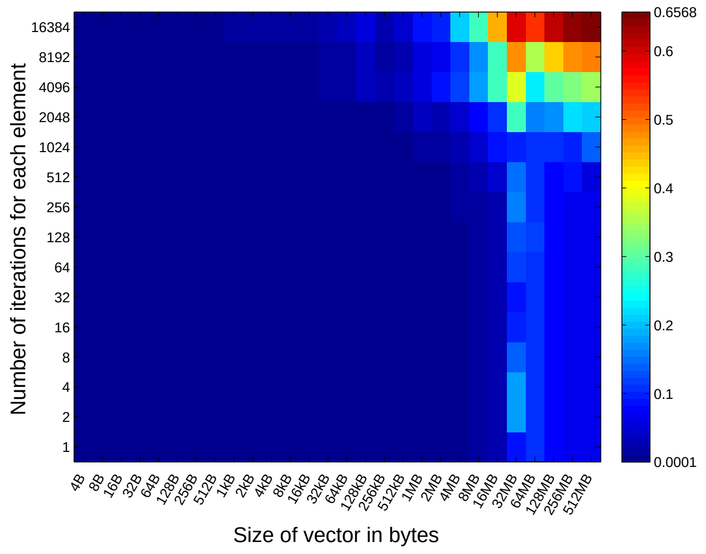
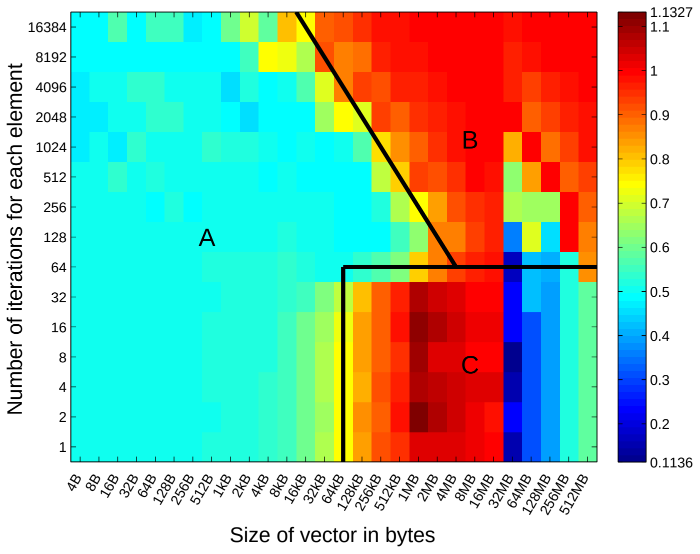
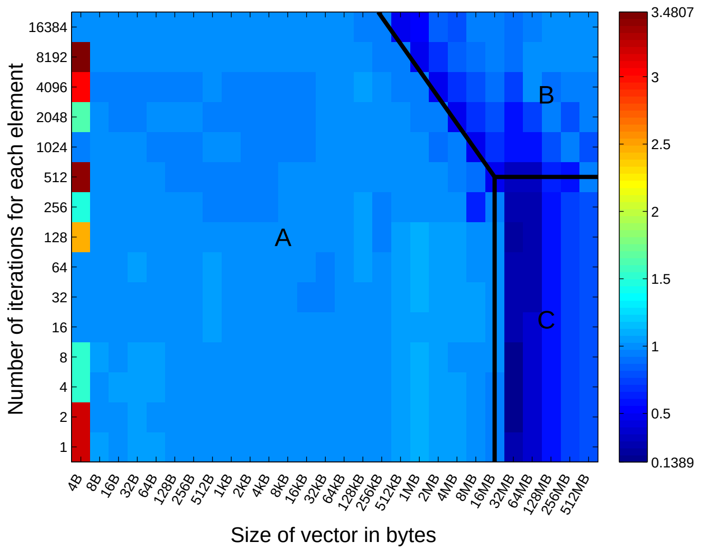


Figure 12: Speed-up of the discrete GPU using copied memory, compared to using mapped memory, for the vector increment benchmark when the CPU is running at 100%.

Effects of the CPU running at 100% To find the effects of the CPU running at 100%, we analyze the speed-ups for each device separately, comparing execution times for the device when the CPU is working to when the CPU is idle. The speed-ups can be seen in figure 13, where figure 13a shows the speed-up for the integrated GPU, and figure 13b shows the speed-up for the discrete GPU using mapped memory.



(a) Speed-up of the integrated GPU



(b) Speed-up of the discrete GPU using mapped memory

Figure 13: Speed-ups when running the vector increment benchmark while the CPU works at 100%, compared to running the same benchmark when the CPU is idle. Note that the graphs use different colour scales. The reason for this is that the outliers in figure 13b make the scale too broad which in turn makes figure 13a hard to interpret if using the same scale.

As can be seen in figure 13, the CPU affects the integrated GPU quite heavily, while the discrete GPU is mostly unaffected. This is expected as the integrated GPU shares power with the CPU, while the discrete GPU has its own power supply.

Area A of figure 13a shows a speed-up of 0.5, indicating that the integrated GPU is clocked down. This makes sense as the problems in this area are rather small, and therefore not enough to make AMD Turbo Core redirect more power to the GPU. Area B and C show a speed-up of 1, indicating that the execution time is the same as when the CPU was idle. The diagonal border of area B shows that the changes of the clock frequency is based on the execution time, while area C indicates that there are also other factors involved as the GPU clocks up for problems of sizes as small as 64 kB.

Figure 13b shows that the discrete GPU is mostly unaffected by the CPU as all speed-ups are close to 1, especially in area A, but also in area B.

Area C in both graphs does however show a major loss of performance for problem instances of size 32MB and larger, especially for lower arithmetic intensities. By investigating the execution times closer for the integrated GPU in that area, we can see that many values are close to 45 ms, 90 ms, 135 ms, which are all multiples of 45, as well as to 67.5, which is the median of 45 and 90. The values of this area from figure 11a is shown in table 3.

As for the case of 45 ms when using copied memory on the discrete GPU, the reason for this could be context switches on the CPU, forcing the task to wait for its turn, even though the computation is finished already. To see if this really is the problem, the benchmark was run again, but this time with real-time priority.

