Low Level GPU Performance Characteristics Using Vendor Independent Benchmarks

Navid Saremi - Utrecht University

Supervisors:

Vincent Hindriksen - StreamHPC
Jacco Bikker - Utrecht University

July 2018

Abstract

In parallel processing, GPUs are one of the most common devices used for computing. Each GPU architecture is different than others and usually has difference performance characteristics under different loads. In order for an application to run at optimal performance when using GPUs as compute devices, it is necessary to know the low level behavior of that device. In this study we first create a framework that is device independent and can be compiled by each vendor's native compiler. We then classify the GPU hardware modules that have the most impact on performance and create a series of benchmarks with different execution patterns and loads in order to create an overview of the GPU's performance characteristics. These characteristics can then be used as a basis for other applications in order to use the results to optimize for a certain device or decide which device provides optimal performance for that application.

The source code will be available at: https://github.com/ncrkblacksmith/low-level-gpu-micro-benchmarks-HIP

Contents

1 Introduction

In recent years, with the advancement of heterogeneous computing using GPUs and the desire for parallel computation in all fields of science, a lack of understanding of the performance characteristics of running an application in a massively parallel environment can be felt when traditional analysis methods are insufficient in determining the run-time performance of algorithms run in parallel environments. For example, two different algorithms, both with a theoretical time complexity of O(n) might have different run-time complexities when they are executed in a parallel processing environment.

In this study, we analyze the low-level characteristics of recent high-end GPUs (Produced by AMD and Nvidia) to determine what hardware resources could have an impact on the performance of a given algorithm, by analyzing the usage of each hardware resource and determining its impact on the execution time of the algorithm. This is useful when a problem has a set of solutions but the run-time performance may vary on different devices.

We focus our study on platform dependent optimizations and to do so we select multiple basic low-level modules within the GPU that can have an impact on performance. A list of such entities are: Global memory data communication, local data communication, atomic instructions performance, registers available and register spilling, hardware scheduler latency, data locality and cache utilization. Multiple characteristics will be tested and their performance results discussed in future chapters. Each hardware characteristic is related to the way a hardware resource is being used.

We propose a series of platform independent tests in order to determine the capacity of each GPU in each one of the hardware resources in order to determine which optimizations are beneficial given the hardware and what the possible bottlenecks are for each resource on each GPU.

Since we want to experiment of different GPU architectures from different vendors, we use the HIP programming language [GHIP17], [BS17] for our experiments. The main feature of HIP is that the source code is portable and it can be compiled using the GPU's native compiler for the target GPU. The other reason for selecting HIP is because there is an ongoing dispute about performance of applications when written in CUDA or OpenCL on each device [KK11], [JF11] which is out of the scope of this research [NVCC18], [GHCC17].

The contribution of this work:

- We propose a classification of the GPU hardware into hardware modules that can be tested in our experiments to find out their performance characteristics under different loads.
- We propose a set of platform independent benchmarks to analyze each one these modules on any given GPU.

- Using the results of our experiments we will select a practical algorithm that makes use of most of the hardware modules and try to improve its performance using the knowledge available through our experiments.

2 Graphics Processing Unit

The first generation of graphics processing units was only used to output images created by the CPU to the display device. They lacked processing power and being programmable for doing anything else. With further development of 2D image processing and the introduction of real-time 3D games, developers began creating APIs for accessing the graphics hardware. The most notable API that we know now was the OpenGL API library [KHR92] which was a hardware independent API for real-time 3D graphics.

The early GPU architectures consisted of a fixed-function pipeline [KHR15] as shown in Figure 2.1. The GPU required a list of vertices and a lighting model in order to color them as desired. The programmer had to work with the hard-coded shading model available the GPU in order to perform graphical operations. The API provided some fixed functions for the programmer to choose from which mapped approximately to the GPU hardware available.

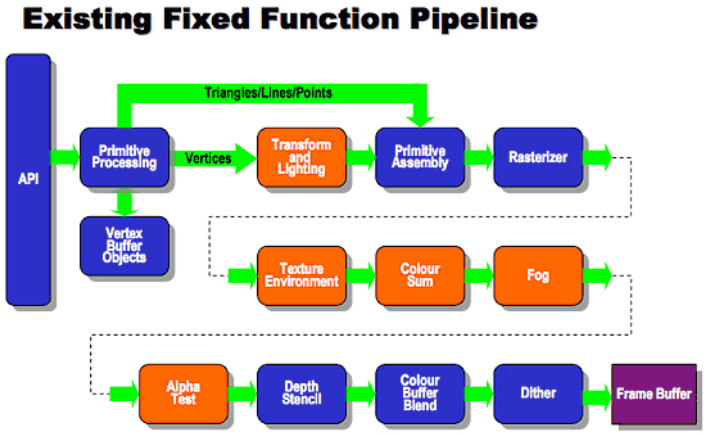


Figure 2.1: OpenGL's fixed function pipeline

With future generations of GPUs and APIs in the early 2000's, Nvidia was the first to introduce a GPU that was capable of running a shader program for every vertex and pixel [NV01], although a very short program(Figure 2.2).

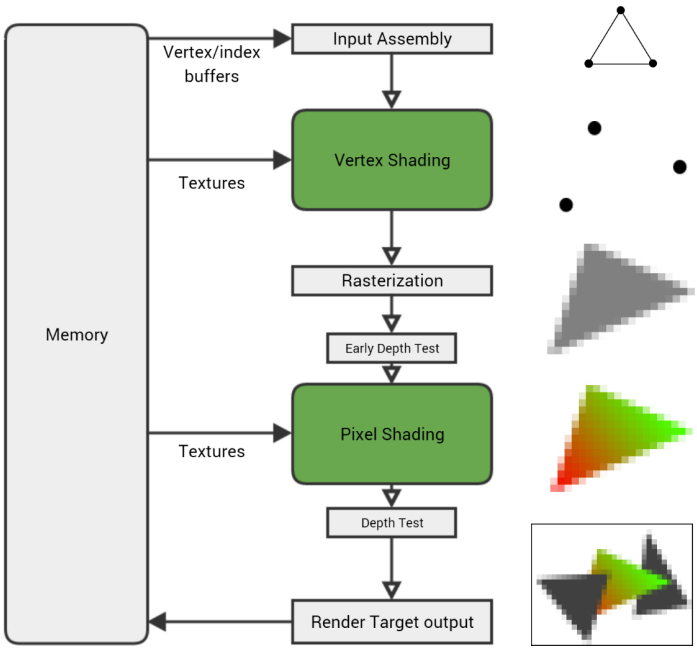


Figure 2.2: Programmable shading pipeline in earlier GPUs [CA17]

This allowed researchers to present their problem using graphic entities such as vertices and pixels to the GPU and make use of the available hardware in order to run their program. At first, researchers and programmers were using already available graphics APIs such as OpenGL [KHR92], but after the introduction of general purpose languages such as OpenCL[KHR09] and CUDA [CU18] which were close to the C and C++, the researchers opted to use those languages. An example comparison of standard C code running in sequential manner and the equivalent parallel CUDA code can be seen in Figure 2.3.

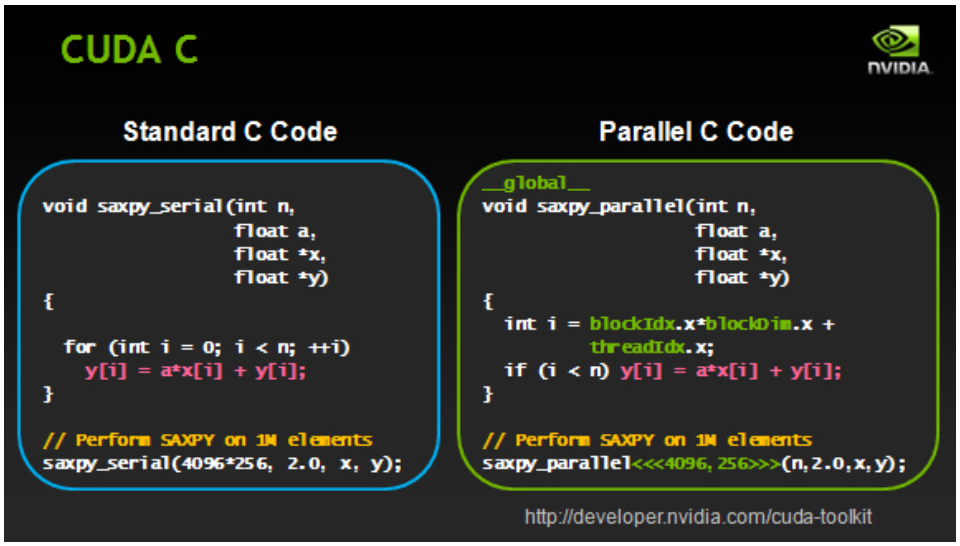


Figure 2.3: Example CUDA code

The GPU's massively parallel design allowed parallel processing paradigms to become more relevant when writing programs on the GPUs. In this context GPUs were also referred to as *streaming processors* [JB15]. The term *stream processing* is used to describe a data-centric computing model in a parallel processing environment in which each processor is working with a stream of data, as shown in Figure 2.4. This model is used in applications that are compute-intense with high data locality and data parallelism. This means that algorithms that could take advantage of GPUs are algorithms that could divide their data into smaller pieces that have little to no data dependency to each other and therefore each part can be executed separately. Each one of these small pieces is called a stream and the GPU is running the same code for each of its processors using a stream of data which is part of the total data.

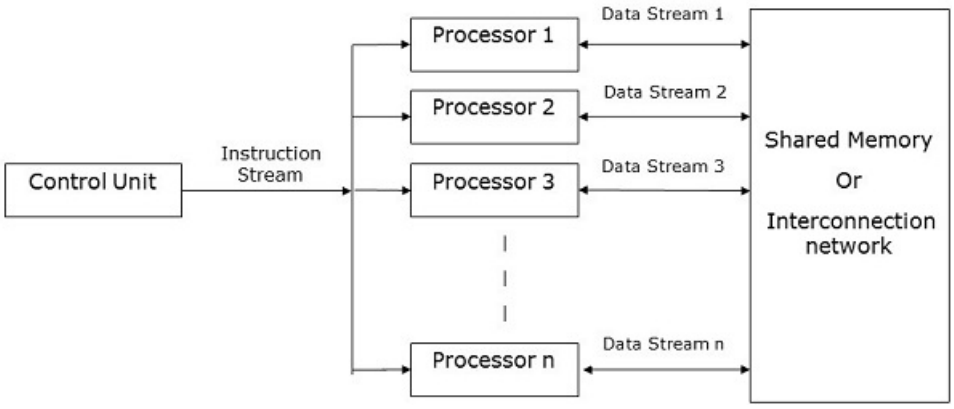


Figure 2.4: Stream processing diagram [CB17]

In order to use the streaming model on the GPUs we will discuss some programming terminology that is often used to describe each part of a streaming model on the GPU. In a parallel environment there are multiple processing units available for processing which are referred to as *steaming processors* . Data is divided into many *streams* and each processor will perform an operation on the sequence of data assigned to it. The more compute resources of the processor used the more *occupied* the device is which could in theory result in more performance.

As shown in Figure 2.5 and Figure 2.6, at a high level the GPU consists of a set of processing cores which are called *streaming multiprocessors* (or shading multiprocessors) or SM for short. Each with its own set of registers and ALU's (Arithmetic Logic Unit) and *shared memory* in order to perform simple operations on the data. The registers and their data are only available to the processor. In addition to the processor's local memory there is also an L2 cache on the device. At the highest level there is *global memory* which is usually off-chip memory modules that have a high latency. This hierarchy is similar to what is used in CPU's. The first generations of GPU's did not have any cache hierarchy, because it was considered not to be beneficial for streaming processors due to not reusing data often.

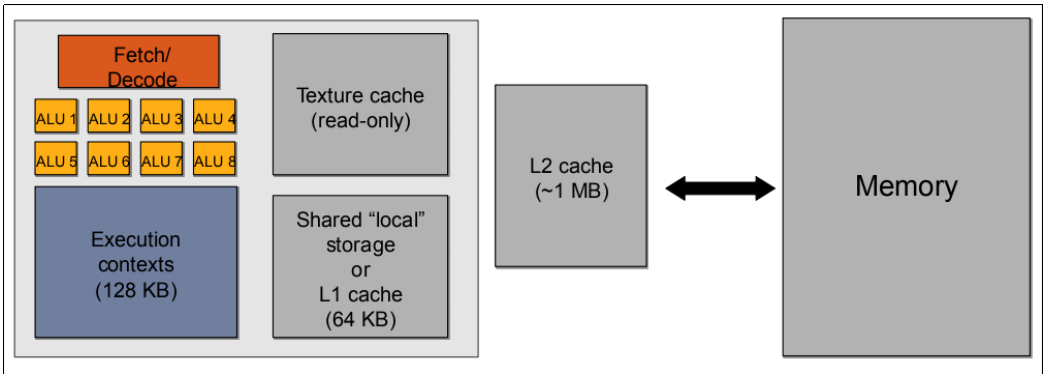


Figure 2.5: GPU architecture for one core [OR11]

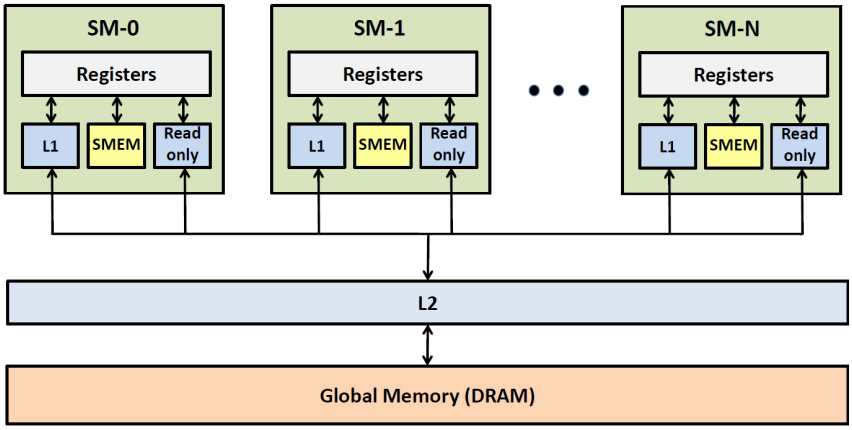


Figure 2.6: GPU architecture [PM12]

Each processor is in charge of running multiple threads together and each thread represents one stream of data. As figure 2.7 shows, a group of threads form a block of threads and a group of blocks form a grid. The grid is basically the entire application that will be run on the GPU. During run-time each processing core is in charge of executing a block. But not all threads within a block are executed simultaneously. The subset of threads that are executed together are referred to as a *warp* . Each block is executed using multiple warps. The number of threads within a warp is variable between different vendors, for example Nvidia uses warp size of 32 threads, while AMD uses warp size of 64 threads. During run-time the processing core has to decide which instructions to execute and in which order. Further parallelism can be achieved by executing multiple instructions in a thread. This would provide more work for the processing core to choose from. For example when one instruction is requesting memory, during the time it takes for the memory controller to fetch that memory the processing core can execute the next instruction. This concept is commonly referred to as *instruction level parallelism* , ILP. ILP is very similar to *out of order execution* , OOE in CPUs.

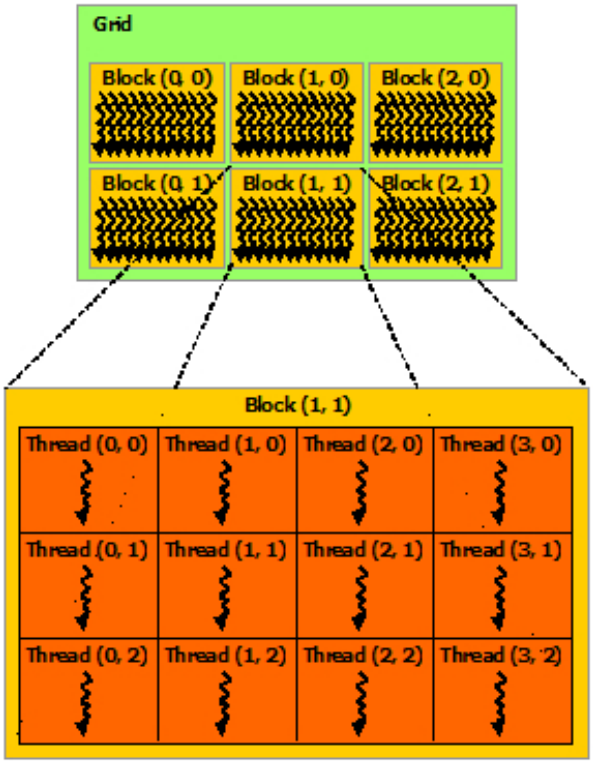


Figure 2.7: Blocks of threads [NV18]

3 GPU Programming Frameworks

Any language that allows the code running on the CPU to poll a GPU shader for return values, can create a GPGPU framework." [MH05]. In recent years, several languages have been developed that provide a framework for programming on the GPU such as: OpenCL, CUDA, C++ AMP, DirectCompute, etc.. It is important to describe two terminologies used throughout this chapter. We divide the system into two parts which will be called *host* and *compute device* . The host consists of a system using a regular CPU and is not necessarily running in a parallel environment. In this system the compute device is the GPU.

