DTU Compute
Department of Applied Mathematics and Computer Science

High Performance Computing

Special Course

Alexander Rosenberg Johansen (s145706@student.dtu.dk)
Elias Khazen Obeid (s142952@student.dtu.dk)

Kongens Lyngby
January 25th 2016

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Contents

CHAPTER 1

Introduction

This report is a documentation of Alexander and Elias' work during the special course "High Performance Computing" at the Technical University of Denmark. The project stretched from 4'th of January 2016 to the 25'th of January 2016.

The development of this report is based on the book, "CUDA - Application Design and Development" by Rob Farber [3] and the online course "CS 344 - Introduction to Parallel Programming" by John Owens and David Luebke [13].

Our code base developed throughout this course is publicly available.[1]

1.1 Motivation

Forty years ago, Gordon E. Moore predicted that the number of transistors will double every two years. Until now this prediction has been correct and is widely known as "Moore's Law". Previously, increasing the transistors and thus computational power was achieved by making smaller and faster transistors, which in turn was utilized for higher clock speed. However, within the last decade we have reached the ceiling for clock speed due to power and heat challenges. The current approach has thus been focussing on creating many simple and power efficient computational units and utilizing these in parallel. [13, 11, 1]

The performance of processors is often divided into two areas, latency and throughput. Latency is the speed of which a single task can be completed, whereas throughput is the total amount of computation produced over a given time frame. [3]

The two different approaches of building processors translates to the Central Processing Unit (CPU) and General-Purpose computation on Graphics Processing Units (GPGPU). The CPU is focussed on reducing latency by having a few complex processors maximising clock speed for computing. The GPGPU on the other hand is designed for optimizing throughput using many simple and economical processors. However, the theoretical throughput that GPGPUs is based on the ability to utilize all of its simple processors at once.

To fully exploit the potential of the GPGPU a task has to be parallelisable and instructions developed with parallelisation in mind, especially as the memory structure of current GPGPUs are often the bottleneck in reaching the theoretical throughput.

[1] https://github.com/obeyed/dtu-hpc-course/tree/master/code

The applicational interests, which is one of our reasons for taking this course, are often based on tasks requiring linear algebra of massive matrices. Linear algebra is by nature a very parallelisable task as the problem can be reduced to smaller parallel subproblems using divide-and-conquer methods [12, 2]. As linear algebra has been the corner stone of machine learning, numerical analysis and graphics for decades we find the challenge of developing parallel algorithms for GPGPUs appealing, why we are taking this course.

1.2 Help

The style of work for the special course has relied on independent studies. To facilitate our independent studies we have utilized online learning platforms and online communities to handle questions and challenges. Below are a listing of our questions posted to the stack overflow community.

- Why does shared memory ensure coalesced writes [2]
- Debugging out-of-bounds errors [3]
- Large values in local memory [4]
- Developing a reduce algorithm [5]

Furthermore, we have used the GPU Lab at the Technical University of Denmark (DTU) to run our CUDA code. We connected with `ssh` to the gpu lab address with our username as

```
ssh username@login.gbar.dtu.dk
```

To load the needed modules and get access to the correct resources we used the following commands

```
qrsh && k40sh && module load cuda
```

We compiled our CUDA code with the `nvcc` compiler and ran it with the following command

```
nvcc -arch=sm_35 -o test cudaCode.cu && ./test
```

where `-arch` defines the compute capability of the device for which we aim to compile the code.

[2] http://stackoverflow.com/questions/34881190/shared-memory-writes-coalesced-writes
[3] http://stackoverflow.com/questions/34898507/cuda-out-of-bounds-write-when-transposing
[4] http://stackoverflow.com/questions/34655893/cuda-large-input-arrays
[5] http://stackoverflow.com/questions/34596490/cuda-reduce-algorithm

CHAPTER 2

Hardware

In this chapter we present the general architecture of CPU (section 2.1), general architecture of a GPGPU (section 2.2), GPGPU memory architecture (section 2.3) and a discussion of the difference between CPU and GPGPU (section 2.4).

2.1 CPU Architecture

The Central Processing Unit (CPU) is the processing unit in a modern day computer. The CPU handles processing and control of data in the computer's memory space. The CPU is usually made of a small amount of complex cores, referred to as Multiple Instruction, Multiple Data (MIMD) Arithmetic Logic Units (ALU) that are optimized for reducing the latency of a given task.

The bottleneck of computation is often placed at the memory's ability to transfer data to/from the CPU, also known as Input/Output (I/O). To increase the I/O the CPU has a built-in memory architecture with high speed, but low storage capabilities. The optimized memory architecture allows the CPU the store, "cache", memory for later reuse and thus reduce the memory bottleneck. [1]

2.2 General GPU Architecture

The General-Purpose computation on Graphics Processing Units (GPGPU) is a microprocessor that performs computation traditionally handled by the CPU. In our project we only work with CUDA-enabled devices, but we assume that the features of CUDA-enabled devices are equivalent to those of GPGPUs for the ease of notation. The GPGPU consists of a large amount of simple and economical processing units that individually are weaker than those of the CPU, but combined possess up to several magnitudes of larger throughput capabilities. The GPGPU is attached to the CPU such that the CPU utilizes the GPGPU to accelerate computation that the GPGPU is more suitable at handling, e.g. processes that are parallelisable.