		Vector size					
		4MB	8MB	16MB	32MB	64MB	128MB
Iterations	64	1.7144	3.1915	5.9960	60.8716	43.6146	87.7918
	32	1.5398	3.0000	5.8287	44.8648	43.5773	87.8206
	16	1.5266	3.0012	5.8239	44.8875	90.2197	87.8141
	8	1.5294	3.0049	5.8639	91.4493	90.2117	87.7965
	4	1.5216	3.0091	5.7561	68.1448	90.1962	87.7245
	2	1.5257	3.0016	5.8721	44.9323	90.1940	87.7421
	1	1.5315	3.0095	5.8534	68.2130	90.1911	87.8800

Table 3: Some of the results from figure 11a, showing that something happens for vectors larger than 16MB, most likely because of the CPU scheduling.

5.3.4 Raising the priority of the test

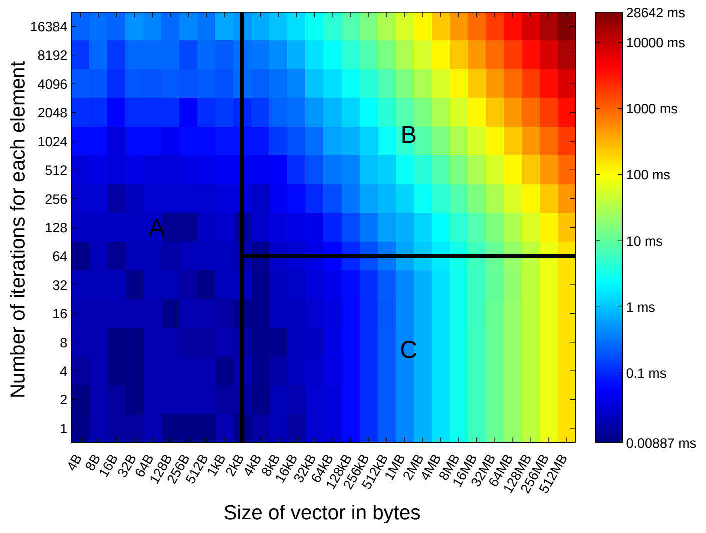
As the test showed signs of interference from the CPU scheduling, the test was run again, but with priority set to real-time. The idea is that all scheduling-related effects on the result should disappear, while the effects of sharing power should persist.

As this seems to be an issue, not only for mapped memory on both devices, but also when using copied memory with the discrete GPU, the test will take up copied memory for testing again.

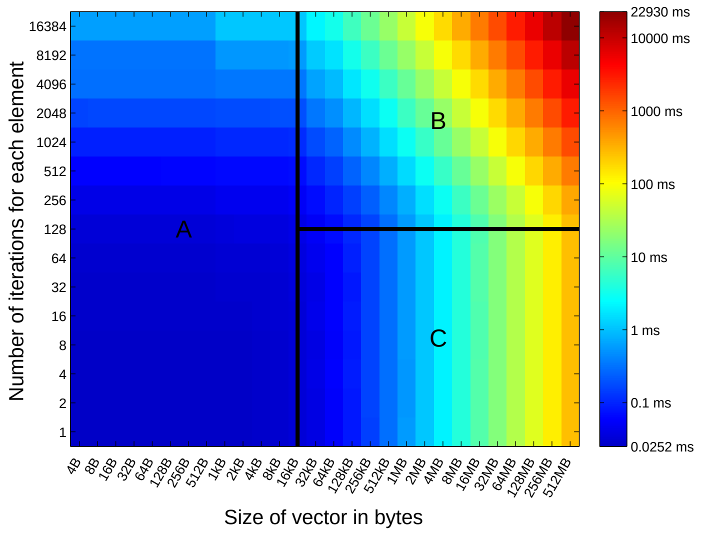
5.3.5 Results when running with raised priority

Following is the results for the benchmark. All times are measured in milliseconds and all graphs show the amount of data in the vector, using single-precision floats, on the x-axis, and the number of iterations for each element on the y-axis.

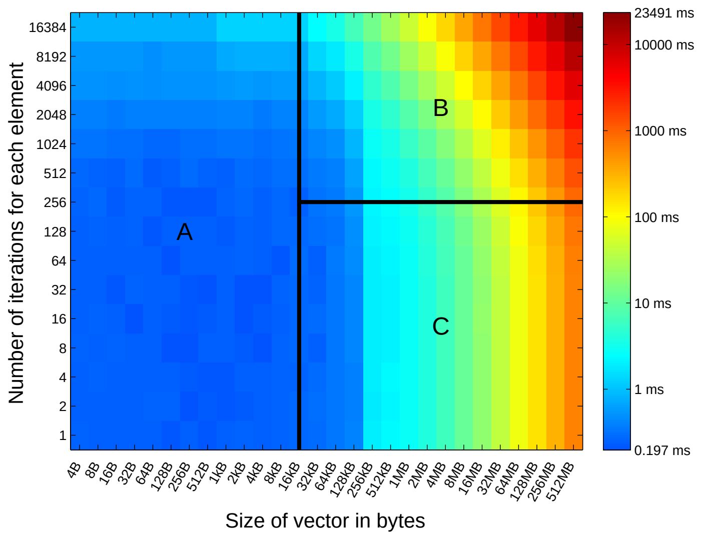
Figure 14 shows the results of the benchmark, where figure 14a shows the execution times for the integrated GPU, figure 14b shows the execution times for the discrete GPU using mapped memory, and figure 14c shows the execution times for the discrete GPU using copied memory.



(a) Execution times on the integrated GPU



(b) Execution times on the discrete GPU using mapped memory



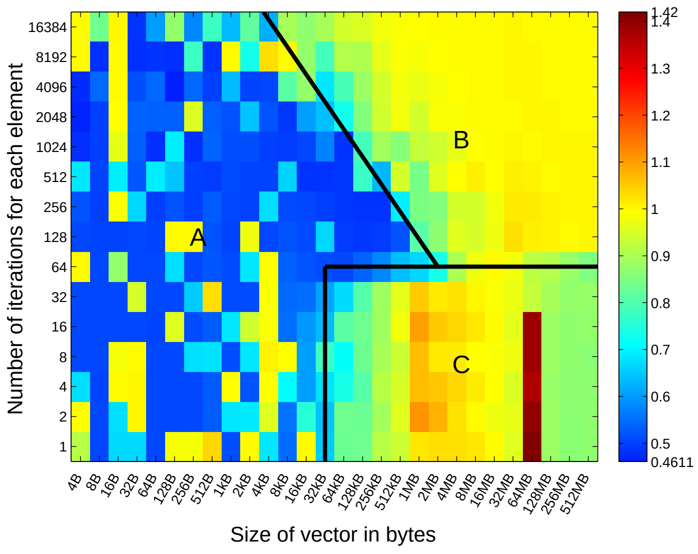
(c) Execution times on the discrete GPU using copied memory

Figure 14: Execution times in milliseconds for the vector increment benchmark with raised priority when the CPU is working at 100%.