In parallel processing using a compute device, the host program and the GPU device program are separate entities. The host process simply prepares the computing device for execution on the data the host has transferred to the compute device for its task.

One of the popular frameworks is OpenCL [KHR09] which is a vendor-independent language, meaning that it can support a wide range of devices. This means that OpenCL can run on a variety of hardware such as CPUs, GPUs and FPGAs. Covering a lot of processors means that there is a high level of abstraction in the language that will limit access to low level hardware instructions.

Nvidia launched CUDA in 2006. CUDA is a framework for using Nvidia's own GPUs in general purpose applications. The language used in CUDA was C and later on C++. CUDA is mostly used when targeting Nvidia GPUs since it provides better optimization and lower level access to the Nvidia hardware compared to OpenCL. CUDA is the most popular framework in systems that are using Nvidia GPUs and since its launch there has been a lot of libraries implemented specifically for CUDA, which has attracted a lot of programmers.

To study the different aspects of GPUs in relation to algorithms, there first thing to consider is a framework that gives access to the GPU device in order to execute parallel application. Since we want to generalize our research to all hardware we need to select a framework that is platform independent but also provides low level access and optimization to the GPU hardware. Some frameworks would satisfy the first requirement but they are not specialized enough to take advantage of the low-level hardware of the GPUs. We thus need a framework in which the source code is platform independent and can provide low level access to the hardware. This means that the framework acts as a preprocessor and provides the necessary files for each hardware's native compiler in order to create executable files that can be run on the host device and execute the parallel application on the compute device.

We chose the HIP language for this study. The HIP programming language is very similar to CUDA, with a similar syntax and program structure. HIP was designed to help converting CUDA code (which only runs on Nvidia devices) into portable C++ code that can be compiled using AMD native compilers. HIP's source code is platform independent and can be compiled on each platform using that platform's

native compiler (NVCC on Nvidia, HCC on AMD) which gives the freedom we need in order to inspect the behavior of algorithms on each platform.

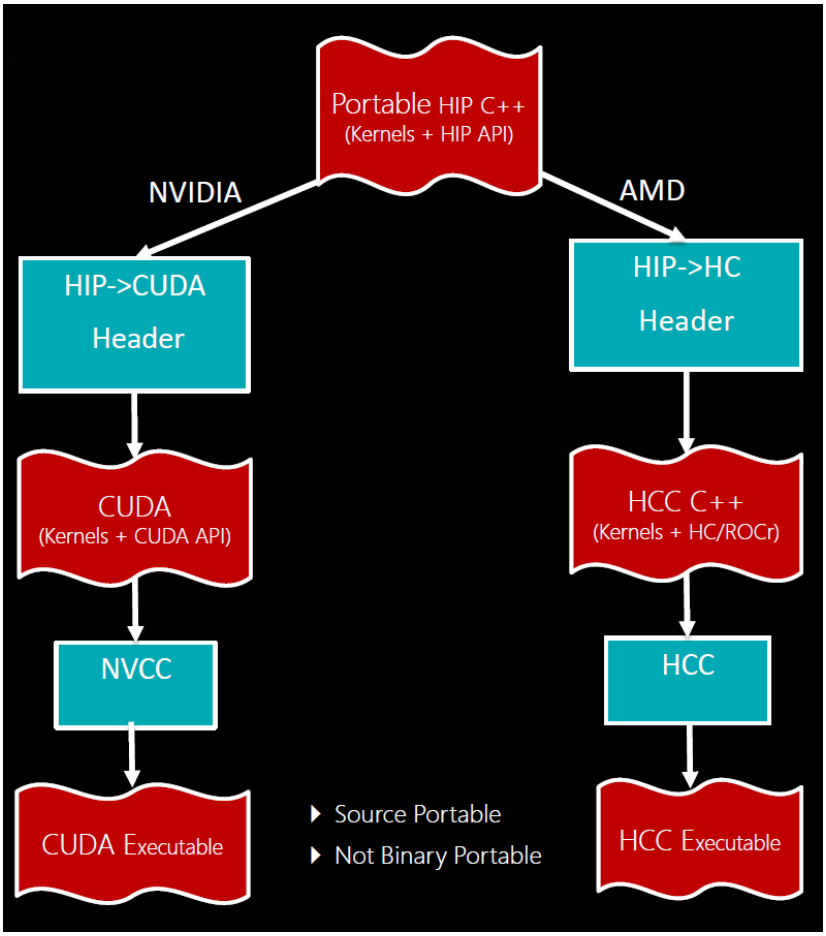


Figure 3.1: HIP compilation hierarchy [BSR17]

4 Related Work

Traditionally memory latency has always been a bottleneck for algorithms running on a streaming processor. In the work done by Wen-mei Hwu [WH14], memory bandwidth and load balance are two of the main challenges. Xinxin Mei and Xiaowen Chu [XM16] also performed research on the memory hierarchy of GPUs using microbenchmarks in order to extract memory characteristics of a GPU device on different levels. They observe that the difference in latency when accessing a higher-level memory (such as global memory) compared with lower level memories (such as shared memory or cache). This shows that data locality on lower level memories could in theory increase the performance of an application by reducing the latency of accessing a data element from a lower level memory.

Recent GPU architectures make use of shared memory within each streaming processor. Xinxin Mei and Xiaowen Chu [XM16] and Koki Hamaya and Satoshi Yamane [KH17] focus on the shared memory throughput in their research. The shared memory in each processor is divided into multiple banks, where each bank can supply 4-bytes or 8-bytes. In theory, if memory requests from threads are mapped to one bank, the requests are processed sequentially. This is in contrast to a situation where each thread requests data from a different bank which can then be done in parallel. As discussed in their study *bank conflicts* within active threads in a processor could reduce the throughput of shared memory by a significant margin, preventing some GPUs from even reaching 20% of their theoretical shared memory throughput. It is thus essential to analyze memory access patterns on shared memory in order to find and prevent possible bottlenecks.

In the work previously done in analyzing GPU behavior [SH09], we can find a detailed description on how memory latency is hidden. When each multiprocessor is executing a warp it may request a memory read/write instruction and when encountering a memory operation by a warp, the multiprocessor quickly swaps the warp for another warp and queues up the memory operations requested by the previous warp. While executing a new warp the memory operations of the previous warp will be done in parallel and therefore when a warp is being stalled waiting for memory the other warp starts executing and the execution of the next warp hides the latency of previous warp by executing the new warp while waiting for the requests of previous warp to be fulfilled.

In recent years GPU manufactures have opted to use HBM memories [JK14], [MOC14] which use multichip-modules [WIKI13] and are closer to the processor die than the traditional off-chip memories. The work done by Maohua Zhu et al. [MZ17] shows that in order to optimize different applications for GPUs (for example AlexNet [AK12] on Caffe [YJIA14] and Breadth-First Search) that use HBM memory, there are certain considerations for fully exploiting the HBM memory's performance. They proposed a three-stage software pipeline in order to control memory transactions for the algorithms in order to reduce the number of memory transactions between the compute device and the host device that occur when OpenCL allocates memory buffers in host and create a bottleneck because of using the PCI bus for data transfers

which has lower speed.

Several studies have analyzed and benchmarked the GPU's memory operations in order to find out characteristics of newer generations of GPU hardware and expose possible limitations when it comes to memory operations. Xinxin Mei and Xiaowen Chu [XM16] show that cache utilization on the GPU is different than that of the traditional cache utilization on the CPU,. The main reason is size of the cache in GPUs is much smaller compared to CPUs. The size of cache on the GPUs might vary in run-time based on the size of shared memory an application might use. The memory could be temporarily used when there is a high number of local variables required by a single process. Later hardware generations have opted to increase shared memory size and the number of simultaneous shared memory locations that can be accessed. Dae-Hwan Kim's research [DHK17] shows that if memory access is not coalesced, that is if the access pattern, requests memory that is not being currently worked on, or it does not utilize data locality for all threads (each thread requesting a different part of memory) there is on average a 57 percent decrease in throughput on a pascal GTX TITAN X.

Blem et al. [EB11] has used a set of benchmarks in order to find bottlenecks and predict performance of an algorithm that is run on a SIMT device. In their study they classify the bottlenecks into three categories: parallelism, control flow, and memory limitations. A GPU simulator is then used for each algorithm in order to determine which resources is the algorithm using and the occupancy level of each one of the resources. After finding resource limitations the simulator then speculates the potential speedup if that resource was to be fully utilized. It was noted that on average 19x more speed achievable based on raw hardware characteristics, but that there is no definitive way of determining how the algorithm should be altered in order to achieve the peak performance.

While the general conception is that higher occupancy would result in higher throughput because of more work being done in every cycle, the study done by Volkov. [VV10] showed that this is not always the case. In the research, there are three categories that are being discussed and tested demonstrate where would lower occupancy result in better performance: Hide arithmetic latency using fewer threads, hide memory latency using fewer threads, run faster by using fewer threads. Volkov [VV10] argues that increasing instruction level parallelism (ILP) is another way of hiding latency and shows that having more instructions per thread can optimize the execution pipeline usage and not halting execution when reaching an instruction that has a higher latency.

The study also shows that reducing the number of threads and increasing the number of registers used per thread would increase performance. This is due to the fact that registers are much faster than shared memory in any device (up to 3x) and therefore having more local data in registers (and hence less threads) would increase data locality at a register level for the application.

Following up on the work done by Volkov [VV10] and Blem et al. [EB11], Lal et al. [SL14] studied parallel kernels that have low occupancy but are constrained by

other factors: limited by number of blocks, limited by registers and limited by shared memory. The research considers multiple factors when analyzing GPU resources:

- Instructions per cycle,
- Ratio of arithmetic instructions to total instructions
- Ratio of branch instructions to total instructions
- Ratio of memory instructions to total instructions
- Ratio of bandwidth utilized and bandwidth available
- Ratio of global memory instructions and global memory transactions
- Average utilization of SM core for issued cycles
- The fraction of total cycles where pipeline is stalled and could not issue instructions
- Number of active warps per SM

The study shows that when kernels are not limited by Instruction Level Parallelism and are not running at full occupancy, increasing the occupancy further improves the throughput of the kernel. It also denotes that kernels that run at full occupancy may have low performance due to high bandwidth usage, low coalesced memory access patterns or low utilization of SM cores.

Just focusing on hardware capabilities might not be enough for analyzing run-time performance of a given algorithm. Thus a more detailed view of different algorithms might be needed. Majumdar et al. [AM15] studies the behavior of multiple kernels when more processing units are available in newer generations of hardware. The study shows that in general, kernels are classified into three different categories:

- Compute-Bound Kernels: Kernels that have low amount of memory transactions and benefited from more compute processors available to them. (38% of kernels tested)
- Memory-Bound Kernels: These kernels are mostly bound by memory bandwidth and not only do they not benefit from more processors but also increasing the number of processors might result in conflicts at the L2 cache which in turn actually slows down the kernels considerably. (30% of kernels tested)
- Balanced Kernels: these kernels have a compute to bandwidth ratio that allows them to reach maximum performance. (16% of kernels tested)
- Kernels That Do Not Scale: These kernels suffer from algorithmic limitation that does not allow them to scale when more bandwidth or compute processor is available to them. (15% of kernels tested)

Based on the related research, we will focus our study on GPU memory hierarchy and available resources related to it. We will create a benchmark and define experiments that stress parts of the GPU that are being used in almost every GPU application. We will verify the results of some previous works whilst adding more information about the behaviour of the device. Different execution configurations will be experimented on to provide more insight about each module.

5 The Framework

The framework acts as a middle-ware between the host and the GPU and initializes the GPU, makes the GPU execute the program, measures the time until completion of the kernel and reports back the results to the user.

To measure the performance results of each benchmark there are some considerations for normalizing the results in different situations:

- Select the size of data that the benchmark has to work on base on the type and use this size for all others tests (even when using different data-types). For example

  ```
  megabytes<uint32_t>(128)
  megabytes<uint64_t>(128)
  ```

  which both result in exactly 128 megabytes of data which are being accessed by different types.

- Run the benchmark for n iterations and measure the average time between kernel launch and kernel finish in all iterations(default is n=100).

- Report the performance of that benchmark in the amount of data processed per execution time(default is gigabytes/second).

- The data can then be presented in various forms using external applications.

The data is separated into to arrays, one for input and one for output. This separation is done in tests that require the data not to be cached in order to measure the memory modules performance in fetching them from the global memory. The benchmark also needs to know which general operation it is going to perform. Other benchmarks can be made using multiple operations together if the user desires. This is achieved using template specializations for each operation. An example of an operation is provided in Figure 5.1.

```
// kernel_operation base struct
template<
    class T,
    unsigned int BlockSize,
    unsigned int ItemsPerThread,
    kernel_operation_mode KernelJob
>
struct kernel_operation;

// Specialization for direct load/store
template<
    class T,
    unsigned int BlockSize,
    unsigned int ItemsPerThread
>
struct kernel_operation<T, BlockSize, ItemsPerThread, memory_load_store_direct>
{
    __device__ inline
    void operator()(T* input, T* output)
    {
        T items[ItemsPerThread];
        load_direct<T, ItemsPerThread, BlockSize>(items, input);
        store_direct<T, ItemsPerThread, BlockSize>(items, output);
    }
};

// base kernel
template<
    class T,
    unsigned int BlockSize,
    unsigned int ItemsPerThread,
    class Operation = typename kernel_operation
        <T, BlockSize, ItemsPerThread, no_operation>::value_type
>
__global__
void kernel(T* input, T* output, Operation operation)
{
    operation(input, output);
}
```

Figure 5.1: Example kernel specialization for an operation. Each *kernel_operation* structure is corresponding to one operation. T is the type of data.

The first template argument is the type of the data the benchmark has to work with. This can be anything from fundamental types like int, float, double to custom structures defined for that benchmark. The second template argument will determine the number of threads per each block. The third argument has some flexibility as to what it specifies, for example in memory tests the third argument determines items processed per thread, in shared memory benchmarks it is used for the number of active banks for each kernel and for atomics it will determine which regions of output are available for writing.

Using the template arguments each benchmark can create multiple kernels, each for one configuration and run all kernels in order to benchmark each kernel. It is important to note that we run the benchmarks for a specific size regardless of the type we use as our base type. An example is provided in Figure 5.2.

```
run_benchmark<T, 256,  1,   memory_load_store_direct>(megabytes<T>(128),...);
run_benchmark<T, 512,  2,   memory_load_store_direct>(megabytes<T>(128),...);
run_benchmark<T, 1024, 4,   memory_load_store_direct>(megabytes<T>(128),...);
run_benchmark<T, 32,   8,   memory_load_store_direct>(megabytes<T>(128),...);
...
```

Figure 5.2: Creating a kernel for each configuration. T is the type of data.

Using the templates, an operation can be created in order to perform a benchmark that will stress a certain part of the GPU. For example loading and storing from global memory. The framework runs each kernel n times and measures its run time. After the run is finished, we determine the kernel's performance by considering the amount of bytes it processed and the time it took to process that data. The final result is given in gigabytes per second for each kernel. For an example of data gathered using the framework for a certain benchmark, see Figure 5.3.

```
Device name: GeForce GTX 1080
L2 Cache size: 2097152
Warp size: 32
Shared memory per block: 49152
direct_load_store           <256,1>                 110.696 GB/s
direct_load_store           <256,2>                 110.226 GB/s
direct_load_store           <256,3>                 106.33 GB/s
direct_load_store           <256,4>                 109.859 GB/s
direct_load_store           <256,5>                 80.1591 GB/s
direct_load_store           <256,6>                 71.1674 GB/s
direct_load_store           <256,7>                 62.0664 GB/s
direct_load_store           <256,8>                 68.3808 GB/s
direct_load_store           <256,9>                 56.223 GB/s
direct_load_store           <256,10>                55.7024 GB/s
direct_load_store           <256,11>                54.6475 GB/s
direct_load_store           <256,12>                55.1145 GB/s
direct_load_store           <256,13>                52.5941 GB/s
direct_load_store           <256,14>                51.4011 GB/s
direct_load_store           <256,15>                50.3878 GB/s
direct_load_store           <256,16>                57.5653 GB/s
```

Figure 5.3: Framework output example for a device

In our benchmarks we use 32-bit, 64-bit and 128-bit data-types to benchmark the device and show each type's results in a separate chart.

6 Experiments

6.1 Cacheline Request Utilization

As discussed in Chapter 4, The most time consuming operations in a GPU are memory operations. In GPUs the memory controller accesses the data in groups of bytes called cachelines. This is similar to the way CPUs access memory. When a request is made for a memory location, the memory controller has to find which cacheline includes that location, fetch the cacheline and pass it to the streaming processor. After that the data will be available in the streaming processors local L1 cache.

When the data is available on the local cache, they have to be written into the streaming processor's registers. Each active warp can create a number of memory requests during its execution. However the number of cacheline requests at each cycle can influence the performance of the application. In this chapter we will analyze the performance of a device when performing memory operations with regards to the number of cachelines requested and the access pattern.

In parallel applications, dividing the data between processing entities can be done in different ways. Each distribution of data between processing entities has its own performance characteristics. The distribution of data may be restricted to a certain way defined by the application or it can be done freely as long as all the input data is being processed.