The GPGPU's architecture is based on a parallel scheme, which has promoted an architecture of independence among processing cores on the GPGPU. The basic building block of this architecture is the streaming multiprocessor (SM) shown in fig. 2.1, which is independently responsible for its own resources, cores, and memory. By making each SM independent, memory access and computation are performed faster as the memory is placed physically closer to the cores. The cores on each SM are Single Instruction,

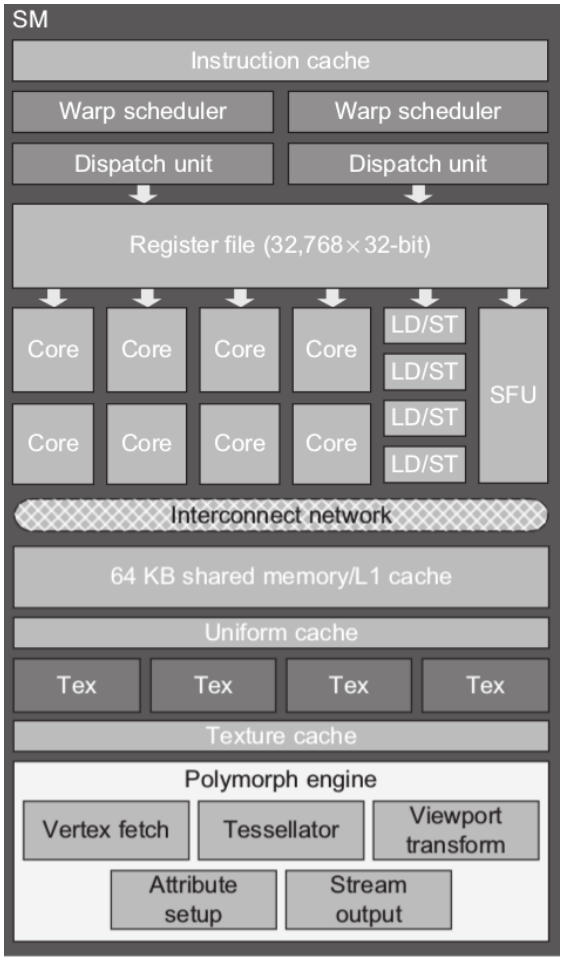


Figure 2.1: Example of a Streaming Multiprocessor for reference [3]

Multiple Data (SIMD) ALUs, which means that the cores are constrained to run the same piece of code, though with different data. The work given to the GPGPU is distributed to the SMs by the Giga-Thread global scheduler. The Giga-thread scheduler holds metainformation on the SMs, which allows it to optimize workloads across the GPGPU's independent processors. The SMs receive instructions from the scheduler in a cache and schedules the work for execution on the cores in the SM [3]. We will go more into depth regarding the specifics on scheduling of independant processes in chapter 3.

At runtime, every independent process has access to its own registers called local memory. Each SM has shared memory for high-speed data sharing between threads in a block called shared memory. To supply the SMs with data the GPGPU contains a larger amount of global memory available to all SMs.

Furthermore, a SM has load/store (LD/ST) units and Special Function Units (SFU). LD/STs calculate source and destination addresses and load/store data as needed. SFUs

execute special functions, e.g. sin, sqrt, etc. Compared to the SIMD cores, the SFUs are designed specifically to perform their designated functions whereas the SIMD cores are general purpose cores. [4]

2.2.1 Hardware Specific Numbers

Using CUDA functionality we are able to extract hardware specific numbers from the GPGPU. Throughout this report, we will be using the Tesla K40 GPU [8]. For future reference table 2.2 presents the specifications for our GPU device. In appendix A we describe which commands were used to find the specifications for our device.

item	limit
warp size	32
max threads / block	1024
total cores	2880
registers / block	65,536
constant memory	65,536 B
L1 cache / block	49,152 B
L2 cache / core	1,572,864 B
global memory	12,079,136,768 B

Table 2.2: Tesla K40 GPU's specifications

Furthermore, the maximum dimension of grids and blocks are presented in table 2.3.

item	x	y	z
block size	1024	1024	64
grid size	2,147,483,647	65,535	65,535

Table 2.3: Tesla K40 GPU's block and grid sizes

2.3 GPGPU Memory Architecture

This section aims to give a brief overview of the memory architecture of GPGPUs.

Figure 2.4 illustrates the connections between the different layers of memory. The data moves from the CPU's main memory, known as host memory, to the GPU's main memory, known as global memory. From the global memory the data is dispatched to the various SMs. It is worth noticing that transferring data from host to global memory and from global memory to the SMs are very expensive operations as they are several magnitudes slower than memory transaction between memory closer to the cores.

The L2 cache fetches data in a Least Recently Used (LRU) fashion, which means that it is built for temporal locality, i.e. recently accessed data is assumed to likely be used again

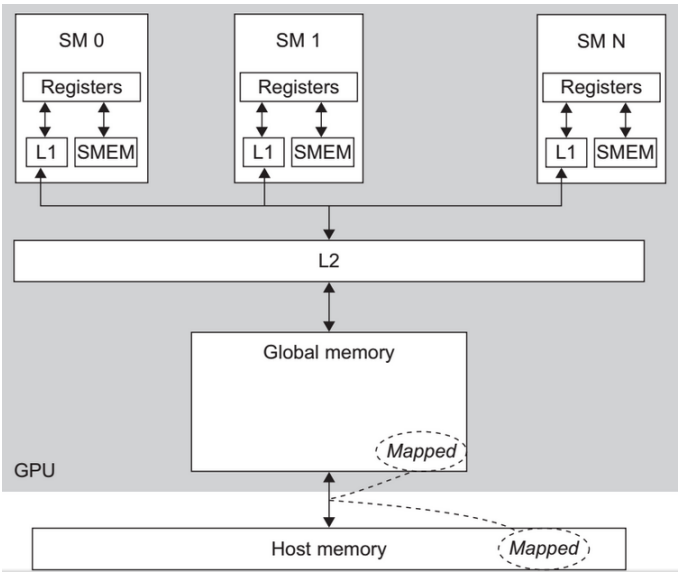


Figure 2.4: CUDA memory hierarchy [3]

in the near future. The L1 cache is designed for spacial locality, i.e. adjacent memory locations to the wanted data is also cached, with the assumption that neighbouring data will be used in the near future. This means that the memory that is inside the SM is best suited for coalesced memory access (see section 4.4.1).

memory	speed
register	$\approx 8000\,\mathrm{GB/s}$
shared	$\approx 1600\,\mathrm{GB/s}$
global	$177\,\mathrm{GB/s}$
mapped	$\approx 8\,\mathrm{GB/s}$

Table 2.5: GPU memory bandwidth [3]

Table 2.5 summarises the transfer rates of different connections of the different types of memory.