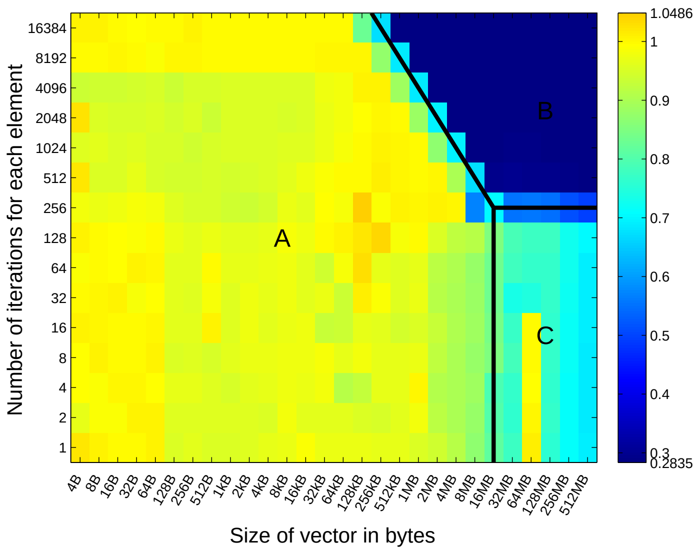
5.3.6 Discussion

As can be seen in figure 14, all graphs now fit with the pattern in figure 6, indicating that the previous results for the discrete GPU using copied data was indeed a side-effect of the CPU scheduling.

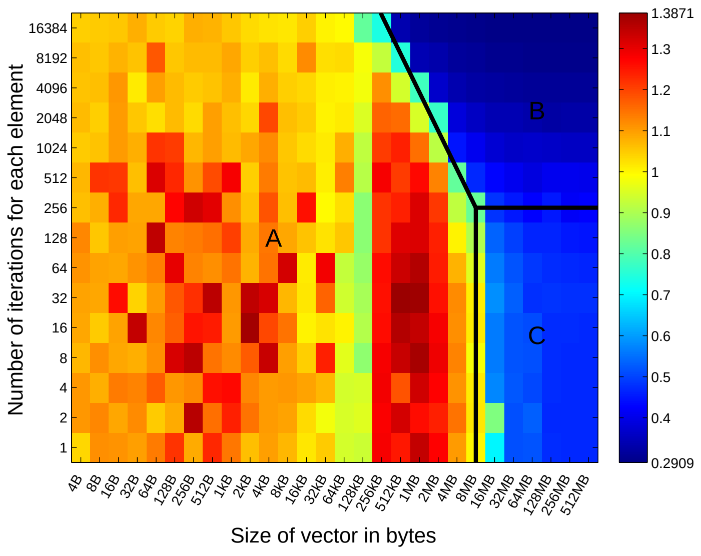
Effects of raised priority To find out if the CPU scheduling was the problem behind the unexpected result for certain vector sizes, we have to investigate the new speed-ups of each device, compared to when the CPU is idle. The new speed-ups can be seen in figure 15, where figure 15a shows the speed-up for the integrated GPU, figure 15b shows the speed-up for the discrete GPU using mapped memory, and figure 15c shows the speed-up for the discrete GPU using copied memory.



(a) Speed-up of the integrated GPU



(b) Speed-up of the discrete GPU using mapped memory



(c) Speed-up of the discrete GPU using copied memory

Figure 15: Speed-ups when running the vector increment benchmark with raised priority when the CPU works at 100%, compared to running the same benchmark when the CPU is idle.

By comparing the previous speed-ups in figure 13 to the new speed-ups in figure 15 we can see that the sudden drop in speed-up for vectors of size 32MB and larger has decreased, indicating that the problem was at least partly caused by the CPU scheduling.

Area A of figure 15 shows that the integrated GPU still works at a lower clock-frequency most of the time. The raised priority does however allow it to clock up occasionally as there are speed-ups of up to 1 even in this area. Area B and C are still mostly unaffected, most speed-ups being close to 1.

Area A in figures 15b and 15c shows that the discrete GPU is unaffected by the CPU working at 100%, and even gain a speed-up for copied memory, most likely because of the raised priority.

There is however a major loss in performance in area B and C. As the performance-drop follows the vertical lines in the bandwidth-bound area C, and the diagonal lines in the compute-bound area C, it seems to follow a certain execution time.

By investigating the execution times closer, it can be seen that the performance-drop occurs for all instances with an execution time of about 20 ms and higher. A possible explanation for this could be that the GPU is used for screen updates while the CPU is busy. As screen updates occur at a frequency of 60 Hz, or every 16.7 ms, this could explain why the performance drop occurs at a very specific execution time.

It should also be noted that the performance spike at 64 MB from figure 7 can be seen also in figure 15 when using mapped memory, both on the integrated and the discrete GPU. Again this is hard to explain, but could be because of hardware optimizations.

Mapped vs copied memory Figures 15b and 15c show that the speed-up of copied memory is slightly larger than the speed-up of using copied memory for the discrete GPU when the CPU is working at 100%. However, as can be seen in figure 16, mapped memory still performs better than copied memory, as the speed-up of copied memory compared to mapped memory never exceeds 1.

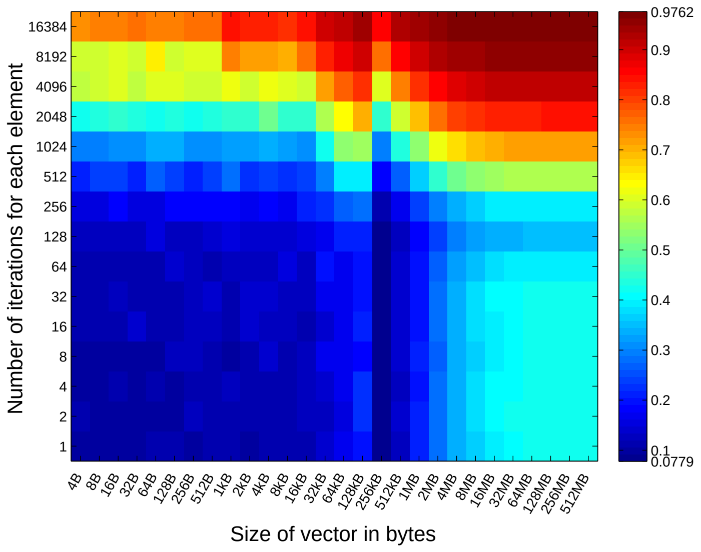


Figure 16: Speed-up of the discrete GPU using copied memory, compared to using mapped memory, for the vector increment benchmark with raised priority, when the CPU is running at 100%.