In previous work done by Dae-Hwan Kim [DHK17], the coalesced memory access pattern proved to be the most optimal for GPU applications. When threads within a warp are requesting sequential memory locations the access is coalesced. In order for successive instructions within a thread to have coalesced access we need an offset for each instruction within a thread so that parallel instructions within a warp all require sequential data. We call this pattern a *striped* pattern because of accessing the memory in striped partitions as seen in Figure 6.1.

Another pattern for accessing memory is when each thread has to process sequential part of memory. This is in contrast to the striped pattern where sequential memory locations belong to sequential threads. In this scenario the number of cachelines requested at each warp depends on the number of items assigned to each thread, as shown in Figure 6.1.

The other consideration is whether the device supports certain data type that the applications intends to use, such as vectors or half precision floats. Using vector data types might have some benefits which will be later discussed in this chapter.

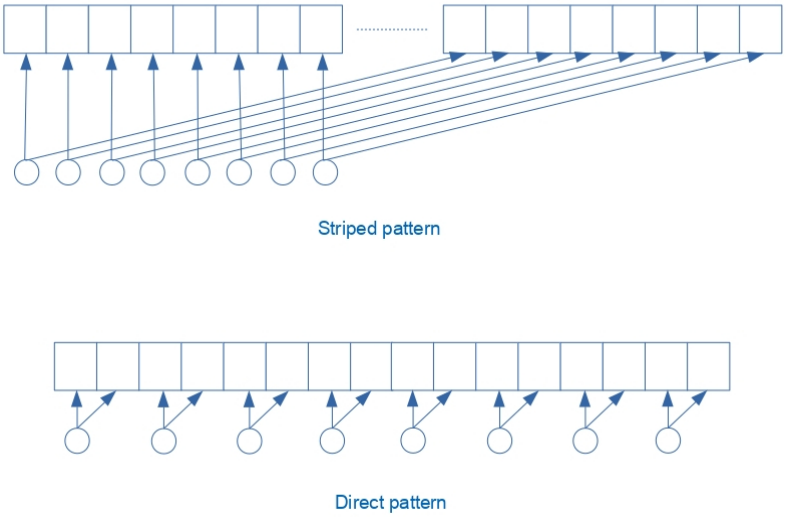


Figure 6.1: Memory access patterns

In this section we will would like to know the memory controller characteristics when performing memory operations in different patterns. There are two main memory operations: read and write. The two operations vary in execution latency when performed in different patterns. Therefore in our benchmarks we use a set of configurations that mixes the memory operations and access patterns. We will use the discussed patterns for benchmarking the memory operations done within a thread, number of cacheline requests and the latency of transfering data between L1 cache and registers within a streaming processor.

To make sure that the requests are actually accessing the global memory, we choose a data size that is larger than the L2 cache. We do the analysis for warps of n threads and how they behave under different loads. For this purpose we select a fixed amount of threads per block that is a multiple of the number of threads per warp in the device. We choose 256 threads(8 warps) per block.

6.1.1 Striped read and write vs Direct read and write

In the first set of benchmarks we use the each access pattern for both read and write. After running the benchmarks, we analyze their performance with each configuration and their performance on different devices. Figure 6.2, 6.3 shows the results of the benchmark.

By comparing the results of the benchmarks, we see that the striped pattern has a stable performance for the most part. The performance drop is due to register spilling which will be discussed in the coming sections. The direct pattern on the other hand starts to drop in performance after a certain items per thread for each device. To

investigate the drop in performance we need to see what is different between these two patterns.

One of the main differences between the access patterns is the number of cachelines requested per instruction within a warp. In the striped pattern each warp creates a constant number of cacheline requests per instruction regardless of the number of items it is processing. The number of cachelines is equal to *warp_size * sizeof(data_type) / cacheline_size* . In the direct pattern the number of cachelines requested per instruction is equal to *warp_size * sizeof(data_type) * items_per_thread / cacheline_size* . Therefore the number of cachelines requested increases by the number of items per thread. In both AMD and Nvidia the cachelines are 128-bytes when the data is not cached in L2.

When using the direct pattern, a thread is accessing 4 sequential integers, the memory controller will create a cacheline request for the first instruction. The rest of the instructions do not require to fetch a cacheline because the cacheline that holds the data has already been requested by the first instruction. The memory controller is not doing any work after the first instruction and the instructions after the first one will only have the latency of transferring data from the L1 cache to the registers. In contrast in the striped pattern one or more cachelines are requested at each instruction. Therefore the memory controller is busy at each instruction unlike the direct pattern.

Considering the result from the GTX 1080, it can be observed that the performance drops if the items per thread is higher than 4. We can calculate the cacheline requests in both patterns for 4 items per thread. In the striped pattern in both read/write operations, at each instruction 1 cacheline is requested and *warp_size* read/write operations are performed on that cacheline. This process is repeated for each instruction. This suggests that the memory controller is capable of fetching 4 cachelines per cycle.

The AMD MI25 on the other hand has different characteristics. It is important to note that the *warp_size* is 64 in AMD devices. The drop in performance is happening gradually from 4 items per thread onward similar to the GTX1080. In both patterns each warp creates requests for 8 cachelines. In the striped pattern 2 cachelines are requested at each instruction and *warp_size* read/write operations are performed on each cacheline.

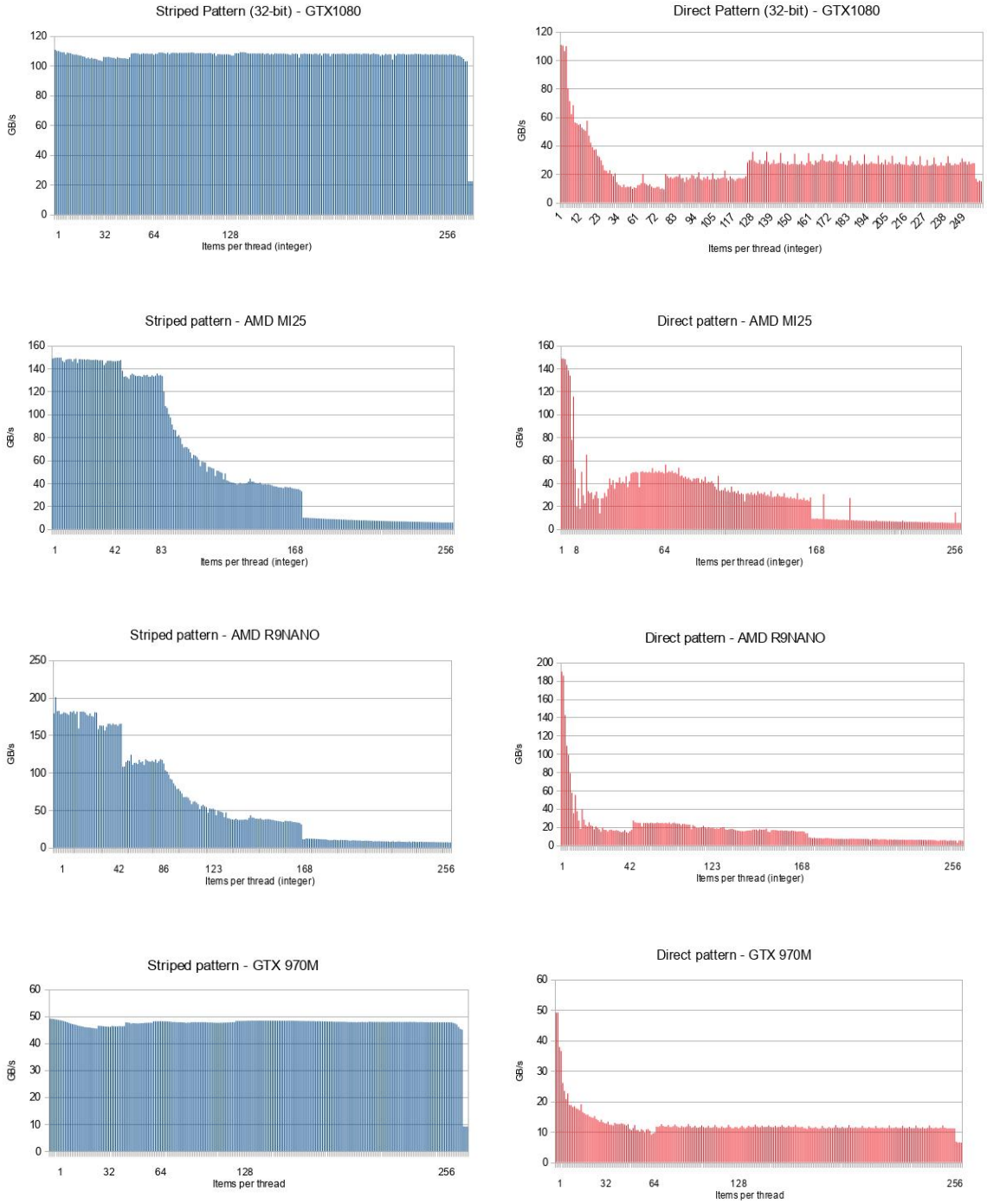


Figure 6.2: Memory access pattern results with different configurations. Each row belongs to one device and each column is for one pattern. The x-axis is items per thread which increases at each iteration. The y-axis shows the performance in gigabytes per second. The color blue is used for striped pattern and red is used for the direct pattern.

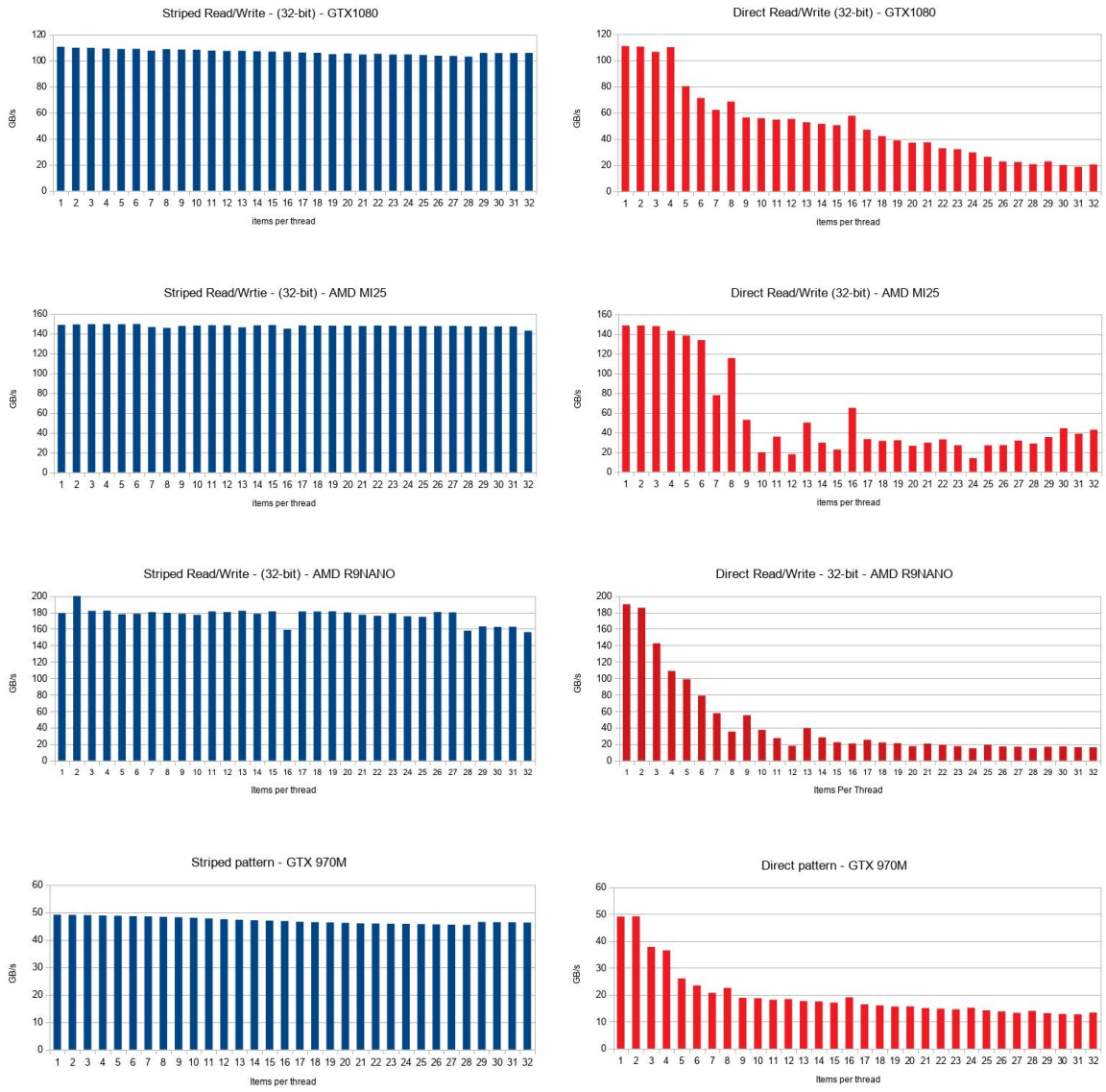


Figure 6.3: Memory access pattern results with different configurations - zooming in for better showing at which configuration the performance starts dropping

To further investigate we need to separate read and write and benchmark them separately. Using the first results we can use the striped pattern for read or write as the pattern that does not create any bottlenecks and test the other one. Later on we will use vector instructions to see their effects on memory transaction performance in GPUs.

6.1.2 Striped Read vs Direct Read and Striped Write vs Direct Write

In this section we show the difference between read and write capabilities of the device. We use two configurations for these benchmarks: 1. Striped read - direct write 2. Direct read - striped write. The reason for this classification is that since the striped pattern will not create any bottleneck, each configuration would only expose the operation that is being done using the direct pattern.

Looking at the results of the striped read - direct write(Figure 6.4 and 6.5) for all devices, we can observe that the capacity for read operations per cacheline is more than the capacity of write operations per cacheline. From the results we can retrieve the maximum number of read or write operations per cacheline on each device. This value can then be used to change the execution pattern of an application on a device based on the memory requests it makes.

Considering the GTX 1080, when using direct write and 4 32-bit integers per thread, 4 cachelines can be fetched per 4 instructions and 8 write requests can be executed per cacheline in each instruction. This is in contrast to the striped pattern where 1 cacheline can be fetched per instruction and 32 write operations performed per cacheline. When items per thread starts increasing to more than 4 the performance drops significantly.