2.4 Do we ditch the CPU, and keep the GPGPU?

It is true that a GPGPU has many more cores than a CPU, but the cores are significantly slower than the ones in a CPU. The GPGPU neither have features for more general computer such as interrupts and virtual memory (used in modern day operating systems). As a result, the two processing units are developed with two different goals and thus have different characteristics.

As every type of problem might not have a parallel solution, and writing parallel code is a lot trickier than serial code, the CPU is still very much needed. The role of the GPGPU has thus evolved to be an ”accelerator” to the CPU, a specialized computing unit to accelerate parallelisable tasks. At runtime the CPU will thus initialize the GPGPU and send it data to process when faced with such parallelisable tasks, the relationship is called a host (CPU) to device (GPGPU) relationship.

CHAPTER 3

Software

In this chapter we present a deeper explanation of grids, blocks and threads (section 3.1), kernels (section 3.2) and challenges with parallel programs (section 3.3).

3.1 Grids, Blocks, and Threads

In section 2.3 we introduced the notion of scheduling and independent processes on the GPGPU. In this section we will further elaborate on the scheduling paradigm used in CUDA GPGPU programming.

The computation on the GPGPU is done through the use of threads. Threads are scheduled concurrently in warps by blocks, where the blocks are organized in grids. A thread is an independent sequence of work where the sequence is defined by instructions of the kernel and multiple threads can run concurrently. A thread has an ID, being the given threads number in a set of threads, which it uses to coordinate its work without exchange of information between other processors.

To organize threads and utilize the ability to communicate between threads on an SM where memory access is fast, the concept of blocks has emerged. A block contains a set of threads, which has a some amount of memory (L1 cache) reserved on the SM and it is able to synchronize between threads at any point along execution of a kernel. It further ensures that every thread is run before terminating the block.

As a single block might not be able to carry out all work in a job, a grid is defined to numerate the amount of blocks that is required to support computation of the job. However, the grid does not support the same type of operations as the block, such as synchronization at a specific point in the kernel or fast shared memory. The reason is that a grid might contain more blocks than a single SM can handle, why shared memory would not make sense. Further, the total amount of blocks might be larger than what the GPGPU can support at once, why deadlocks can appear if inter-kernel synchronization was supported. The only feature supported by the grid, is that it finish launching all blocks in a kernel before starting the next kernel - why using kernel wrappers might be required in some algorithms that are designed with synchronization across blocks. As the grid does not provide guarantees of how many blocks to run at once or where to run them, the grid allows the scheduler to fit the workload onto any GPGPU supporting the CUDA instructions in the kernel and device specifications, e.g. block size.

When computing threads in a block the given SM uses a warp scheduler to launch the kernels onto the cores on the GPGPU. The warp scheduler launches a fixed amount of

threads (which is known as a warp) and executes the threads simultaneously. The warp further forces every thread to execute the same workload at every instruction (as the cores are SIMD). If the block does not have sufficient threads to fill the warp's capacity the warp will execute empty threads resulting in potential lost computational power. In section 4.5.2 we will go more into depth of the performance penalty when packing a number of threads not a multiple of the warp size.

From a development perspective the position of each block within the grid and each thread within its block is fetched with the built-in variables presented in table 3.1, which all have three dimensions (`x, y, z`).

variable	explanation	type
`gridDim`	dimension of grid	`dim3`
`blockIdx`	block's index within grid	`uint3`
`blockDim`	dimension of block	`dim3`
`threadIdx`	thread's index within block	`uint3`

Table 3.1: Built-in variables that are only valid within functions that are executed on device

These built-in variables are of type `uint3` and `dim3`, which are integer vectors. The variables are used to specify dimensionality and receive position when running a kernel. When constructing the `dim3` variable all unspecified components are set to 1. As presented in section 2.2.1 the max number of threads per block is 1024 for the device we will be working with. [9]

3.2 Kernels

In section 2.2 we introduced the software perspective of how to run code on the SIMD ALU cores on the GPGPU. In the programming language CUDA a kernel is a `C++` function with the `__global__` identifier in front of its declaration as illustrated in listing 3.1. This function first computes the thread's index (`mid`) in the block, then checks if it is out of bounds, and finally copies the value at `mid` from `d_in` to `d_out`.

```
1 __global__
2 void map_kernel(const int *d_out, int *d_in, const int size) {
3   // thread ID
4   const int mid = threadIdx.x + blockDim.x * blockIdx.x;
5   // check if out of bounds
6   if (mid >= size) return;
7   // copy item by thread's index
8   d_out[mid] = d_in[mid];
9 }
```

Listing 3.1: Kernel declaration example

The map kernel illustrated starts by computing the ID of the thread, `mid`, by using its thread and block index. As the problem of mapping might not fit perfectly into a number

of blocks it is necessary to launch additional blocks, because the amount of threads for one block might exceed the block's maximum amount of threads. Because of the risk of an exceeding number of blocks the kernel checks the `mid` for being out of bounds.