Is the integrated GPU faster? The speed-up graphs clearly show that the integrated GPU is affected by the CPU to a higher degree than the discrete GPU, especially for smaller amounts of data, where the integrated GPU previously has shown an advantage. Figure 17 shows the speed-up of the integrated GPU compared to the discrete GPU.

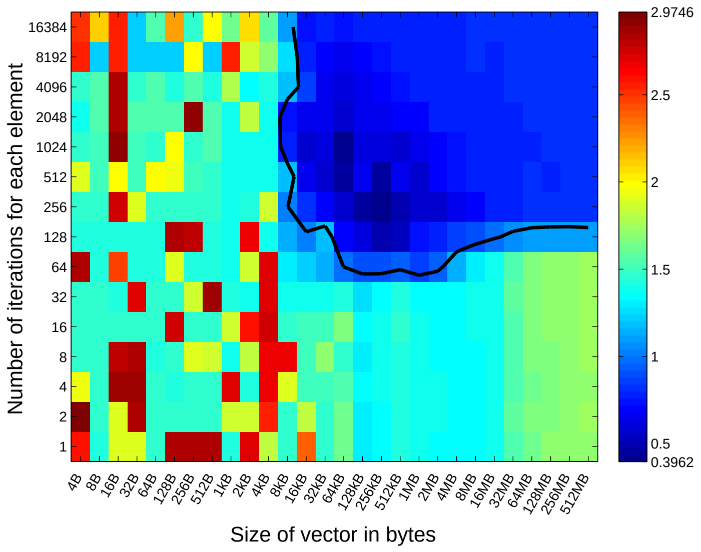


Figure 17: Speed-up of the integrated GPU compared to the discrete GPU for the vector increment benchmark with raised priority and with the CPU running at 100%. The black line denotes a speed-up of 1, i.e. where the devices are equally fast.

The graph shows a speed-up curve similar to figure 8, which shows the speed-up when the CPU is idle. The main difference between the graphs is the magnitude of the speed-up, which clearly shows the impact of under-clocking the integrated GPU for smaller problem sizes.

The integrated GPU is however still faster than the discrete GPU for smaller problems, making it efficient even in situations when the CPU is working intensively.

5.4 Vector increment benchmark with memory load

The vector increment benchmark with memory workload is a way of testing how the shared memory bus affects the performance of a GPU. As an integrated GPU shares memory bandwidth with the CPU, it is likely that the performance drops if the CPU is using a lot of memory at the same time.

The benchmark was run on the integrated GPU, and on the discrete GPU using mapped and copied memory. The discrete GPU has so far performed poorly using copied memory, but as it has its own memory, it could possibly perform better in this benchmark, especially for the instances where the memory bandwidth is a bottleneck. The CPU was not considered at all as it was working on the test settings.

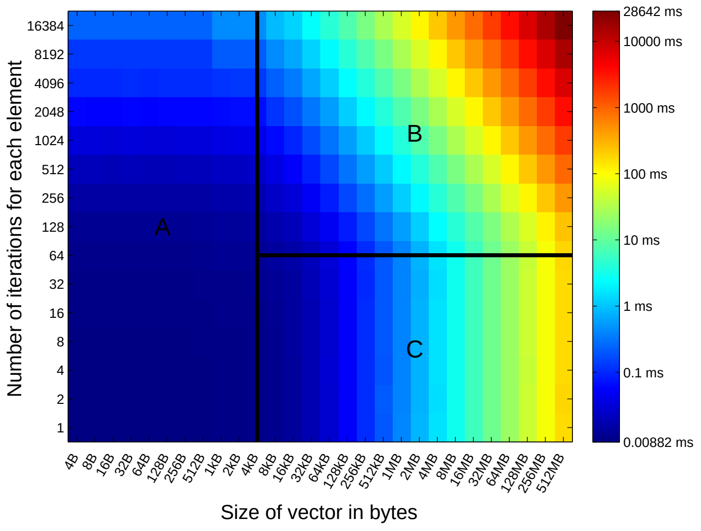
5.4.1 Implementation

The benchmark is using the same kernel as the vector increment benchmark. The only difference is that a second program is running at the same time, utilizing as much of the memory bandwidth as possible. To do this, the program runs an infinite loop, with a nested loop over an array of 33 million floats, performing a multiplication on each cell. The program is run in two instances to raise the number of memory operations, but not use all cores as that could affect the benchmark as in the previous test.

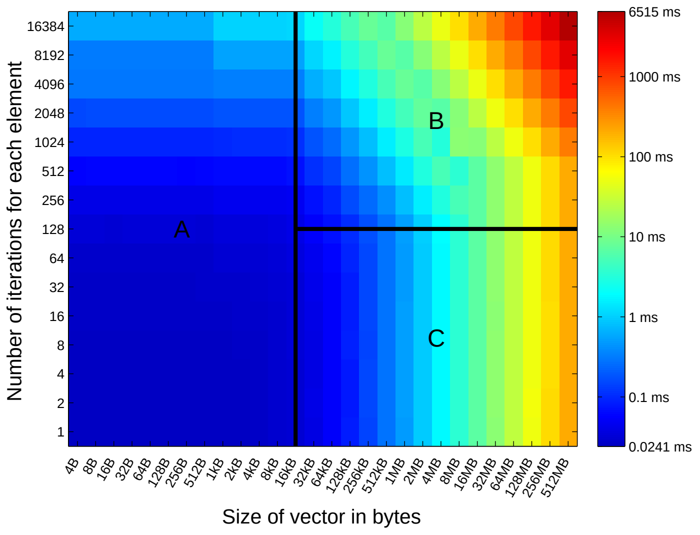
5.4.2 Results

Figure 18 shows the results of the benchmark, where figure 18a shows the execution times for the integrated GPU, figure 18b shows the execution times for the discrete GPU using mapped memory, and figure 18c shows the execution times for the discrete GPU using copied memory.