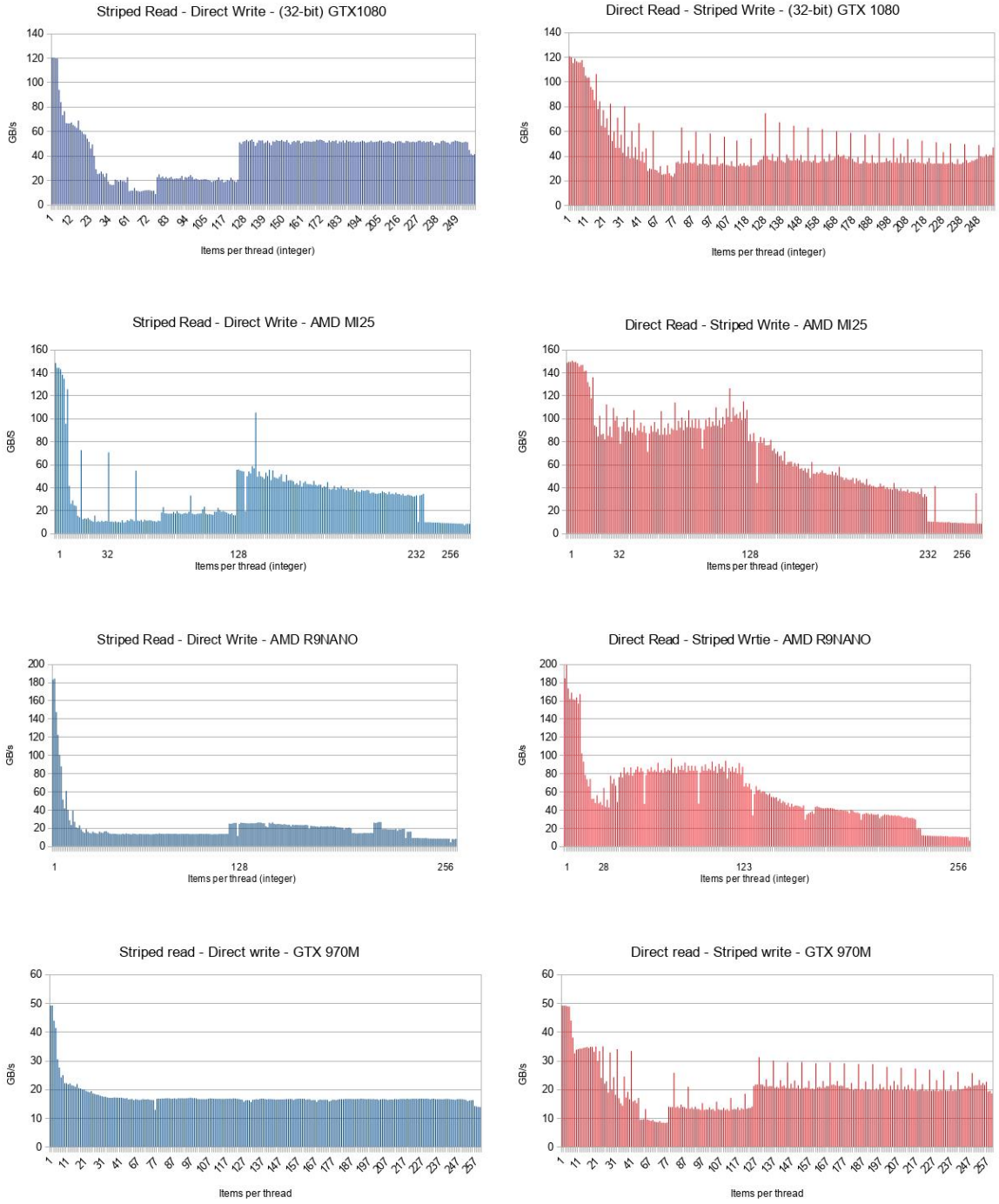


Figure 6.4: Memory access pattern results with different configurations

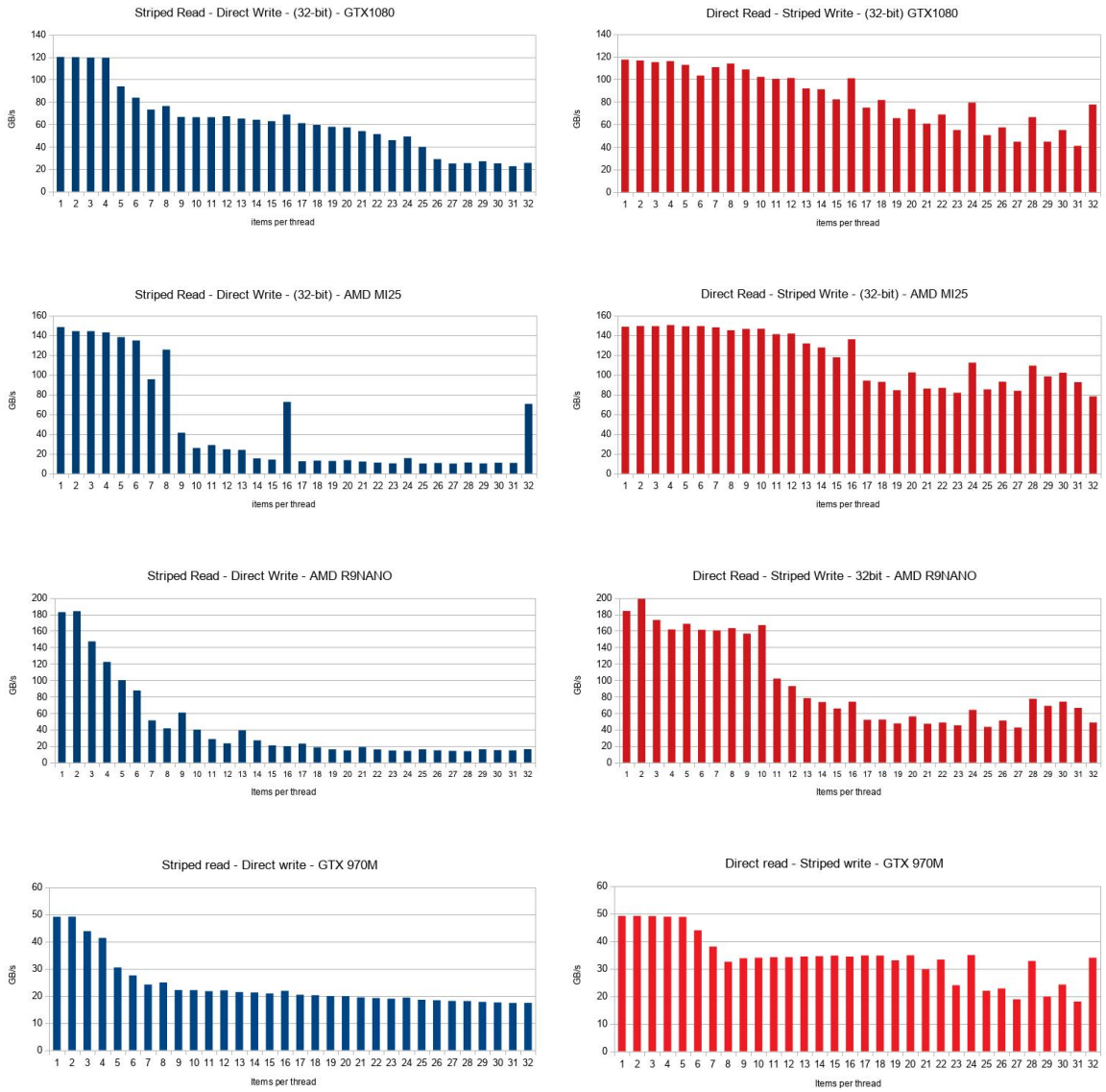


Figure 6.5: Memory access pattern results with different configurations - only showing up to 32 items per thread as this part is where the drop in performance happens

6.1.3 Using Vector Instructions and L1 Transfer Penalty

We investigated the effects of accessing global memory and its latency on the performance of the application. However as mentioned in the beginning of this chapter using vector data-types could have different performance characteristics while operating on the same amount of data.

In this section we start using vectors data types to investigate their effect on performance. After running the benchmarks we will compare the results with previous sections. We would also compare the low level assembly of this sections benchmarks with previous sections. This is useful when the same items per threads has different performance characteristics when using vector data types.

In previous benchmarks all instructions were using 32-bit data types. They gave us general information about the performance of any device when using 32-bit types for basic memory operations. We would like to investigate whether the direct pattern's performance drop is due to memory controller capacity or because of the latency of reading and writing from L1 cache. As it was mentioned before, when using the direct pattern the first instruction creates a request for the cacheline but the rest of the instructions will request data that has already been cached. But when using the vector data type the transaction is done in less instructions. We ran the benchmarks for both 64-bit and 128-bit vector types.

We run the benchmarks for 64-bit and 128-bit vector types. The results are available in Figure 6.6, 6.7, 6.8. Later in this section we will analyze the assembly code for the 128-bit vector type vs the 32-bit data type. We have omitted the results for when both read and write are using the striped pattern because using vector data-types does not influence performance.

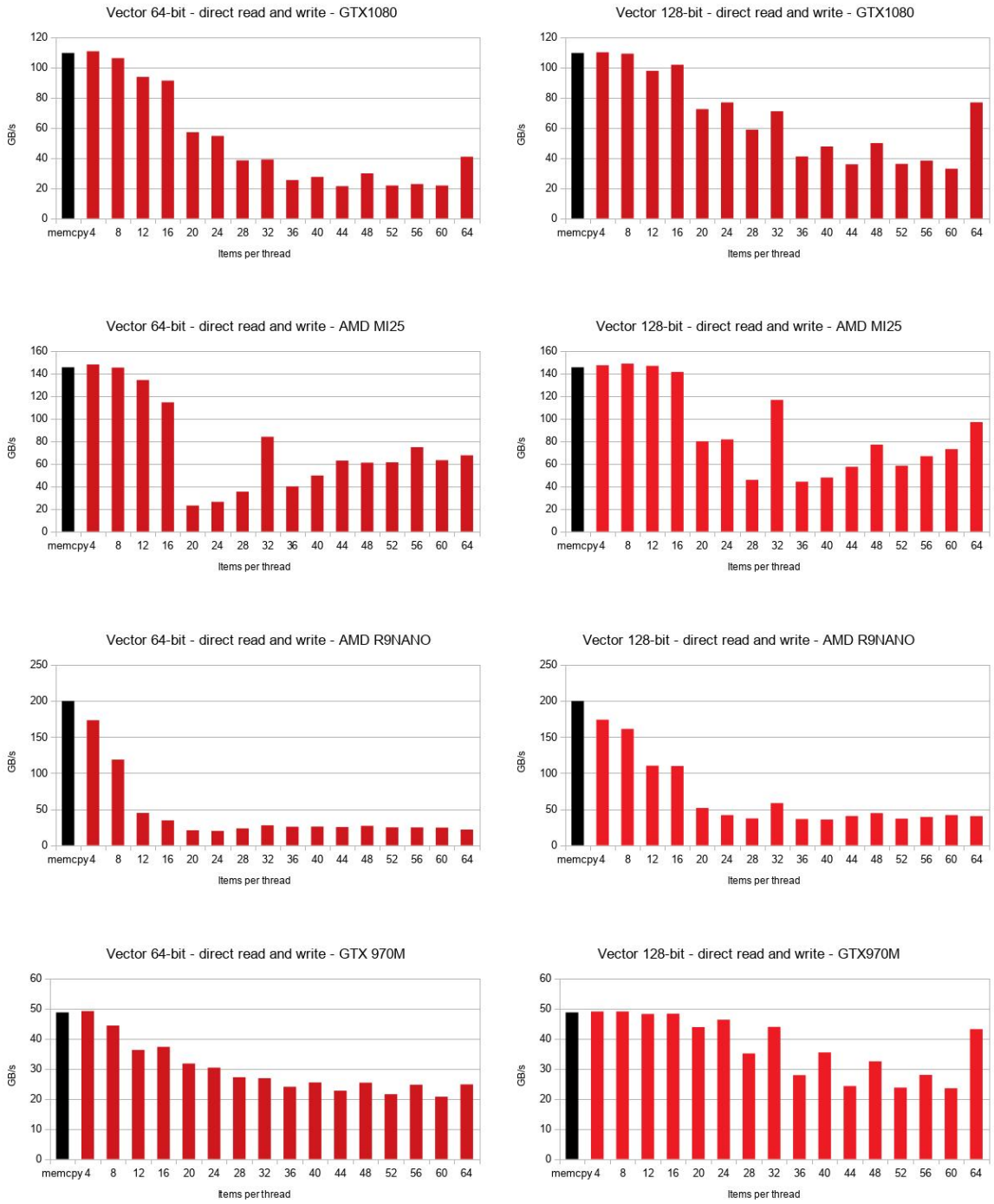


Figure 6.6: Direct pattern comparison between 64-bit and 128-bit vectors. 64-bit and 128-bit vector types side by side for each device. On the left is the results for 64-bit and on the right for 128-bit data type. Each row is representing one device.

Figure 6.7: Direct read striped write comparison between 64-bit and 128-bit vectors. 64-bit and 128-bit vector types side by side for each device. On the left is the results for 64-bit and on the right for 128-bit data type. Each row is representing one device.

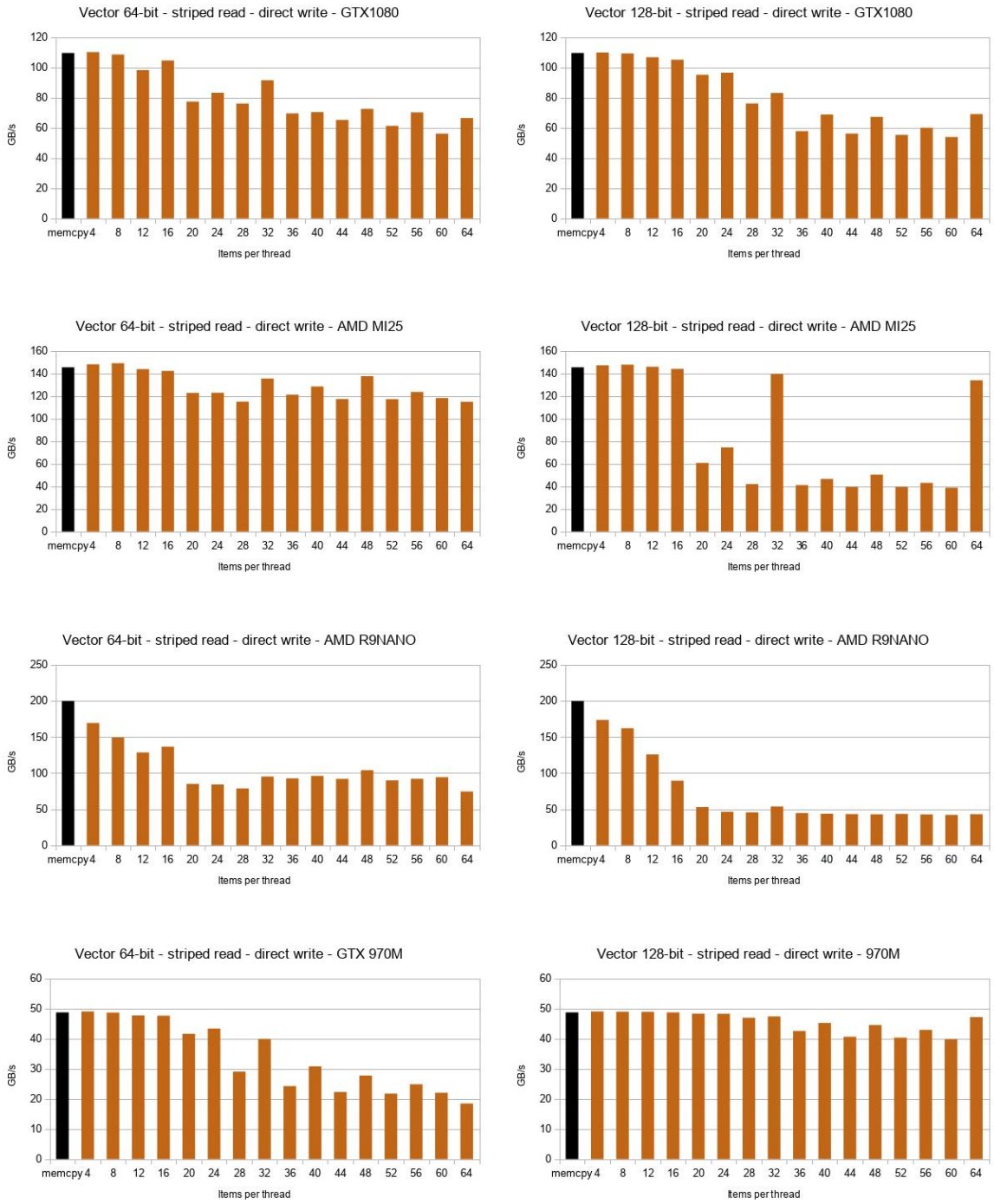


Figure 6.8: Striped read direct write comparison between 64-bit and 128-bit vectors. 64-bit and 128-bit vector types side by side for each device. On the left is the results for 64-bit and on the right for 128-bit data type. Each row is representing one device.

Lookin at the results and comparing them from the results of the previous section, we can see that when using the direct pattern, the vector data type has better stability

in performance when items per thread is increasing. This shows that when using scalar instructions, the application is limited by transferring data to and from registers to the L1 cache. Now that we are not limited by L1 latency per instruction we can determine the number of cachelines that can be requested per instruction in a warp for each device using formulas from the first section.