Recall that the cores of a GPGPU are SIMD cores as presented in section 2.2. As a result the kernel will be executed by every thread on the GPGPU. The threads then use their `mid` to access different data, such as setting device output, `d_out`, to being the device input, `d_in`, in the mapping kernel.

3.2.1 Calling a Kernel

A kernel is called from the main program by calling the function with triple chevrons, which encapsulate the number of grids, blocks, and threads to run the kernel. This is illustrated in listing 3.2, where the kernel is called with 2 blocks with 128 threads per block.

```
1  int main(int argc, char **argv) {
2    const int SIZE = 1<<8,
3              BYTES = SIZE * sizeof(int);
4    const dim3 BLOCK_SIZE(128),
5               GRID_SIZE(2);
6
7    // set up host memory
8    int h_in[SIZE], h_out[SIZE];
9    for(int i = 0; i < SIZE; i++) h_in[i] = 3;
10
11   // set up device memory
12   int *d_in, *d_out;
13   cudaMalloc((void **) &d_in, BYTES);
14   cudaMalloc((void **) &d_out, BYTES);
15
16   // copy host memory to device memory
17   cudaMemcpy(d_in, h_in, BYTES, cudaMemcpyHostToDevice);
18
19   // call kernel
20   map_kernel<<<GRID_SIZE, BLOCK_SIZE>>>(d_out, d_in, SIZE);
21
22   // copy device memory from device to host
23   cudaMemcpy(h_out, d_out, BYTES, cudaMemcpyDeviceToHost);
24
25   // testing the mapping kernel has executed successfully
26   for (int i = 0; i < SIZE; i++) assert(h_in[i] == h_out[i]);
27
28   // print content of array
29   for (int i = 0; i < SIZE; i++) printf("%d: %d" i, h_out[i]);
30
31   // free GPU memory
32   cudaFree(d_in); cudaFree(d_out);
33
34   return 0;
35 }
```

Listing 3.2: Calling a kernel

First, we construct the size, `SIZE`, of the array to be computed and the amount of bytes, `BYTES`, of the array. We then construct the number of threads as a multiple of warp

size and small enough to fit into a block, followed by the grid size, `GRID_SIZE`. The host's (CPU's) input `h_in` and output array `h_out` are declared and the input array `h_in` is populated. The device's (GPGPU's) input `d_in` and output array `d_out` are declared and device memory is allocated to the arrays using `cudaMalloc()`. It is good naming practice to call variables on the CPU, the host, with the prefix `h_` while GPGPU variables, the device, with the prefix `d_`. The contents of the input `h_in` is copied to `d_in` using `cudaMemcpy()` where the transfer is from host to device, which is dictated by the `cudaMemcpyHostToDevice` keyword.

The kernel is launched with the `GRID_SIZE, BLOCK_SIZE` given as arguments, which instructs the scheduler to launch a grid of 2 blocks with 128 threads each, giving a total of 128 threads $\times$ 2 blocks = 256 threads. Further the kernel is given its required input parameters, `d_out, d_in` and `SIZE`.

Finally, the output from the device, `d_out`, is copied back to the host's array, `h_out`, by a device to host transfer, `cudaMemcpyDeviceToHost`. The contents is asserted to be correct and printed, the GPU's memory is freed using `cudaFree` and the main function returns 0, indicating everything went well.

A kernel takes up to four arguments within the triple chevrons. The first and second arguments are of type `dim3`, and the third and optional argument determines how much shared memory, per block, to allocate on the device, and the fourth optional argument defines which stream to run the kernel on.

3.3 Challenges with Parallel Programs

Parallel computing has a number of challenges related to concurrency such as deadlock and race conditions, which we will discuss in this section.

As depicted in fig. 3.2 we will consider three different cases of challenges with parallel programming: The map, the gather and the scatter operation. The map operation is a trivial parallel algorithm which do not pose any parallel challenges. Often map operations will have fast parallel implementations as we will show in section 5.2 and section 6.1. The gather operation is more difficult to handle as it has race conditions because it is a many-to-one operation, which we also present in section 5.3 and section 5.5. The scatter operation also suffers from challenges with race conditions such as shown in section 5.4.

A program is in a deadlocked state if two or more threads are waiting for the other to finish, resulting in neither ever finishing. This results in a program that will never advance and is thus considered being in a "deadlock". A race condition is a situation where two or more threads attempt to perform two or more operations at the same time, where these operations must be performed in a proper order to be correct. For instance, simultaneous update operations to the same memory address will cause a race condition. [3]

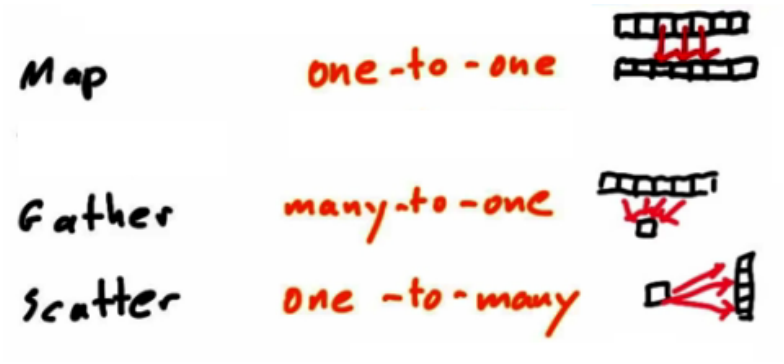


Figure 3.2: Parallel communication patterns

To give an example on how a race condition arises we present a naive implementation of a histogram in listing 3.3. In the naive histogram we have no guarantee that the value of `d_bins` have not changed between reading the value at line 9 and writing to it at line 10. With no guarantee of having a correct representation of the bin, the thread might make an unintended write to the array as many other threads could have tried to increment in between. In order to handle this race condition we need to utilize the synchronization tools given through CUDA.

```
1  __global__
2  void naive_histogram_kernel(int *d_bins, const int *d_in, const int SIZE) {
3    int tid  = threadIdx.x + blockDim.x * blockIdx.x;
4    // fetch item from input
5    int item = d_in[tid];
6    // compute where to put item wrt. bin count
7    int bin  = item % SIZE;
8    // update amount for that bin
9    current_value = d_bins[bin];
10   d_bins[bin] = current_value + 1;
11   // atomicAdd(&(d_bins[bin]), 1);
12 }
```