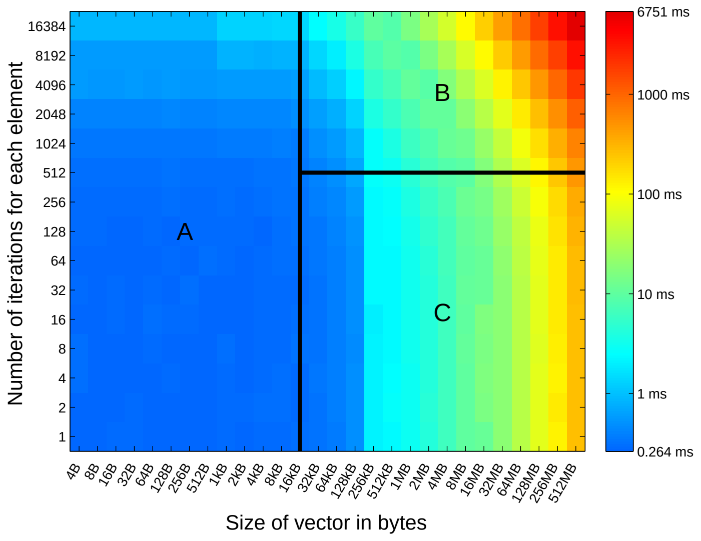
All times are measured in milliseconds, and all graphs show the amount of data in the vector, using single-precision floats, on the x-axis, and the number of iterations for each element on the y-axis.



(a) Execution times on the integrated GPU



(b) Execution times on the discrete GPU using mapped memory



(c) Execution times on the discrete GPU using copied memory

Figure 18: Execution times in milliseconds for the vector increment benchmark when the CPU utilizes as much of the memory bandwidth as possible.

5.4.3 Discussion

Copying data to the GPU has given worse results than using mapped memory in the previous benchmarks. This benchmark could possibly change this as the CPU uses the memory bus during the test runs. Surprisingly, as can be seen in figure 19, mapped memory still outperforms copied memory for all cases except one, which is most likely a measurement error. This can again be explained by the low number of memory accesses.

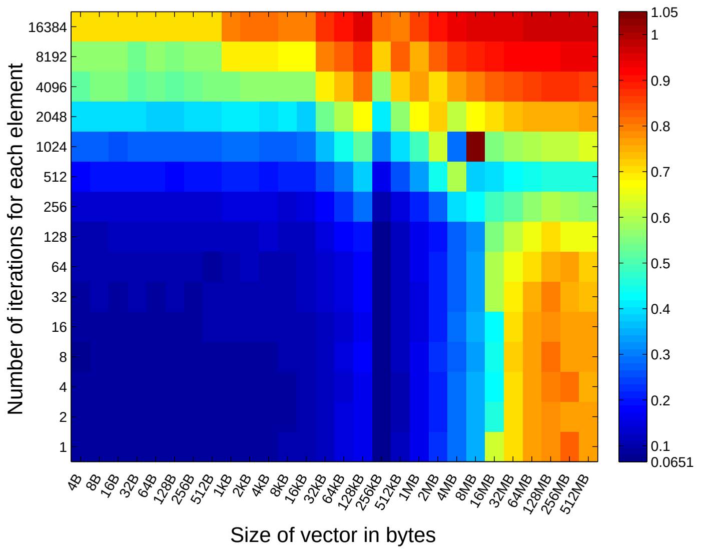
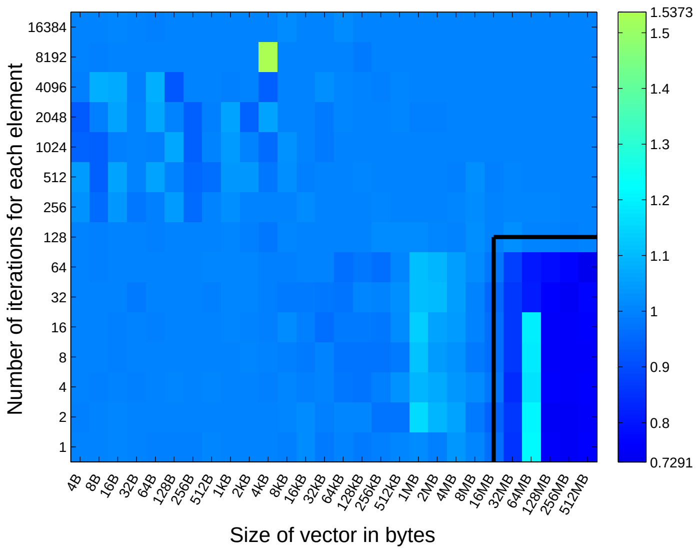
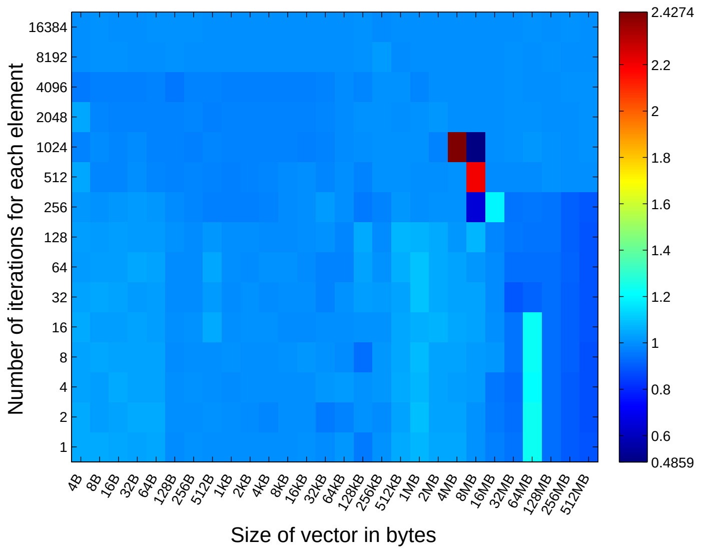


Figure 19: Speed-up of the discrete GPU using copied memory, compared to using mapped memory, for the vector increment benchmark when the CPU utilizes as much of the memory bandwidth as possible.

Effects of the CPU using a lot of memory To find the effects of the CPU using a lot of memory, we analyze the speed-ups for each device separately, comparing execution times for a working CPU to an idle CPU. The speed-ups can be seen in figure 20, where figure 20b shows the speed-up for the discrete GPU, and figure 20a shows the speed-up for the integrated GPU.