To show the difference between low level assembly files generated for each benchmark, we use simple block of code in Figure 6.7 as an example for when using vector data-type, extract the assembly code and compare it to the benchmark that used scalar data-type. The assembly code is available in Figures 6.8 and 6.9.

```
vector_count = ItemsPerThread / 4;
block_offset = hipBlockIdx_x * BlockSize * vector_count;
offset = block_offset + hipThreadIdx_x * vector_count;
// for AMD devices a specific vector type had to be used
// so the compiler creates the correct assembly
vector_type items[vector_count];

for(index = 0; index < vector_count; index++)
{
    items[index] = input_vectorized_pointer[offset + index];
}

for(index = 0; index < vector_count; index++)
{
    output_vectorized_pointer[offset + index] = items[index];
}
```

Figure 6.9: Device independent C++ source code - This is the source code that will be used for all devices regardless of compiler in order to find differences between generated assembly code

In the vectorized kernel there is only 1 instruction for loading(Figure 6.8 - *LD.E.128* and Figure 6.9 - *flat_load_dwordx4*) or storing(Figure 6.8 - *ST.E.128* and Figure 6.9 - *flat_store_dwordx4*) the data to and from registers. When execution reaches the load or store instruction in both benchmarks, the first step is to create a request for fetching the cacheline. After the cacheline has been fetched it has to be transferred to the register. In the kernel using the vector data-type, all the four 32-bit elements of the vector are transferred into a register immediately, but in the scalar kernel only one 32-bit data-type will be transferred into a register after fetching the cacheline. Therefore in the scalar kernel the process of fetching data from L1 cache is repeated three times. This creates an overhead of transferring data from and to L1 cache.

```
code for sm_30

/*                                  VECTORIZED                                  */
    MOV R1, c[0x0][0x44];
    S2R R0, SR_CTAID.X;
    PRMT R0, RZ, 0x6540, R0;
    S2R R3, SR_TID.X;
    LOP32I.AND R0, R0, 0x3fffff00;
    IADD R0, R0, R3;
    MOV32I R3, 0x10;

    ISCADD R4.CC, R0, c[0x0][0x140], 0x4;
    IMAD.U32.U32.HI.X R5, R0, R3, c[0x0][0x144];
    LD.E.128 R4, [R4];                          /* load operation */
    ISCADD R2.CC, R0, c[0x0][0x148], 0x4;
    IMAD.U32.U32.HI.X R3, R0, R3, c[0x0][0x14c];
    ST.E.128 [R2], R4;                          /* store operation */
    EXIT;
    BRA 0x80;
    ................................................................................

/*                                  SCALAR                                      */
    MOV R1, c[0x0][0x44];
    S2R R0, SR_CTAID.X;
    MOV32I R5, 0x4;
    S2R R3, SR_TID.X;
    IMAD R0, R0, 0x100, R3;
    SHL R0, R0, 0x2;
    ISCADD R2.CC, R0, c[0x0][0x140], 0x2;

    IMAD.U32.U32.HI.X R3, R0, R5, c[0x0][0x144];
    LD.E R7, [R2];                              /* load operation */
    ISCADD R4.CC, R0, c[0x0][0x148], 0x2;
    LD.E R9, [R2+0xc];                          /* load operation */
    LD.E R10, [R2+0x8];                         /* load operation */
    IMAD.U32.U32.HI.X R5, R0, R5, c[0x0][0x14c];
    LD.E R8, [R2+0x4];                          /* load operation */

    ST.E [R4], R7;                              /* store operation */
    ST.E [R4+0xc], R9;                          /* store operation */
    ST.E [R4+0x8], R10;                         /* store operation */
    ST.E [R4+0x4], R8;                          /* store operation */
    EXIT;
    BRA 0xb0;
    ................................................................................
```

Figure 6.10: Kernel assembly code CUDA - GTX1080

```
/*                                  VECTORIZED                                  */
    s_load_dwordx2 s[0:1], s[4:5], 0x0
    s_load_dwordx2 s[2:3], s[4:5], 0x8
    s_lshl_b32 s4, s6, 8
    v_or_b32_e32 v0, s4, v0
    v_mov_b32_e32 v1, 0
    v_lshlrev_b64 v[4:5], 4, v[0:1]
    s_waitcnt lgkmcnt(0)
    v_mov_b32_e32 v1, s1
    v_add_co_u32_e32 v0, vcc, s0, v4
    v_addc_co_u32_e32 v1, vcc, v1, v5, vcc
    flat_load_dwordx4 v[0:3], v[0:1]            /* load operation */
    v_mov_b32_e32 v6, s3
    v_add_co_u32_e32 v4, vcc, s2, v4
    v_addc_co_u32_e32 v5, vcc, v6, v5, vcc
    s_waitcnt vmcnt(0) lgkmcnt(0)
    flat_store_dwordx4 v[4:5], v[0:3]           /* store operation */
    s_endpgm
/*                                  SCALAR                                      */
    /* initialization code removed due to space*/
    flat_load_dwordx2 v[14:15], v[14:15]        /* load operation */
    s_nop 0
    flat_load_dwordx2 v[12:13], v[12:13]        /* load operation */
    s_nop 0
    flat_load_dwordx2 v[10:11], v[10:11]        /* load operation */
    s_nop 0
    flat_load_dwordx2 v[8:9], v[8:9]            /* load operation */
    v_add_co_u32_e32 v4, vcc, s2, v4
    v_addc_co_u32_e32 v5, vcc, v16, v5, vcc
    v_add_co_u32_e32 v6, vcc, s2, v6
    v_addc_co_u32_e32 v7, vcc, v16, v7, vcc
    v_add_co_u32_e32 v0, vcc, s2, v0
    v_addc_co_u32_e32 v1, vcc, v16, v1, vcc
    s_waitcnt vmcnt(0) lgkmcnt(0)
    flat_store_dwordx2 v[2:3], v[8:9]          /* store operation */
    s_nop 0
    flat_store_dwordx2 v[4:5], v[10:11]        /* store operation */
    s_nop 0
    flat_store_dwordx2 v[6:7], v[12:13]        /* store operation */
    s_nop 0
    flat_store_dwordx2 v[0:1], v[14:15]        /* store operation */
    s_endpgm
```

Figure 6.11: Kernel assembly code HCC - AMD MI25

In our previous sections the first drop in performance was due to having multiple requests which had L1 latency but were accessing sequential data. The performance drop first started because of executing multiple requests which had L1 latency and later on because of cacheline fetching latency of the memory controller. In contrast in the vectorized kernel there is no L1 latency and the drop in performance is solely due to requesting more cacheline requests than what the memory controller could handle in one cycle.

6.1.4 Conclusion

Considering the benchmarks results in this section we can conclude that the difference in performance of different patterns and different items per thread is related to 2 factors:

1. Latency resulting from transferring data from global memory to L2 and L1 cache.

2. Latency of transferring data from the L1 cache to register.

The striped pattern still performs at optimal performance level, whether using vector or scalar instructions. Using the striped pattern we request a cacheline at each instruction and transferring data to and from L1 cache is done while the next instruction's memory request has been made due to ILP. Essentially next cacheline request can be done simultaneously with the transferring of data from L1 cache to registers of the previous instruction.

Finding the characteristics of each device, we can alter an applications execution pattern and order to maximize the throughput of the memory controller for the application. This enables user to prevent bottlenecks by analyzing memory access patterns of the application and adjusting to device capabilities accordingly.

6.2 Register Spilling

The GPU consists of many streaming processors and each one has a certain amount of registers available. The number of registers per streaming processor is varying between different GPU architectures. At run-time the streaming processor assigns the registers to threads for holding local variables. These registers are then occupied until the thread's corresponding block finishes execution. Since the number of registers is limited, when number of registers required by a block of threads exceeds the amount of registers available for a streaming processor, the streaming processor allocates variables in higher level memory. This is called *register spilling* . Register spilling causes heavy performance penalties, because the latency of accessing a register is orders of magnitude lower than accessing higher level memory.

It is necessary to keep track of the amount of registers each block of threads needs. When running the application on a device, knowing the number of registers per streaming processor must be considered when deciding the execution configuration for that application on that device. When this is not taken into account, it is likely that running the application on another device casue register spilling and a drop in performance.

In our previous benchmarks for memory operations, we first loaded data from global memory to registers and then dispatched them to global memory. In this case, the number of items that a thread has to process is exactly the number of local variables it requires. Since we are only interested in the performance impact of register spilling, we choose the results of the striped pattern a basis for analyzing register spilling.

In the results shown in Figure 6.10, the Nvidia devices see a sharp decrease in performance when the block of threads is requesting more registers than there are available in the streaming processor. In contrast, the performance drop in AMD devices happens more slowly and at multiple levels. This can be related to the fact that the AMD devices used for benchmarking in this study, were all using HBM memory whereas the available Nvidia devices did not have HBM memory. This could also be because of the way each vendor handles register spilling, where AMD spills to L1 memory then L2 but Nvidia spills directly to global memory. However we cannot verify whether this is the case because we did not have access to an Nvidia device which has HBM memory.

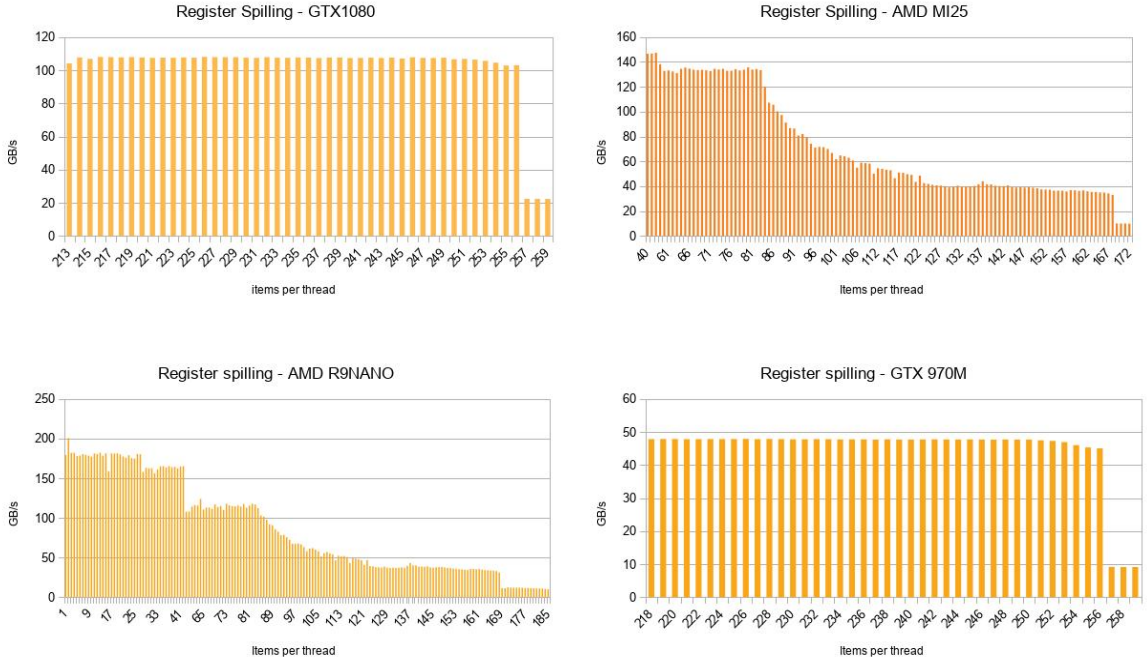


Figure 6.12: Register Spilling on different devices

6.3 Shared Memory

In the GPU memory hierarchy, each streaming processor has an amount of on-chip memory with lower latency than the global memory dedicated to it. This memory is accessible by the block of threads running on that streaming processor. This memory has two purposes, first it is acting as the L1 cache for the streaming processor and second it can be used as shared memory between threads running within a block on that streaming processor, for passing data between each other to limit global memory usage. Distributing this local memory between L1 cache and shared memory is done at run-time when the device is aware of the amount of shared memory requested by each block of threads. Although having threads communicate with each other is against the principle of stream processing, in reality many applications benefit from shared memory and use it.

Shared memory access is organized in n columns of m bytes. The columns are referred to as banks and each bank has multiple rows as shown in Figure 6.11. At any given cycle each bank can provide access to only one of its rows to any thread that requests it.

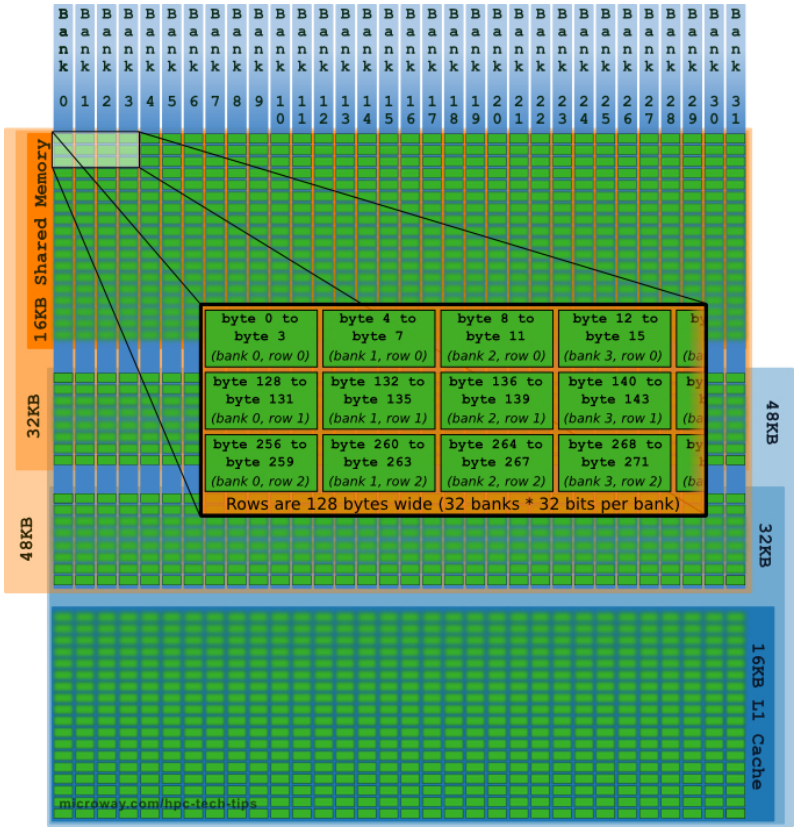


Figure 6.13: Shared memory banks. Microway tech tips[MM13]

If two threads within a warp issue a shared memory request to the same bank, then

the access will be sequential, resulting in a performance penalty. All threads have to wait for the threads with the bank conflict to finish processing before continuing execution.

In this benchmark we will investigate the performance characteristics of each device when different number of bank conflicts occur. Because accessing shared memory has much lower latency than accessing global memory, we run the shared memory operations part of our benchmark in a loop and repeat it multiple times.

In our shared memory benchmark we dynamically control the number of banks that are being accessed by each warp at each iteration. We go from 1 active bank to 32 active banks and test each configuration for read and write. The reason for limiting banks to 32 is that all current GPU devices are using 32 wide banks.

Just like the benchmarks from previous sections, we have separated the read and write results. Judging by the results shown in Figure 6.12, the main performance factor when using shared memory is the number of bank conflicts. It can be observed from Figure 6.12, that the throughput of shared memory is different on each device.

Even though shared memory bank conflicts serialize access and stall the execution depending on the number of conflicts happening, reducing the bank conflicts to 50% can greatly improve performance specially on AMD devices. The performance penalty is proportional to the number of conflicts per bank and not total conflicts. For example, when more than half the banks are active we will at most see 1 conflict per bank which means the access has to be serialized for 2 threads to perform read/write on the bank. It is worth noting that when only part of the threads within a warp are active due to branching instructions managing shared memory access and preventing bank conflicts might be easier. In most applications the exact amount of bank conflicts cannot be calculated but the probability of such conflicts happening can be calculated.

Figure 6.14: Shared memory bank conflict performance

6.4 Atomic Instructions

In parallel applications ran on any multi-core device, two or more threads might want to operate on the same memory location and its previous contents are important to them. This is where atomic instructions are used. Atomic instructions synchronize the access to a memory location by ordering the requests and executing them in order. The cost of synchronizing access to a memory location is higher than a normal access. Using atomic instructions will reduce the output performance of the application if the latency of the atomic instruction cannot be hidden. Atomic instructions can be used at each level of the GPU memory hierarchy. The access conflict may happen between processing cores or within threads of the same block executing on the processing core.

In this chapter we investigate the performance characteristics of atomic instructions at each level when changing the number of atomic operations per memory location at each iteration. Different vendors have different implementations of atomic instructions for synchronizing access to a memory location. This could result in varying performance levels for each device when an atomic conflict load is applied.

Every atomic instruction, regardless of the arithmetic operation it performs, still includes a memory transaction and in this chapter we will also analyze the impact of atomic conflicts per cacheline to see whether reducing the amount of conflicts per cacheline while performing the same number of atomic instructions has any effect on performance.

The benchmarks performed in this section:

1. Inter block conflict: All blocks share the same part of memory for their transactions. Most memory transactions reside in the same cacheline.

2. Inter block conflict with cacheline padding: In this benchmark we create many segments (each with more than 1 cacheline padding space between them) on the output. We start with 1 activated segment and continue increasing the number of segments.

3. Intra block conflict: Each block of threads share a part of memory and conflicts can happen between threads of the same block. At each iteration only a segment of the block output is available which results in conflicts between threads within a block.

For better comparison between different vendors we put the results of Nvidia GTX1080 and AMD MI25 next to each other in the following charts. In these charts we are interested in the performance of the device for different conflict counts. At each iteration we select the number of segments that are active for use. The fewer active segments, the higher the probability of a conflict. The first charts are done using a padding between output locations so that each segment contains a full cacheline. In the intra block conflict charts, each block has its own segment which has as many elements as it has threads. At each iteration a number of items is activated for writing and all threads have to write to those locations only.