Listing 3.3: Naive histogram implementation with race condition

One solution to this problem is to use the atomic function where the given memory is locked while a thread is updating it and unlocked when the thread has completed the update. By replacing line 9 and 10 with line 11 the kernel will execute an atomic operation. The atomic operation has a slight overhead, but the cost is small [13]. The issue with using atomic is that it will force a serial access to a memory slot. In the worst case, when size of bin is one, the GPGPU is forced to act in a serial manner.

CHAPTER 4

Optimisation

In this chapter we present an introduction to analysis of optimisation (section 4.1), an introduction to parallel problems (section 4.2), software optimisation (section 4.3), memory optimisation (section 4.4) and hardware optimisation (section 4.5).

4.1 Introduciton to Analysis of Optimisation

In order to do optimisation we need to first understand where the algorithm spends its time and afterwards analyse how much we are able to speed up this process. In understanding the problem and bottlenecks we utilize the profiling tool by nvidia called `nvprof`, elaborated in section 4.1.1. For analysis purposes we approach the problem from two perspectives, a theoretical based approach, in section 4.1.2 on understanding the relationship between parallelisability and the number of cores and a more hardware specific, elaborated in section 4.1.4, where we analyse theoretical memory bandwidth usage. Hardware-wise the bottleneck is often to read/write to global memory. As many of our tasks are relatively simple and does not require much computation we will only consider theoretical memory bandwidth as a concrete performance measure.

4.1.1 Profilling

To understand our problem from a runtime perspective we must quantify the execution time in metrics of the different events in our program. An event is some quantifiable activity, action, or occurrence on a GPGPU device. Metrics are a set of runtime descriptions of one or more events.

In order to gather metrics about our events we utilize the built-in CUDA tool `nvprof`, which is a command-line debugging tool to profile compiled CUDA code. The property of being a command-line tool makes `nvprof` especially useful for us as we are limited to executing code on remote servers.

By default `nvprof` gives a short summary of how much time was spent on the different invocations. The debugger is invoked as follows

```
nvprof [options] [CUDA-application] [application-arguments]
```

To give a detailed metrics description the flag `--print-gpu-trace` can be utilized, this flag will print a list of all kernel invocations. Further `--print-gpu-trace` will show metrics such as the amount of memory used, where it is used, what the memory's transfer rate

was, on which GPGPU the kernel ran, the execution time, etc. [10] Aside from writing its results to the terminal it is also possible to write the profiling results to a file. Furthermore, to gain insight into opportunities for optimisation the `--analysis-metrics` flag allows nvprof to capture statistics for analysis of the GPGPU's metrics. These statistics can be used with nvidia's visual profiler to perform a guided analysis which we utilizes in section 6.2. [7]

4.1.2 Amdahl's Law

Amdahl's Law approximates the potential speed up of a serial program. The equation is presented in eq. (4.1), where P is the portion of the serial code that can be parallelized, $(1 - P)$ is the portion that cannot be parallelized and n is the amount of processors available. Thus, $S(n)$ is the theoretical speed up achievable while holding the workload constant.

$$S(n) = \frac{1}{(1 - P) + P/n} \tag{4.1}$$

Amdahl's Law only applies if the amount of work performed in the parallel version is not significantly different than the serial code's amount of work, this is also known as strong scaling. An illustration of the potential speed up is presented in fig. 4.1 with $n = 1024$. The model thus theorize that given a fully parallizable problem, the problem should be executed n times faster [3].

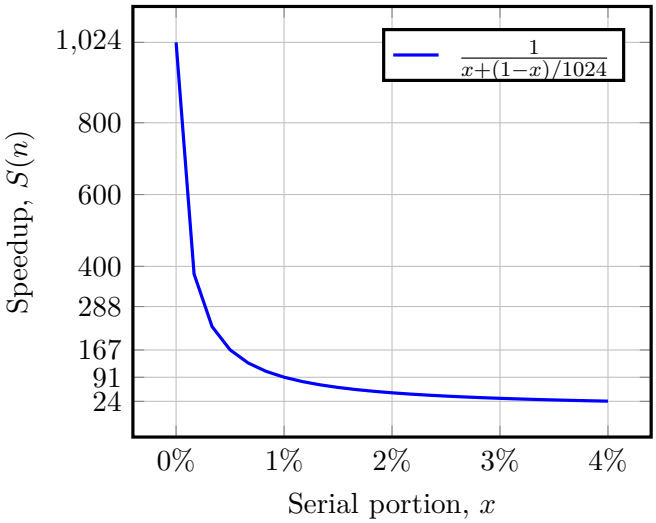


Figure 4.1: Speed up by Amdahl's Law, where $x = (1 - P)$, $(1 - x) = P$, and $n = 1024$

An important property of Amdahls is the decline of the curve as the solution moves from 0% to 1%. With parallel portion $P = 0.99$ the speed up is approximately 91×, and 1024× when $P = 1.00$ and everything can be executed in parallel. According to

Amdahl's Law, with just a tiny portion of the code that cannot be executed in parallel, a high speed up is not likely to be achieved.