(a) Speed-up of the integrated GPU



(b) Speed-up of the discrete GPU using mapped memory

Figure 20: Speed-ups when running the vector increment benchmark while the CPU uses a lot of memory, compared to running the same benchmark when the CPU is idle.

The graphs in figure 20 are roughly the same, all values being close to 1, except for a few outliers, most likely because of measurement errors. There is however an area in the bottom-right corner of figure 20a that is slightly slower, with a speed-up of 0.8 for large vectors and low arithmetic intensities. This is a part of the area that is bandwidth-bound, which makes sense as the purpose of this test is to limit the bandwidth. The difference is rather small though, indicating that the memory bandwidth is of less importance unless the problem is highly bandwidth-bound.

5.5 Chained kernels

The purpose of this benchmark is to simulate real applications closer. While the previous benchmarks have focused only on a single execution of a kernel, real applications commonly perform several kernels in sequence on the same data, for example in image analysis where you might want to perform a Fourier transform, followed by a filter and an inverse Fourier transform [24]. Executing kernels in sequence also implies that the GPU gets global barriers, as sequential kernels cannot launch before the previous kernel has completed, which can be useful for algorithms requiring synchronization. The drawback of stacking kernels sequentially is that it gets harder to fully utilize the GPU, as compute units might be idle while waiting for the last part of a kernel to finish and the next kernel to start.

5.5.1 Implementation

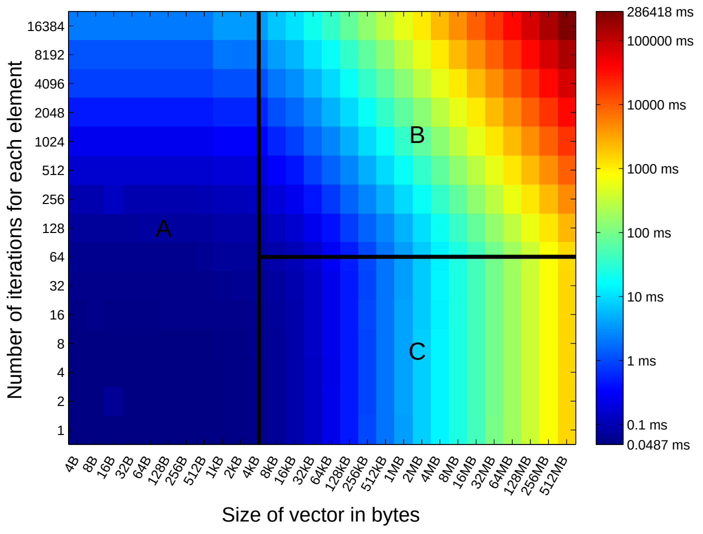
The benchmark is using the same kernel as the vector increment benchmark. The only difference is that the kernel is queued sequentially 10 times, using the same memory object. This will show the effects of multiple kernels, as well as simulate a kernel with

10 synchronization points. This should also show the benefit of using copied memory, as the data is only copied to GPU memory once, while the data needs to be loaded from RAM for each kernel when using mapped memory.

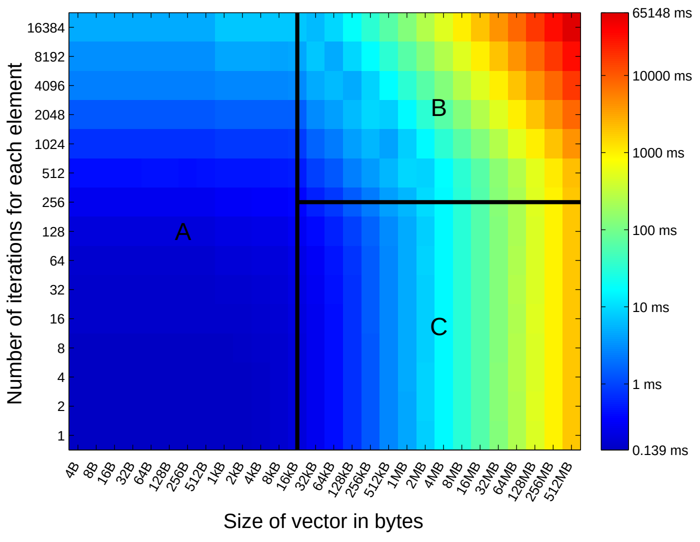
5.5.2 Results

Following is the results for the benchmark. All times are measured in milliseconds and all graphs show the amount of data in the vector, using single-precision floats, on the x-axis, and the number of iterations for each element on the y-axis.

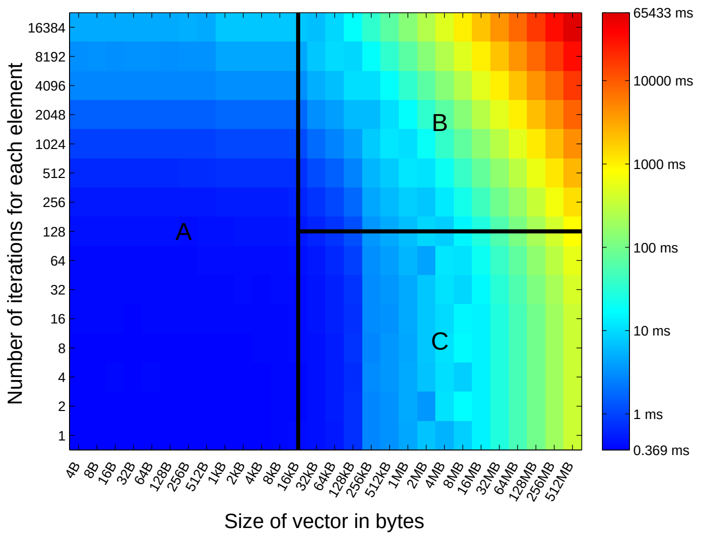
Figure 21 shows the results benchmark, where figure 21a shows the execution times for the integrated GPU, figure 21b shows the execution times for the discrete GPU using mapped memory, and figure 21c shows the execution times for the discrete GPU using copied memory.



(a) Execution times on the integrated GPU



(b) Execution times on the discrete GPU using mapped memory



(c) Execution times on the discrete GPU using copied memory

Figure 21: Execution times in milliseconds for the vector increment benchmark with 10 sequential kernels.