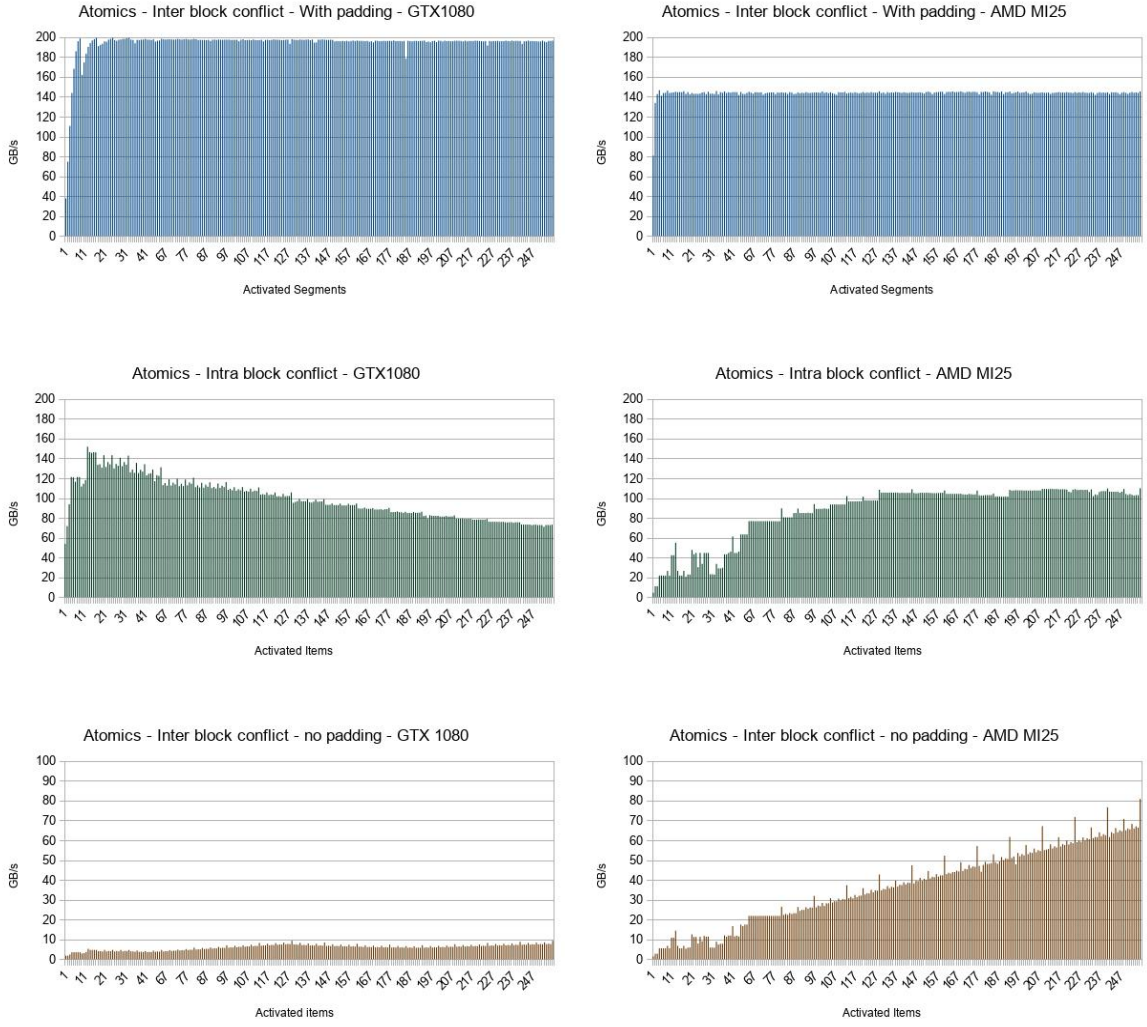


Figure 6.15: Atomic operation performance considering conflicts - scaled for both devices. Left column shows results for the Nvidia GTX 1080 and the right column shows results for AMD MI25. Each row shows the results of a different benchmark.

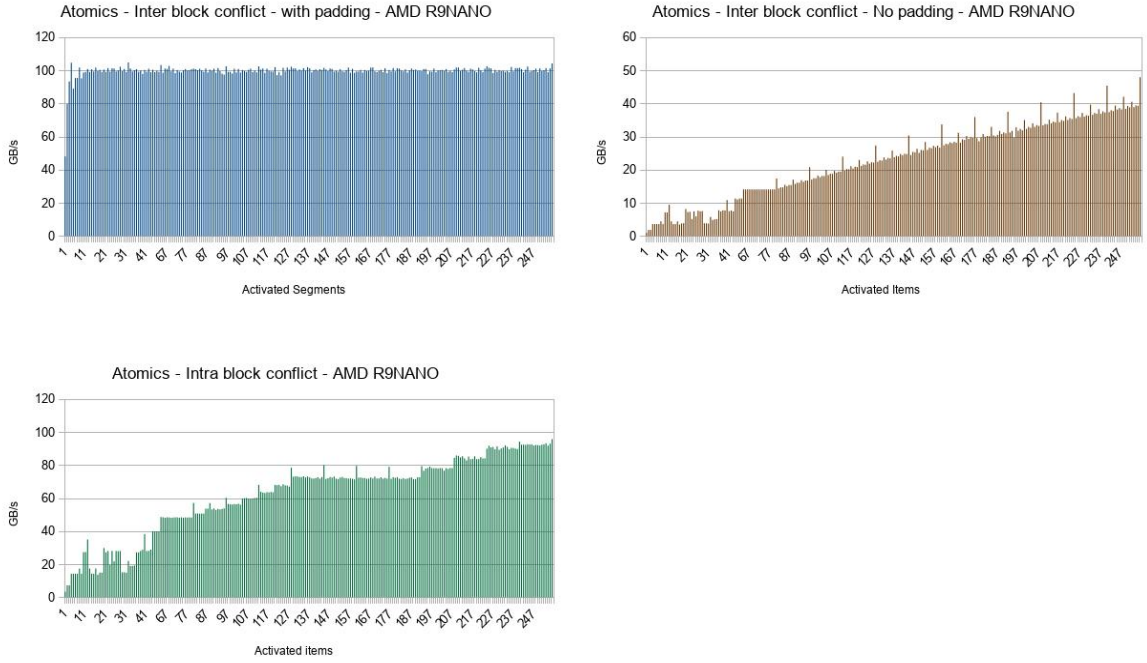


Figure 6.16: Atomic operation performance considering conflicts for AMD R9NANO.

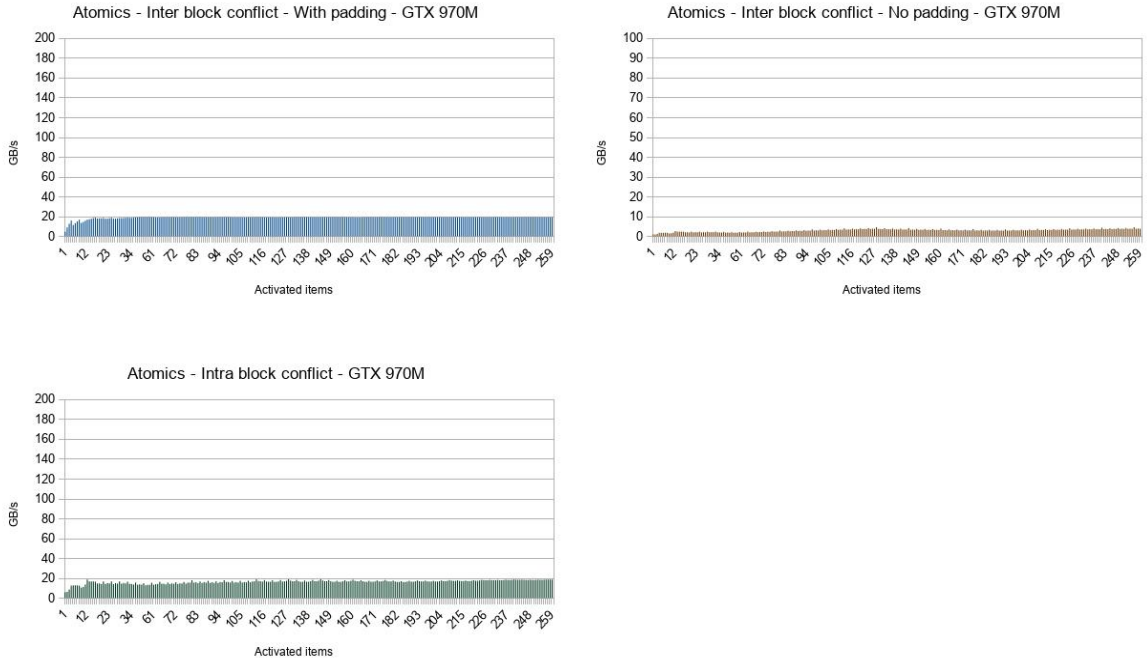


Figure 6.17: Atomic operation performance considering conflicts for Nvidia GTX 970M

The results indicate that different vendor's atomic implementations have different

behaviour for varying atomic loads. We focus on the GTX1080 and AMD MI25 results provided in Figure 6.13. Using a cacheline padding between output items we see that both devices quickly reach stable performance when the number of activated segments is more than 8. This means that the conflict between different executing blocks is no longer the bottleneck. However it is worth noting that AMD devices do not benefit as much compared to Nvidia devices.

When blocks of threads have internal conflicts, the GTX1080 gains performance when the number of activated items starts increasing, until it reaches *warp_size* number of activated items. After that the performance starts dropping by more items being activated. This suggests that limiting intra block conflicts to one warp and not allowing different warps to conflict with each other can help atomics performance. The AMD MI25 starts gaining performance steadily throughout iterations by activating more items. There is a significant jump at 64 items activated (again size of a warp in AMD devices) but continues to gain performance the more items are activated and hence the less conflicts happen between threads within a block.

Without using a padding between outputs the performance is decreased. The GTX1080 is the most vulnerable to this as using no padding increases the amount of conflicts per cacheline by the number of active blocks. Allowing more items to be active wont show any significant benefit. The AMD MI25 suffers from the same thing but it starts recovering performance when more items are activated. This suggests that there is a higher capacity for atomic instructions per cacheline within the AMD MI25 device compared to GTX1080.

7 Using the results of the framework

In order to optimize a GPU application using the results of Chapter 6, we must analyze the code to find the execution patterns for each of the modules discussed in Chapter 6. After that we can focus on each set of instructions and the module they are interacting with and decide which configuration is optimal for that particular module.

The first module to consider are memory transactions, because as the results have shown to have a big impact on performance if used in an inefficient way. When we look at memory transactions we have to consider data-types, size of transactions, number of cachelines used, and data dependency.

We can examine the application to see whether it can benefit from shared memory or not. In streaming processing in general it is assumed that the data is not going to be read or written more than once. In practice we find that a lot of algorithms reuse a set of data multiple times. If the set of data that is being reused multiple times is local to a group of threads or can be localized so that it is used by a block of thread, shared memory can be used as a replacement for global memory for that data set for better performance. Instead of having multiple requests to global memory, the data set is written to the shared memory once and then used by all threads for processing and then the results can be written back to global memory. This will help increase performance by reducing the amount of global memory transactions and replacing them with shared memory transactions which have a much lower latency.

And finally, if the application uses atomic instructions, we can analyze the atomic instructions used and see what is the optimal ordering and pattern for that device.

We select the histogram algorithm to apply our optimizations to. Histogram calculation is a common used method in image processing and machine learning. It was introduced by Karl Pearson in 1895. A histogram is a representation of the distribution of data over quantified bins. The naïve histogram implementation consists of reading the input data, finding out which bin the data belongs to and then increasing the value of that bin. In a sequential processing unit the histogram calculation will be done with an algorithm complexity of O(n). In this chapter we port a naïve histogram algorithm into a SIMT environment which uses GPUs as compute accelerators and try to improve it using our previous results.

M.E.R. Berger [MERB15] study on histograms shows that when running histogram algorithms in a SIMT environment, they mostly suffer from write-collisions. The histogram algorithms write conflicts closely resembles that of the birthday paradox: "In a group of 21 people, there is an approximately 50% chance of a collision in birthdays to occur". Atomic conflicts have an impact on the performance of the application.

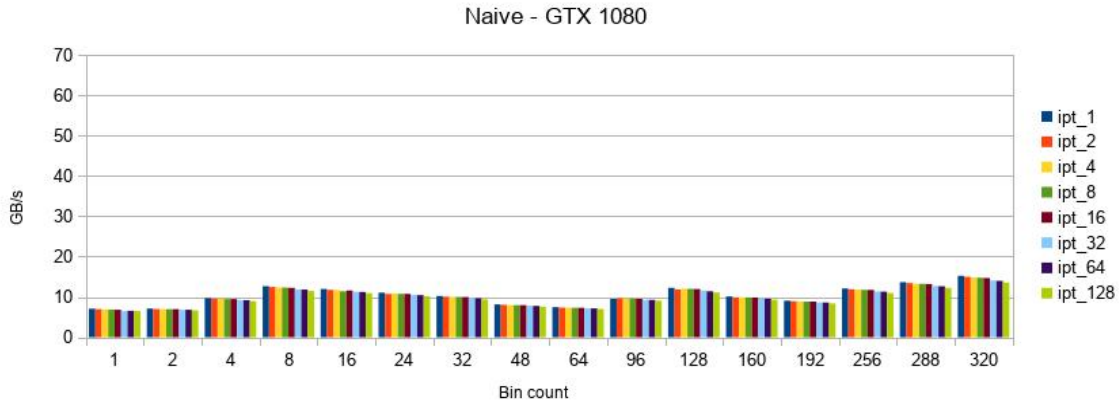
Atomic instructions were not available in the earlier GPU generations. They were implemented for each level (global, shared, ...) by each generation, which resulted in

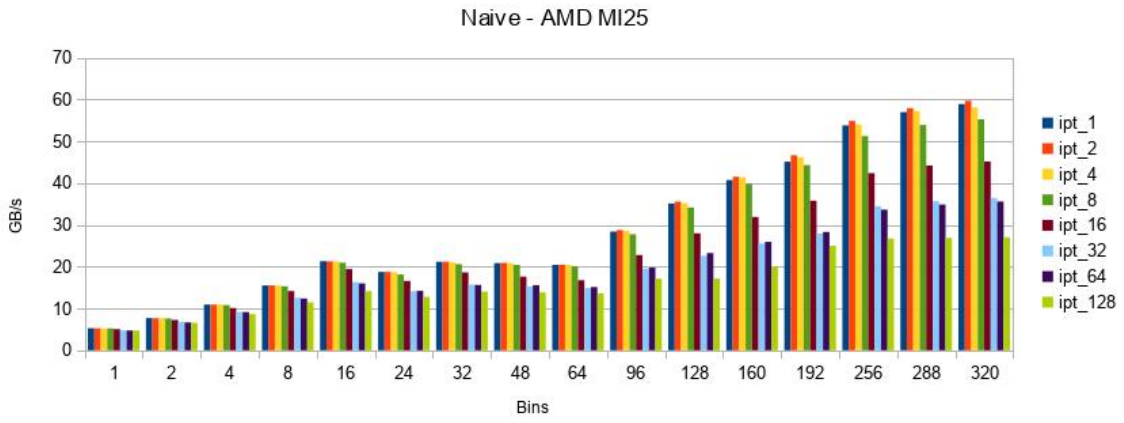
more performance in the histogram calculation as researchers started to use the divide and conquer technics combined with available atomic instructions such as the work done by [RS07] and [MERB15] in order improve histogram calculations performance using the current generation of hardware.

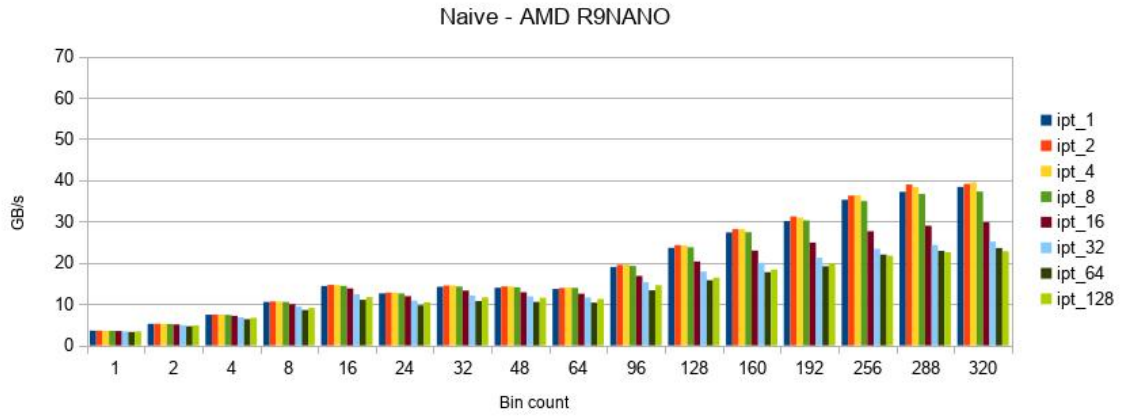
We start with the naïve implementation of the histogram algorithm. Afterwards we perform the steps mentioned in the beginning of this chapter to improve performance and show the results on multiple devices. The algorithm will be ran using different bin counts and different execution patterns on each of the available devices. For this benchmark we use a randomly generated input data. It should be noted that in general histogram performance is sensitive to input distribution, but in this study we are focusing more on the general optimizations than optimizations specific to histogram algorithm.

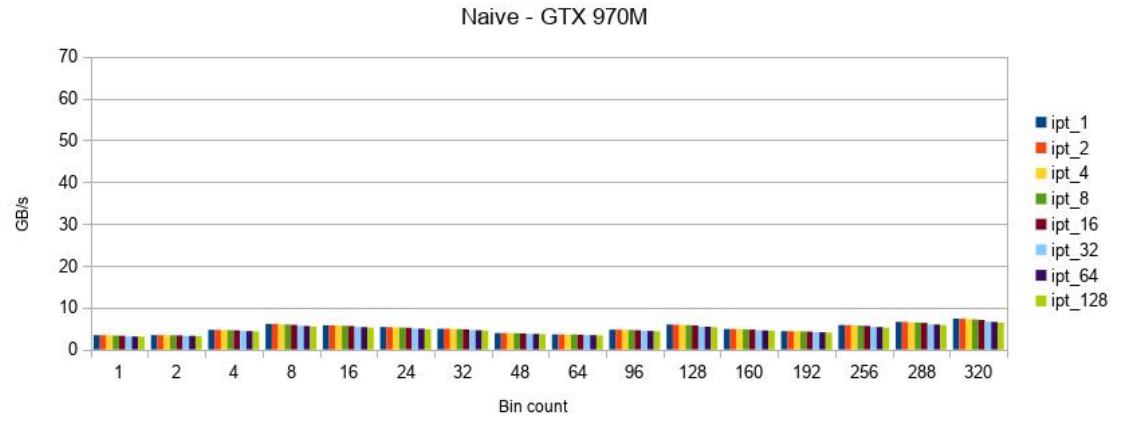
In the naïve implementation, we port the sequential CPU code to the GPU and run it using different configurations. In the results shown in Figure 7.1 it can be seen that the performance is low when the bin count is small and therefore the number of conflicts is high. With more bins available the performance goes up until reaching peak performance on all devices. However it is noticable that the Nvidia devices suffer heavily under high write conflict loads and the performance does not increase by a significant margin. In contrast, the AMD devices quickly start performing faster when the bin count is increased. This was shown in the previous chapter's results for atomics when no padding was used for output, see Figure 6.13, Figure 6.14. It is worth noting that changing the number of items per thread only slightly decreases the performance overall, which is due to reading input using the direct pattern in the naïve implementation.