4.1.3 Gustafson-Barsis Law

Gustafson and Barsis Law approximates how much more throughput can be achieved by increasing processor count given a constant amount of time. This type of formulation is often interesting when computing problems that are open ended such as computing pi – given more computational power, the processors can compute more digits of pi in the same time. It is also known as weak scaling. The amount of extra work is computed as [12]

$$W(n) = n + (1 - n) \times (1 - P) \tag{4.2}$$

where P is the amount of the program that can be parallelised and $W(n)$ is the theoretical increase in throughput over a defined period of time. [5].

4.1.4 Theoretical Memory Speed Up

In order to access the usage of the theoretical memory bandwidth in our optimization we can use the hardware specific numbers to see how much data we should be able to push through the GPGPU within a given time interval. We need to find the clock rate to determine how fast the processors are running and the bus width to determine the amount of data transferred at every clock multiplied by 2 due to double data rate.[1] The formulation for theoretical memory bandwidth thus becomes

$$\text{output} \times \frac{\text{bytes}}{\text{second}} = \frac{\text{clock}}{\text{second}} \times \frac{\text{bytes}}{\text{clock}} \times 2$$

Reading out `deviceQuery` information, as presented in appendix A, we can assess that `Memory Clock Rate: 3004 Mhz` and `Memory Bus Width: 384`. Thus out theoretical memory bandwidth on out K40 is

$$288.384 \times \frac{\text{GB}}{\text{second}} = 3.004\text{Ghz} \times 48 \times \frac{\text{bytes}}{\text{clock}} \times 2$$

In order to understand how close our solution is to the theoretical memory bandwidth we use the following formula

$$\text{percentage of theoretical} = \frac{\frac{\text{read/writes in GBs}}{\text{time in seconds}}}{288.384} \times 100.$$

[1] http://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/

4.2 Introduction to Parallelism

The goal of using GPGPUs is to solve computational problems, either faster or bigger than what was previously possible with CPUs. Often a simple port of a piece of code, given it has a parallel nature, will resolve in a significant speed up (3x to 5x) [13]. However, utilizing the software and hardware it is possible to go way beyond the initial speed up. The principles of GPGPU programming when coding for efficiency is to maximize useful operations by minimizing memory bottlenecks and divergence that force waiting time amongst threads.

It is useful to abstract the optimisation to different levels as some optimisations might be more useful than others.

1. Picking good algorithms
2. Basic principles for efficiency
3. Architecture specific detailed optimisations
4. Micro optimisations

The single most important element in optimisation is to pick an algorithm with strong asymptotic bounds. Optimising insertion sort $\mathcal{O}(n^2)$ as opposed to merge sort $\mathcal{O}(n \log n)$ would make even a naive implementation of merge sort vastly superior to a very optimised version of insertion sort. We will discuss algorithms specific for parallel coding in section 4.3

Further, when considering algorithms for parallel purposes, the parallelisability in an algorithm is of importance. When designing algorithms we will view the computations as a directed acyclic graph (DAG). This graph will have a set of computational steps linked together from top till bottom such as illustrated in fig. 4.2.

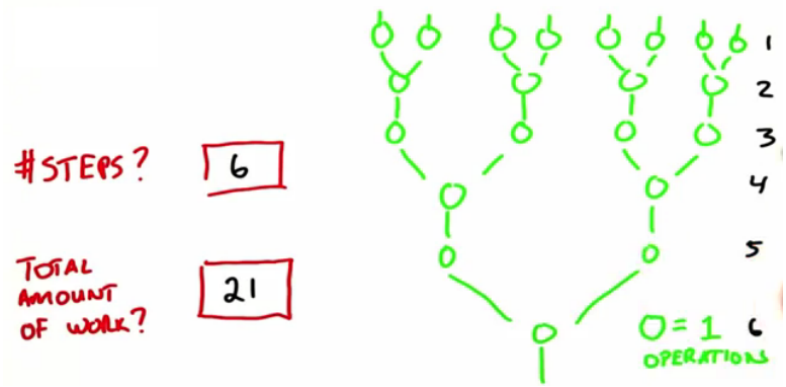


Figure 4.2: Calculating step and work size [13]

The important aspect is the workload to step ratio. The workload decides the total amount of work necessary for that specific algorithm whereas the step size determines

the amount of serial work in the algorithm. For an algorithm to have a parallel nature the step size has to be relatively low compared to the workload. This approach of analysing the properties of the computational steps in an algorithm is tightly knit with Amdahl's law for strong scaling. We describe the algorithms we have implemented in the algorithms section we will also describe the algorithm's workload and step size.

Basic principles of efficiency is the second most important aspect. Developing memory optimised kernels that utilize the cache efficiently is critical to reduce memory bottlenecks. In section 4.4 we will provide tricks to optimize the memory transfers and utilize faster memory.

Architectural specific optimisation concerns utilizing the given architecture of a specific GPGPU's SM such as amount of threads per SM, L1 cache size for shared memoryetc. In section 4.5 we will consider how to utilize the GPGPU architecture when setting kernel parameters.

Micro level optimisation works at the bit-level, such as approximating the inverse of a square root with magic numbers. We will not consider these types of optimisations on the GPGPU in this report as the gains are minimal compared with the other optimisations. [13]

4.3 Software Optimisations

This section presents software optimisations.

4.3.1 Workload and step size

As presented in the previous section the workload to step ratio is of importance when constructing a parallel algorithm. The step size compared with the total workload thus gives an idea of the amount of non-paralelisable content in the algorithm.