5.5.3 Discussion

Copied memory has so far performed poorly due to the low number of memory accesses. This should change in this benchmark as mapped memory will have to load data from RAM for each kernel, while the copied memory just need to perform the copy-operation once. As can be seen in figure 22, copied memory results in a speed-up, compared to mapped memory, for larger amounts of data and arithmetic intensities up to 64.

This is not surprising as this area of the graph corresponds to the bandwidth-bound problems, indicating that the discrete GPU benefits from the high bandwidth between GPU and GPU memory. The difference between copied and mapped memory decreases as the arithmetic intensity increases, indicating that the memory management is of less importance for compute-bound problems.

It is also clear that copied memory performs poorly for smaller problem sizes with a lower arithmetic intensity. This is surprising considering that this is the latency-bound region, and mapped memory suffers from the latency to RAM ten-fold compared to copied memory. This can however be explained by the latency between GPU and GPU memory. Even though the bandwidth to GPU memory is very high, the latency is quite high, causing a bottleneck when it cannot be hidden by computations.

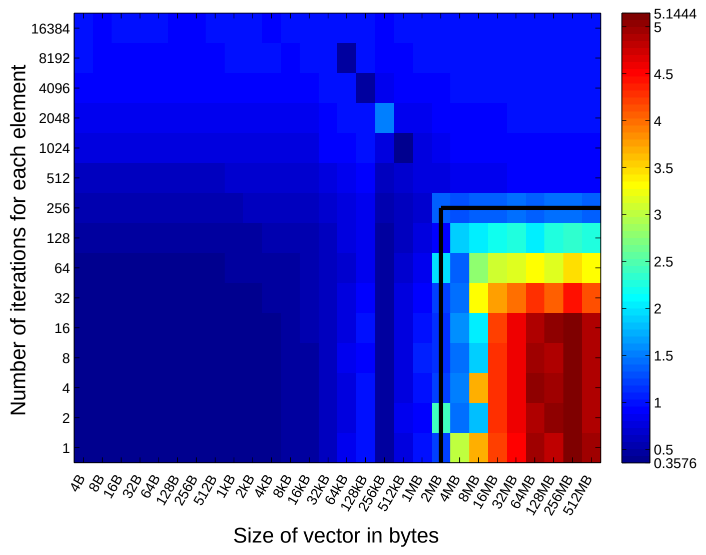


Figure 22: Speed-up of the discrete GPU using copied memory, compared to using mapped memory, for the vector increment benchmark with 10 sequential kernels.

Is the integrated GPU faster? Figure 23 shows the speed-up of the integrated GPU compared to the discrete GPU. The graph considers both mapped and copied memory for the discrete GPU, and the speed-up is calculated to compare to the fastest method for each problem instance. This graph is similar to figure 8, denoting the same comparison, but for only one kernel execution. By analysing the similarities and differences between the graphs, the effects of multiple kernels and synchronization should be found.

As can be seen in both figures, the integrated GPU is always faster for datasets of size $\leq$ 16 kB, and for datasets up to sizes of 1 MB for lower arithmetic intensities. This is expected as this is the latency-bound part of the graph, which should benefit the integrated GPU due to the lower latency to the CPU. The main difference between the two graphs is the bandwidth-bound area. This is already explained as being due to the discrete GPUs superior bandwidth to memory, which is enough for the discrete GPU to outperform the integrated GPU.

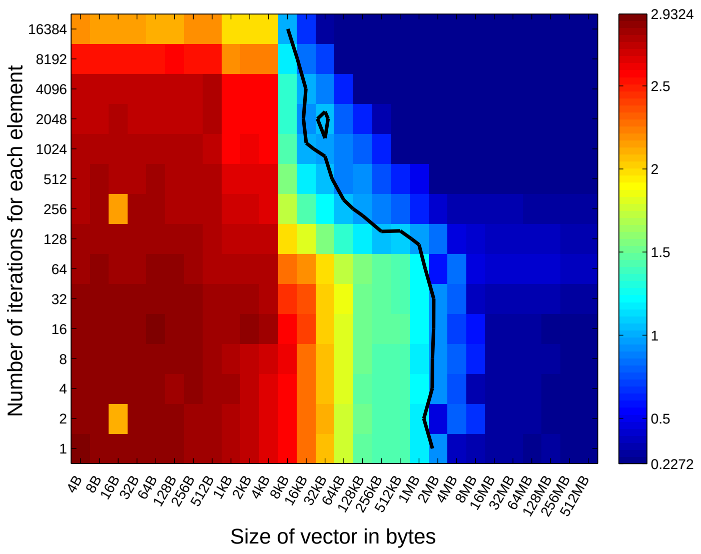


Figure 23: Speed-up of the integrated GPU compared to the discrete GPU, for the vector increment benchmark with 10 sequential kernels. The black line denotes a speed-up of 1, i.e. where the devices are equally fast.

6 Conclusion

The goal of this thesis was to explore the possibilities of using integrated GPUs for high performance computing. This was tested using a vector increment benchmark, which was run under various circumstances, including pushing the CPU, pushing the memory bandwidth, and chaining several kernels. The results clearly show the potential of the integrated GPU, as it outperformed the more powerful discrete GPU in many situations, especially for smaller datasets due to the low latency to the CPU. It should be remembered that the CPU can perform even better for smaller datasets though, due to the higher clock frequency.

The integrated GPU turned out to be the fastest for larger datasets as well, as long as the arithmetic intensity was rather low. Most of the benchmarks did however use a very limited number of memory accesses compared to the amount of data, which is not the case for all algorithms. This is clarified by the chained kernel benchmark, where the discrete GPU outperforms the integrated GPU for lower arithmetic intensities due to its high bandwidth to GPU memory. It should also be noted how the process priority can have an impact, not only on the integrated GPU, but also on the discrete GPU due to the CPU scheduling. Figure 24 shows the instances where the integrated GPU was the fastest for all benchmarks.