Figure 7.1: naive implementation results



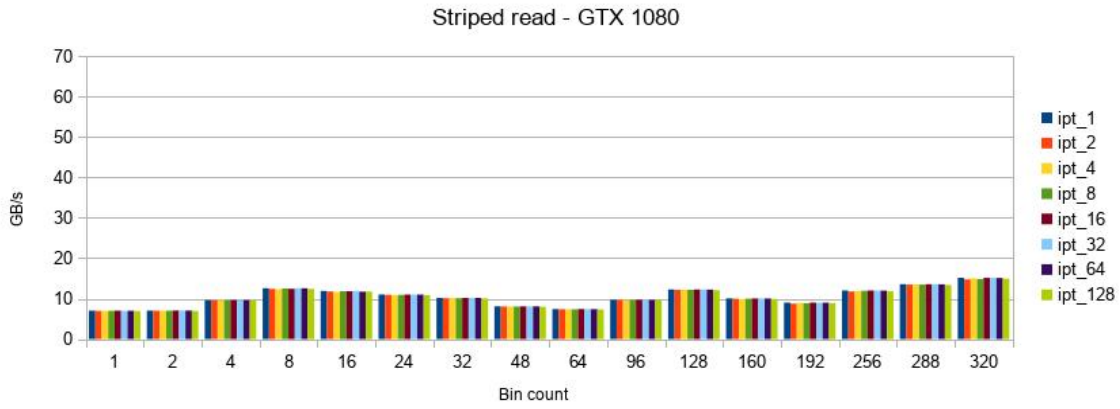


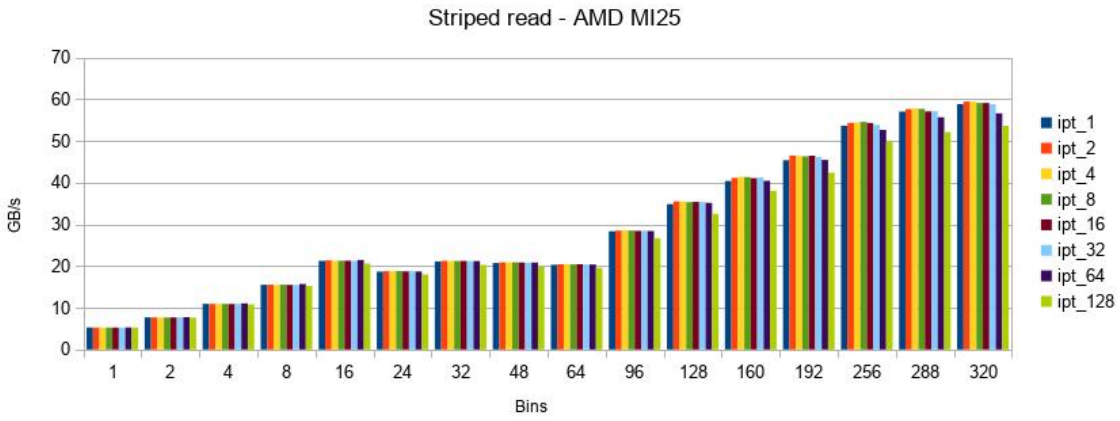


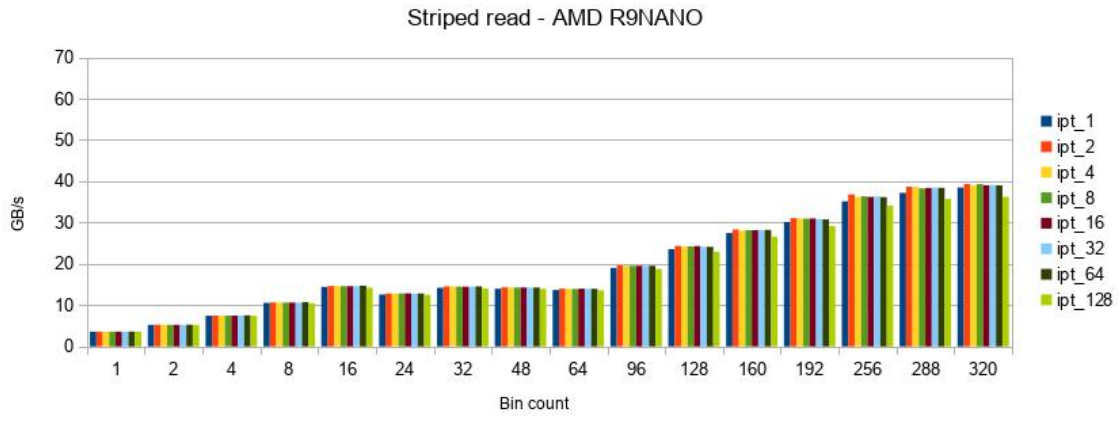


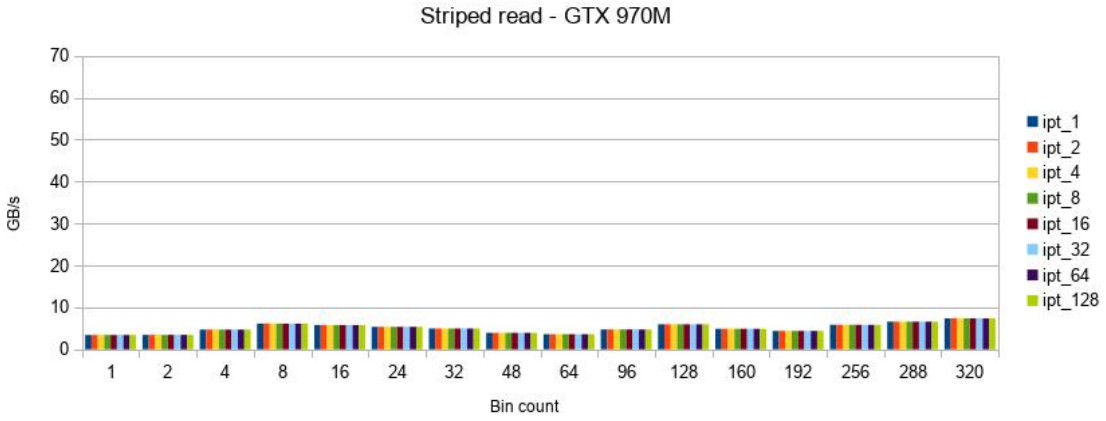
In previous Chapter, we discussed memory access patterns and cacheline utilization. We focus on reading data from the input and then on writing to output. In the next implementation we try to first change the input reading pattern. As discussed before, histogram algorithms suffer from write conflicts, and therefore changing the reading pattern should not have a large impact on the performance.

Figure 7.2: Striped read implementation results





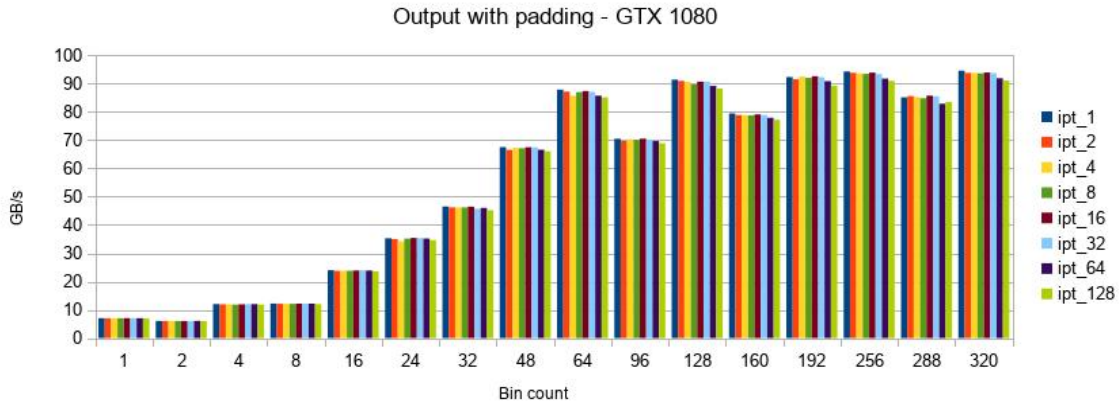


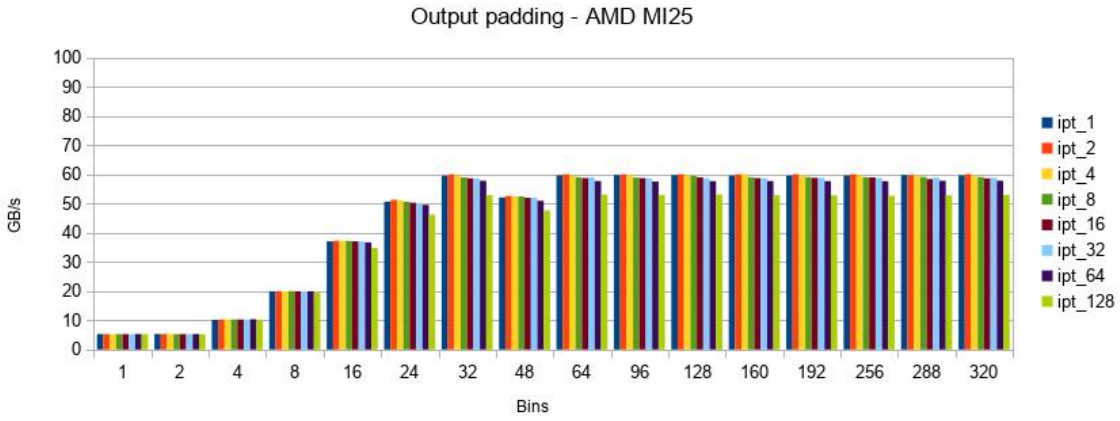


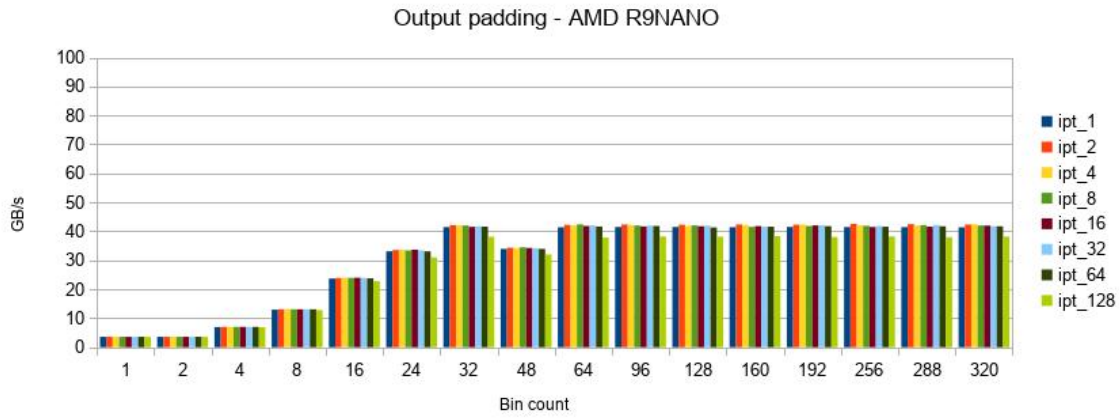
As shown by the results in Figure 7.2, the performance per bin count has not changed. However the performance is now more stable when using different number of items per thread as opposed to the naïve implementation especially on AMD devices.

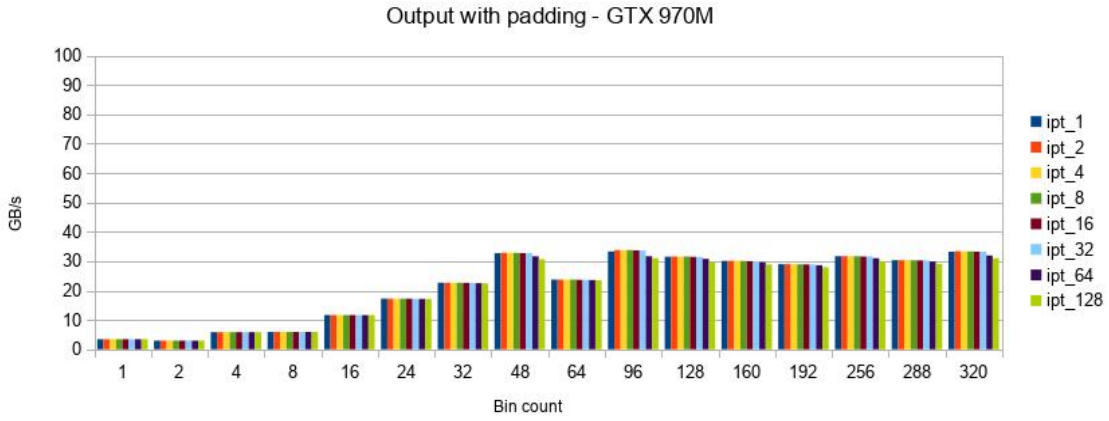
In the next implementation, we change the way the algorithm writes to the output. Based on our results from previous chapter, the amount of atomic write operations per cacheline can affect performance. We saw that adding padding to output so that the atomic operations are spread between multiple cachelines improved performance especially for Nvidia devices. In the next implementation we use a padding between our output memory locations so that neighboring output locations are in different cachelines. This may seem a waste of memory but in these benchmarks we are interested in performance.

Figure 7.3: Output with padding implementation results







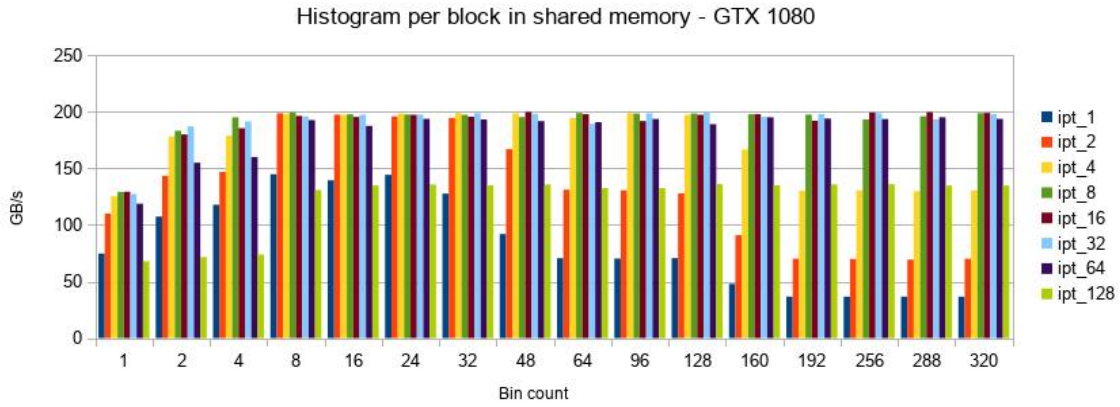


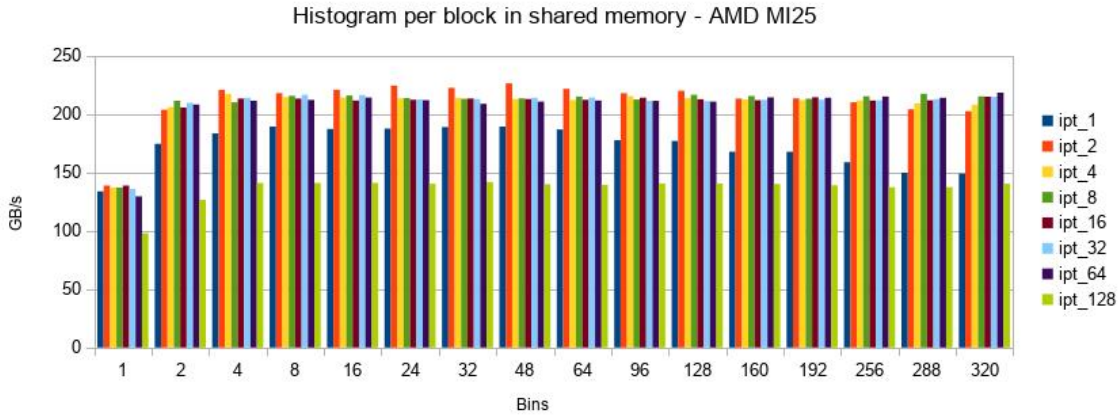
Looking at the results of Figure 7.3, we see that although all devices benefit from using a padding between output memory locations the AMD devices do not benefit when the bin count is higher than a certain number. Nvidia devices continue to perform better as the bin count increases. The difference suggests that we have reached a limitaion in the AMD hardware for atomics.