It is beneficial to both reduce step size (increasing parallelisability) and reduce workload (total amount of work). However, these goals are often not aligned as we will show with the two scan algorithms: Hillis and Steele's inclusive scan and Blelloch's exclusive scan. Our implementation of Hillis and Steele is presented in section 5.4. Table 4.3 presents the work and step for these two algorithms. However, in Blelloch there is a hidden multiple constant of two in the step size.

Hillis and Steele		Blelloch	
work	step	work	step
$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$

Table 4.3: Work and step for Hillis and Steele, and Blelloch

Even though the workload is larger for the Hillis and Steele scan if the amount of work at any given step in the algorithm is smaller than the total amount of available

processors then the Hillis and Steele algorithm will finish faster. However, if the workload is significantly larger than the amount of available processors and the step size is not the bottleneck, then the Blelloch scan will be superior.

4.3.2 Avoiding atomic operations

In section 3.3 we introduced atomic operations to handle race conditions amongst threads. However, using atomic operations serializes the access to the memory cell containing the bin which in the worst case could cause a step size equivalent of the workload. In an attempt to optimise the histogram implementation we aim to minimise the usage of atomic operations, which we present in section 6.2.

4.4 Memory Optimisations

This section presents relevant memory optimisations. We present coalesced memory access, followed by shared memory optimisations, and finally we briefly introduce bank conflicts.

4.4.1 Coalesced Memory Access

In section 2.3 we introduced the L1 cache, which is designed for spatial locality. Spatial locality promotes the use of coalesced memory access by the individual threads. Coalesced memory access means to have threads access successive memory locations. Since querying the global memory for a data value will often cause a memory transfer larger than what the thread requires and place the data in the faster cache on the SM. When the subsequent thread queries for the successive value it will be returned from the L1 cache instead of the global memory, giving a substantial speed up.

In order to utilize coalesced the memory layout the data needs to be in a structure that promotes data to be successive rather than strided. [13] This leads to two different approaches for storing data, namely array of structures (AoS) and structures of arrays (SoA), exemplified with RGB images in listing 4.1.

```
1 struct {
2   int R, G, B;
3 } AoS[N];
4
5 struct {
6   int R[N], G[N], B[N];
7 } SoA;
```

Listing 4.1: Example of SoA and AoS with RGB images

The AoS lays out the images in the following manner

```
R[1], G[1], B[1], R[2] G[2], B[2] ... R[N], G[N], B[N]
```

whereas the SoA would lay out the images in the following manner

```
R[1], R[2], ... R[N], G[1], G[2], ... G[N], B[1], B[2], ... B[N]
```

The SoA thus manages to lay out the data in a coalesced fashion, whereas the AoS lays out the data in a strided fashion.

4.4.2 Shared Memory

In section 3.1 we introduced blocks as having some amount of shared memory available. Shared memory is a defined space of memory in the SM's L1 cache that a set of threads within a block can read/write to during block execution. The memory will disappear when the block terminates. In this report we will consider two types of instances where shared memory could give a beneficial speed up.

The first is when a set of threads have to perform several read/write operations to a fixed set of global memory, which we as use in section 5.3. By mapping the global memory to shared memory, computing the set of operations in shared memory and then writing back to global memory allows us utilize the fast access to shared memory.

The second is to coalesce writes by collecting them in shared memory. This type of coalescing is especially interesting in the transpose problem where data is read row-wise but written column-wise. In section 6.1 will introduce how we used tiles in shared memory to allow coalesced writing.

4.4.3 Bank conflicts

The local memory is divided into memory banks. Transactions within these memory banks can only be accessed in a serial manner. Thus, if the memory bank is busy and a warp requests some data the data retrieval must wait for its turn. This is called a bank conflict. Such conflicts can be avoided if the memory access are successive and do not cross other warp's data requests there will be no bank conflict. [3]

4.5 Hardware Optimisations

This section presents relevant hardware optimisations. First we test how different warp sizes affect idle threads, followed by a test of the bus width.

4.5.1 Testing Warp Sizes

In section 2.2 we introduced the concept of a warp, the smallest amount of threads that can be launched at one time within a kernel. Table 4.4 shows the average elapsed time over 1,000 runs for launching different threads, to see the impact compared to the amount of warps needed. One warp launches 32 threads at a time, asserted from appendix A.

We present the multiple of 32 as the base case for the two tables. By using block size between 32×8 and 32×9 the SM still launched the same amount of warps (9 warps per block), but with idle threads in the 9'th warp as it must have 32 threads. As the

threads (#)	time (ms)	latency
32×10	5.321	
$32 \times 10 - 1$	5.336	+0.28%
$32 \times 10 - 16$	5.600	+5.24%
$32 \times 10 - 31$	5.892	+10.73%

threads (#)	time (ms)	latency
32×9	5.561	
$32 \times 9 - 1$	5.585	+0.43%
$32 \times 9 - 16$	5.892	+5.95%
$32 \times 9 - 31$	6.236	+12.13%