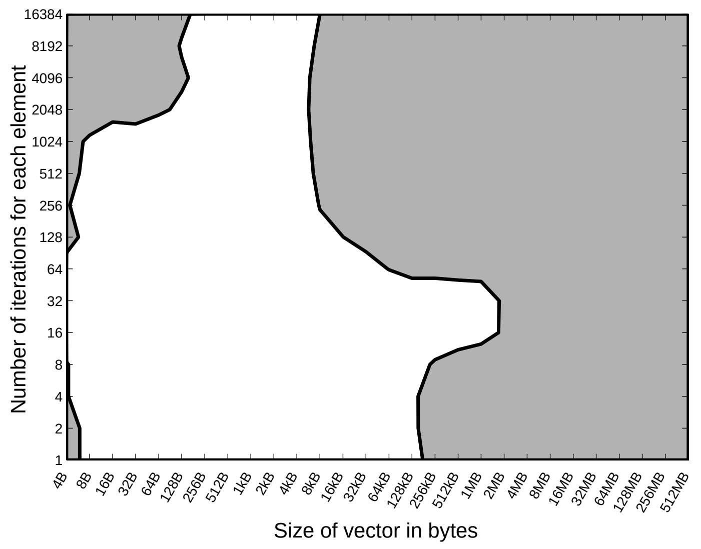


Figure 24: All problem instances where the integrated GPU was the fastest for all benchmarks, denoted by the white area in the graph.

From a pure performance point of view, it is recommended to use integrated GPUs for datasets up to 16 kB, and if the algorithm is of a low arithmetic intensity, for datasets up to 1 MB. The exception is for really arithmetically intense problems on small datasets, as the CPU handles the lack of parallelism better.

This thesis does however only compare performance of integrated and discrete GPUs. While performance usually is one of the most important aspects in HPC, power efficiency is also important in many situations, and due to the integrated GPU using much less power than a discrete GPU, an interesting follow-up could be to investigate integrated GPUs with a focus on performance per Watt, instead of only pure performance.

An interesting, and unplanned, twist in the thesis was the focus on mapped and copied memory. The idea from the beginning was to use mapped memory for the CPU and the integrated GPU, and to use copied memory for the discrete GPU, as that was expected to result in the fastest possible memory accesses for each device during the execution. However, during the vector increment benchmarks, due to latency hiding, mapped memory turned out to perform better for the discrete GPU as well in many cases. This is also highly affected by the kernel, as each element is only accessed twice; once for reading, and once for writing. An interesting follow-up on this could be to investigate when to use mapped memory, and when to copy data, for best performance.

This thesis has exclusively focused on finding out if the integrated or the discrete GPU is the fastest for various kinds of problems, and the results clearly show that different problems are suited to different kinds of devices. OpenCL does however support heterogeneous computing, allowing the integrated and the discrete GPU to be used at the same time by the same program. By using the results from this thesis to pick the most efficient device for each part of a problem, and at the same time running the devices in parallel, it is possible to utilize a system even further, and this could really bring out the potential of GPU computing.

References

[1] NVidia Corporation. What is GPU computing? `http://www.nvidia.com/object/what-is-gpu-computing.html`. Retrieved April 2, 2013.

[2] AMD. *AMD Accelerated Parallel Processing*, December 2012.

[3] Mayank Daga, Ashwin M. Aji, and Wu-chun Feng. On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing. In *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing*, pages 141–149, July 2011.

[4] Matthew Doerksen, Steven Solomon, and Parimala Thulasiraman. Designing APU oriented scientific computing applications in OpenCL. In *Proceedings of 2011 IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 587–592, September 2011.

[5] Nan Zhang, Yun shan Chen, and Jian li Wang. Image parallel processing based on GPU. In *Proceedings of 2010 2nd International Conference on Advanced Computer Control (ICACC)*, pages 367–370, March 2010.

[6] James Fung and Steve Mann. Using graphics devices in reverse: GPU-based image processing and computer vision. In *Proceedings of 2008 IEEE International Conference on Multimedia and Expo*, pages 9–12, April 2008.

[7] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Parallel hybrid evolutionary algorithms on GPU. In *Proceedings of 2010 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, July 2010.

[8] Yanxiang Zhou, Juliane Liepe, Xia Sheng, Michael P. H. Stumpf, and Cgris Barnes. GPU accelerated biochemical network simulation. *Bioinformatics*, 27:874–876, 2011.

[9] Kyle Spafford, Jeremy S. Meredith, Seyong Lee, Dong Li, Philip C. Roth, and Jeffrey S. Vetter. The tradeoffs of fused memory hierarchies in heterogeneous architectures. In *ACM Computing Frontiers (CF)*, pages 103–112, May 2012.

[10] T.R.W. Scogland, H. Lin, and W. Feng. A first look at integrated GPUs for green high-performance computing. *Computer Science - Research and Development*, 25:125–134, 2010.

[11] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU computing. In *Proceedings of the IEEE*, pages 879–899, May 2008.

[12] Kayvon Fatahalian and Mike Houston. A closer look at GPUs. *Communications of the ACM*, 51:50–57, 2008.

[13] Ian Buck. *GPU Gems 2*, chapter 32. Addison-Wesley Professional, 2005.

[14] A. Branover, D. Foley, and M. Steinman. AMD fusion APU: Llano. *Micro, IEEE*, 32:28–37, 2012.

[15] Wu chun Feng and Shucai Xiao. To GPU synchronize or not GPU synchronize? In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3801–3804, May/June 2010.

[16] Shucai Xiao and Wu chun Feng. Inter-block GPU communication via fast barrier synchronization. In *Proceedings of 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12, April 2010.

[17] Phuong Hoai Ha, Philippas Tsigas, and Otto J. Anshus. Wait-free programming for general purpose computations on graphics processors. In *Proceedings of 2008 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12, April 2008.

[18] Aaftab Munshi, Benedict R Gaster, Timothy G Mattson, James Fung, and Dan Ginsburg. *OpenCL programming guide*. Addison-Wesley Professional, 2011.

[19] Guillaume Colin de Verdière. Introduction to GPGPU, a harware and software background. *Comptes Rendus Mecanique*, 339:78–89, 2011.

[20] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38:391 – 407, 2012.

[21] AMD. AMD radeon HD 6970 graphics. `http://www.amd.com/us/products/desktop/graphics/amd-radeon-hd-6000/hd-6970/Pages/amd-radeon-hd-6970-overview.aspx#2`. Retrieved August 8, 2013.

[22] Steven Walton. AMD A10-5800K trinity APU review. `http://www.techspot.com/review/580-amd-a10-5800k/`. Retrieved August 8, 2013.

[23] Chris Angelini. Amd trinity on the desktop: A10, a8, and a6 get benchmarked! `http://www.tomshardware.com/reviews/a10-5800k-a8-5600k-a6-5400k,3224.html`. Retrieved November 17, 2013.

[24] Earl F. Glynn. Fourier analysis and image processing. `http://research.stowers-institute.org/efg/Report/FourierAnalysis.pdf`. Retrieved June 23, 2013.