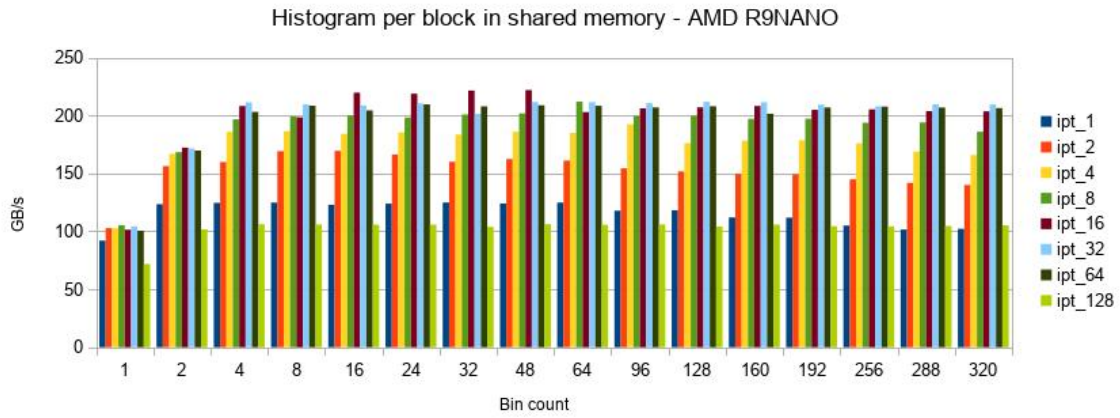
In the histogram algorithm, the output is being reused multiple times. This suggests that having a temporary output in shared memory for each block could result in fewer global memory transactions and conflicts. Therefore in this implementation we use the shared memory for temporary block output and after the block has finished processing the input the end result which is in shared memory will be then added to the global memory output.

In this implementation, we see that the performance increases when items per thread is increasing for all devices. When each block of threads has its own histogram in shared memory it will have to output them to global memory output when finished completing its own histogram. Therefore the total number of global memory atomic transactions is related to the total number of blocks required for the input data. By increasing the items per thread, the total number of blocks required will decrease and therefore there are less global memory atomic transactions after each block has finished processing. Therefore the performance will increase by increasing the number of items per thread until register spilling occurs.

Figure 7.4: Histogram per block in shared memory implementation results







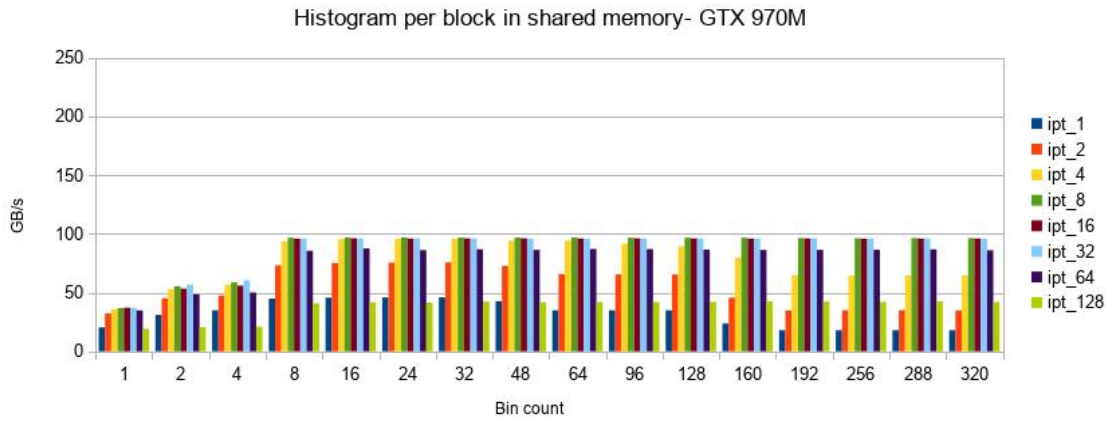
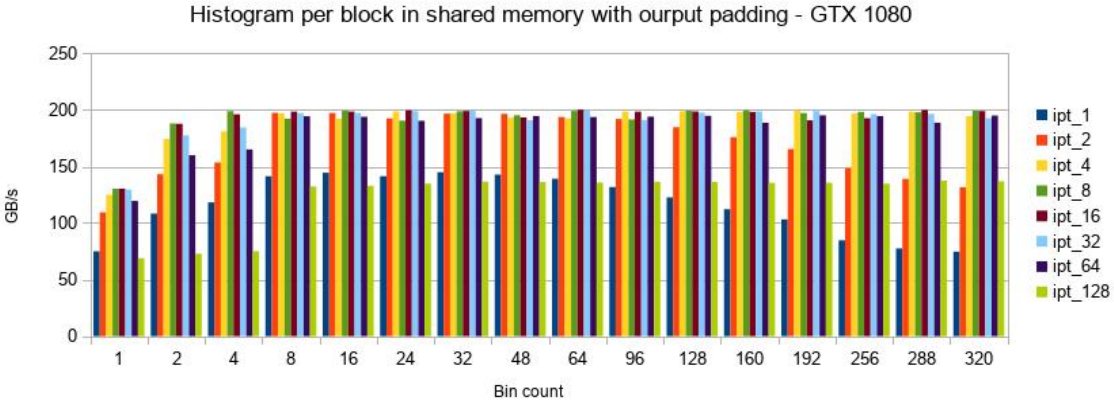
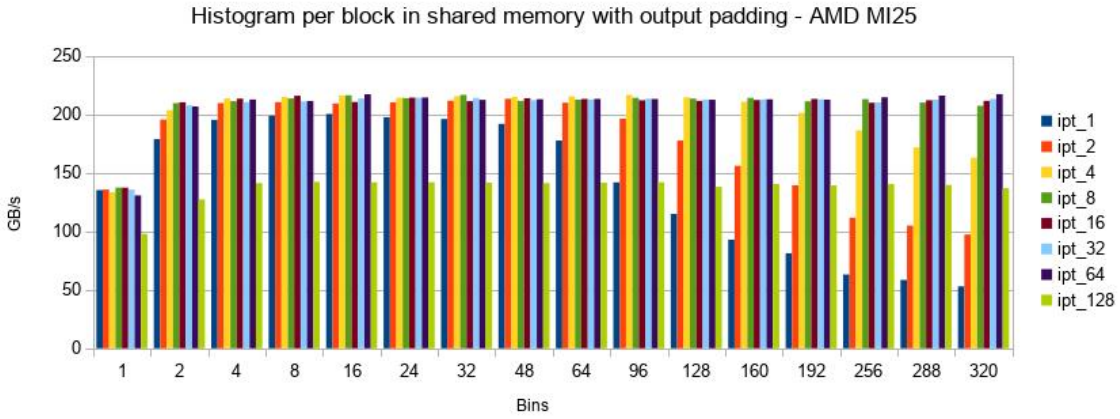


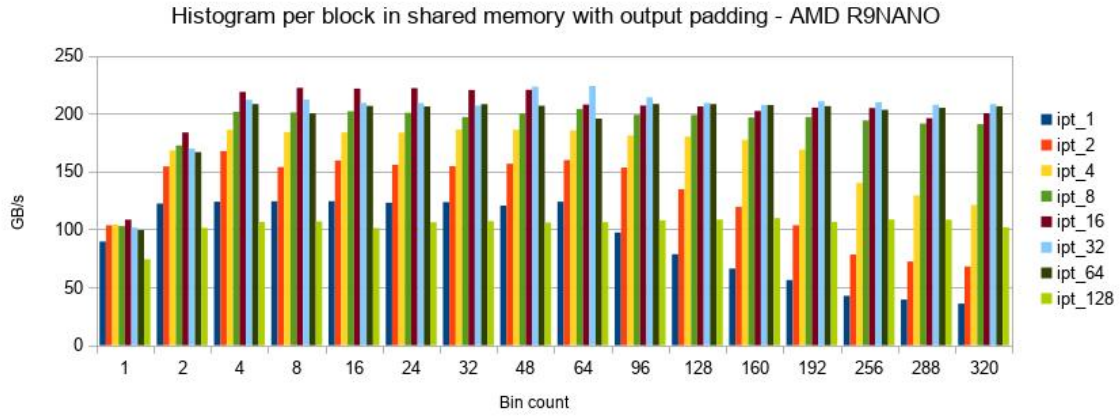
Figure 7.4 shows the results of the implementation. As the results show the performance is much better when using shared memory to reduce global conflicts and memory transactions. We see that the number of items per thread also starts influencing the performance as global atomic conflicts are reduced significantly. It can be seen that the Nvidia devices are more sensitive to items per thread and are reaching a bottleneck when using less items per thread. This suggests that at lower items per thread the implementation is not using the cacheline write capacity of each streaming processor to its fullest at low items per thread.

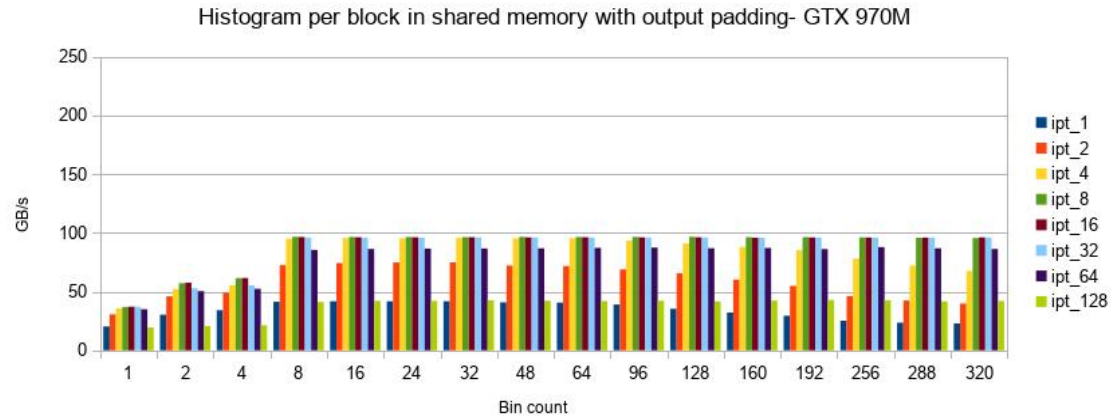
For the final implementation we add the output padding to our shared memory implementation to see whether it improves performance. This implementation is similar to the shared memory implementation and only differs in the writing pattern to the output array in global memory.

Figure 7.5: Histogram per block in shared memory with output padding implementation results









As shown in Figure 7.5, the Nvidia devices gain performance by using the padding specially when using fewer items per thread but the AMD devices do not benefit from this.

8 Conclusion

In this study we created a framework for benchmarking GPU hardware modules and evaluate their performance characteristics under different loads and patterns. The framework's source code is vendor independent and the HIP compiler can use both AMD and Nvidia compilers to create the machine code. This allowed us to write high level C/C++ code for both devices with minor vendor specific code, in order to observe the device behavior under out application load.

We selected low level hardware modules and created benchmarks to apply various loads to those modules and gathered their performance characteristics and presented them in Chapter 6. The results showed that each GPU device's performance characteristics is unique and in order to use that device in an optimal way, there are certain things to consider in execution order and pattern.

We selected an algorithm that makes use of all the hardware modules that were benchmarked. We created a naïve solution for the algorithm and proceed to improve its performance using the results of our experiments.

In the end we can conclude that when writing an application that is going to use a compute device such as the GPU, we have to be aware of the performance characteristics of the device the application has to be executed on. This helps us in choosing the best practices in accordance to our device and not making blind decisions that could affect performance.

In the future, work we would like to add an automation for extracting some hard coded numbers that we used like shared memory banks or cacheline size, in order to fully future-proof the framework in the case of any of the aforementioned values change with newer hardware generations.

Also creating and adding benchmarks which only focus on the ALU in the GPU will further increase our insight about the device and help us improve more algorithms by finding most efficient and optimal solutions, performance wise.

9 References

[AK12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. Advances in neural information.

[AM15] Abhinandan Majumdar, Gene Wu and Kapil Dev "A Taxonomy of GPGPU Performance Scaling"

[BS17] Ben Sander and Gregory Stoner, ROCm: An open platform for GPU computing exploration. https://github.com/ROCm-Developer-Tools/HIP

[BSR17] http://gpgpu10.athoura.com/ROCM$_G$*PGPU*$_K$*eynote.pdf*

[CA17] http://www.creativeapplications.net/news/gpu-performance-101/

[CB17] https://cullenboyytech.wordpress.com/17/08/10/what-is-parallel-processing-system/

[CN11] Cedric Nugteren Gert-Jan van den Braak Henk Corporaal Bart Mesman, High Performance Predictable Histogramming on GPUs: Exploring and Evaluating Algorithm Trade-offs

[CPP13] C.P.Patidar and Meena Sharma, Histogram Computations on GPUs Kernel using Global and Shared Memory Atomics

[CU18] https://docs.nvidia.com/cuda/pdf/CUDA$_C$*programming*$_G$*uide.pdf*

[DHK17] Dae-Hwan Kim Evaluation Of The Performance Of GPU Global Memory Coalescing

[EB11] E. Blem, M. Sinclair, and K. Sankaralingam, "Challenge Benchmarks that Must be Conquered to Sustain the GPU Revolution," in Proceedings of the 4th Workshop on Emerging Applications for Manycore Architecture.

[EK15] Elias Konstantinidis, Yiannis Cotronis, A Practical Performance Model for Compute and Memory Bound GPU Kernels

[GHIP17]
https://gpuopen.com/compute-product/hip-convert-cuda-to-portable-c-code/

[GHCC17]
https://gpuopen.com/compute-product/hcc-heterogeneous-compute-compiler/

[JB15] http://www.jonathanbeard.io/blog/15/09/19/streaming-and-dataflow.html

[JK14] Joonyoung Kim and Younsu Kim, HBM: Memory Solution for Bandwidth-Hungry Processors

[JWC65] James W. Cooley and John W. Tukey, An Algorithm for the Machine Calculation of Complex Fourier Series

[KH17] Koki Hamaya, Satoshi Yamane, Detecting Bank Conflict of GPU Programs Using Symbolic Execution

[KHR15] https://www.khronos.org/opengl/wiki/Fixed$_{F}unction_{P}ipeline$

[KHR92] https://www.khronos.org/opengl/

[KHR09] https://www.khronos.org/opencl/

[KK11] Kamran Karimi, Neil G. Dickson and Firas Hamze, A Performance Comparison of CUDA and OpenCL

[JF11] Jianbin Fang, Ana Lucia Varbanescu and Henk Sips, A Comprehensive Performance Comparison of CUDA and OpenCL

[MERB15] M.E.R. Berger, High Performance Histograms on SIMT and SIMD Architectures

[MH05] Mapping Computational Concepts to GPUs - Mark Harris

[MM13] https://www.microway.com/hpc-tech-tips/gpu-memory-types-performance-comparison/

[MOC14] Mike O'Connor, Highlights of the HighBandwidth Memory (HBM) Standard

[MZ17] Maohua Zhu, Youwei Zhuo, Chao Wang, Wenguang Chen and Yuan Xie, Performance Evaluation and Optimization of HBM-Enabled GPU for Data-intensive Applications

[NV13] https://nvlabs.github.io/cub/structcub$_{11}d$$evice_{h}istogram.html$

[NVCC18] http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

[NV18] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[NV01] http://www.nvidia.com/page/geforce3.html

[OR11] Ofer Rosenberg, Introduction to GPU Architecture

[PM11] Paulius Micikevicius, Local Memory and Register Spilling

[PM12] Paulius Micikevicius, GPU Performance Analysis and Optimization

[RS07] Ramtin Shams and R. A. Kennedy, Efficient Histogram Algorithms for Nvidia CUDA Compatible Devices

[SH09] Sunpyo Hong and Hyesoon Kim, An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness

[SL14] Sohan Lal, Jan Lucas, Michael Andersch Mauricio Alvarez-Mesa, Ahmed Elhossini, Ben Juurlink GPGPU Workload Characteristics and Performance

Analysis

[TP18]
https://www.tutorialspoint.com/parallel$_a lgorithm/parallel_a lgorithm_i ntroduction.htm$

[VV10] V. Volkov, "Better Performance at Lower Occupancy," in GPU Technology Conference, 10.

[WH14] W. mei Hwu, "What is ahead for parallel computing," Journal of Parallel and Distributed

Computing, vol. 74, no. 7, pp. 2574–2581

[WIKI07] https://en.wikipedia.org/wiki/CUDA

[WIKI13] https://en.wikipedia.org/wiki/Multi-chip$_m oduleChip_s tack_M CMs$

[WIKI STRP] https://en.wikipedia.org/wiki/Stream$_p rocessing$

[XM16] Xinxin Mei and Xiaowen Chu, Dissecting GPU Memory Hierarchy through Microbenchmarking

[YJIA14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. Arxiv,

[YJIAO10] Y. Jiao, H. Lin, P. Balaji, W. Feng, Power and Performance Characterization of Computational Kernels on the GPU