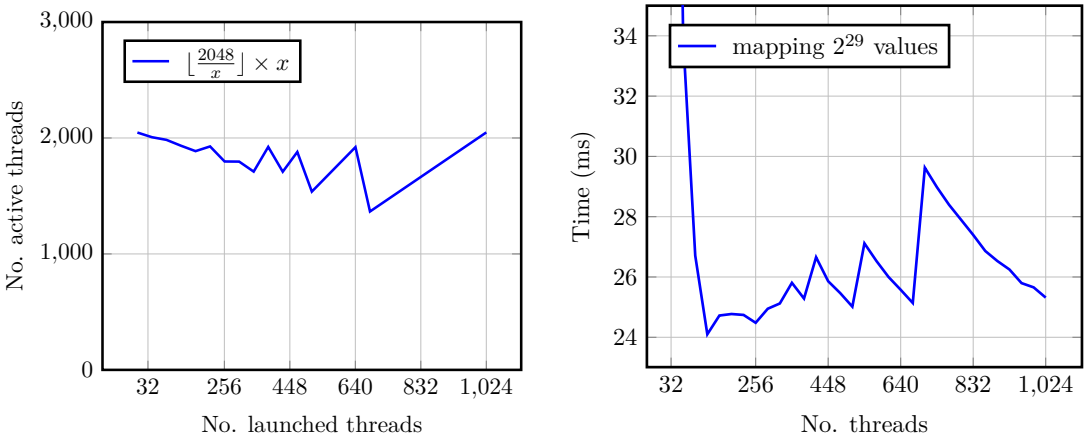
Table 4.4: Testing different warp sizes

amount of threads per block is reduced, the total amount of blocks necessary to complete the job must increase and thus increases runtime. table 4.4 shows how the performance deteriorates as the amount of idle threads in the last warp increases.

4.5.2 Testing BUS width

Figure 4.5(b) shows the average elapsed time over 10 runs for launching different sized blocks that were a multiple of the warp size. We did this to see how the SM's maximum thread bottleneck impacted the the amount of threads in a block. The SM can handle 2048 threads concurrently, asserted from appendix A. The number of active threads is illustrated in fig. 4.5(a) as a function of $\lfloor \frac{2048}{x} \rfloor \times x$, where x is the amount of launched threads.

As visable in the graph it has a "wave" function. Further is a function of the total amount of threads the SM can execute on. There is a clear correlation, which indicates that choosing a block size that maximizes the amount of threads able to concurrently execute is essential to achieving the best performance.



((a)) Active threads vs. launched threads

((b)) Mapping 2^{29} values

Figure 4.5: Threads and warps

CHAPTER 5

Algorithms

In the algorithms chapter we introduce four fundamental algorithms to that are often used in various combinations to model problems in parallel. [13] We will be introducing map (section 5.2), reduce (section 5.3), scan (section 5.4) and histogram (section 5.5). Furthermore, we present the compact algorithm (section 5.6), and radix sort (section 5.7), which uses these basic algorithms to solve more complex problems.

Moreover, in the first three sections we present plots to illustrate the runtime's development while we expand the array size of the input from 2^0 to 2^{29}. We performed these tests by running each array size 10 times and averaging the runtime results. So, we ran 30×10 total runs for each test.

5.1 Debugging

To debug we used two approaches; `printf()` for testing the logic of our application and `cuda-memcheck` to catch CUDA memory errors. The `cuda-memcheck` is a command-line tool that informs the user about CUDA errors during runtime. It can be used to find issues with memory access, thread ordering, race conditions and hardware reported program errors. It is invoked as follows

```
cuda-memcheck [options] application-name [application-options]
```

As with the `nvprof` from section 4.1 the `cuda-memcheck` tool has a variety of flags for the option input. In the option input `memcheck` is sat by default. We further tested the option flag `racecheck`. The `racecheck` allows us to detect write-after-write hazards, where two or more threads attempt to update the same memory location simultaneously.

5.2 Map

As introduced in section 3.3 we present a serial execution of the mapping operation and a parallel implementation of the mapping operation. The serial code for a mapping operation is presented in listing 5.1. The code loops through each index of the input and copies the value to the same index in the output. Given $n =$ `ARRAY_SIZE` this algorithm will have a workload of $\mathcal{O}(n)$, and step size of $\mathcal{O}(n)$.

```
1 void map(int *h_in, int *h_out, int SIZE) {
2   for (int j = 0; j < SIZE; j++)
```

```
3     h_out[j] = h_in[j];
4 }
```

Listing 5.1: Serial map

As we can analyse the problem as a mapping operation, as introduced in section 3.3, we know that every mapping operation is independent from one another why the amount of steps can be reduced to $\mathcal{O}(1)$ and the problem thus becomes 100% parallelisible. Given Amdahl's law we should be able to achieve a speed up in the amount of cores, which is 2880 according to appendix A. We present a simple kernel that perform such a map operation in listing 5.2.

```
1 __global__
2 void map_kernel(int *d_out, int *d_in, int SIZE) {
3   int mid = threadIdx.x + blockDim.x * blockIdx.x;
4   if (mid >= SIZE) return;
5   d_out[mid] = d_in[mid];
6 }
```

Listing 5.2: Map kernel

First the kernel calculates the index of thread as a product of its position in the block and grid. Next, the kernel validates that the index is inside the bounds of the arrays it is operating on. Finally, the input is copied to the new output.

We present the comparison of the map algorithms in fig. 5.2. It is clear that the parallel mapping algorithm has an advantage. All the operations can be performed in parallel, while the CPU has to perform every operation as consecutive operations.

In table 5.1 we list the performance at $n = 2^{29}$. However, the result is nowhere near the prediction of Amdahl's law, which said that we would get a 2880x speed up. The reason is that the mapping operation is an operation with very low computational complexity and many expensive transfers to and from global memory. Thus, the global memory transactions are the bottleneck.

device	algorithm	time in ms	speed up	bandwidth usage	usage percentage
CPU	serial	1200.73			
GPGPU	parallel	25.32	x47.42	169.63 GB/s	58.9%

Table 5.1: Global vs. Shared memory read and writes

At $n = 2^{29}$ it takes the CPU $1200.73ms$ while the GPGPU computes the mapping in 25.32 being 47.42 times faster.

5.3 Reduce

The next algorithm is the reduce algorithm. The reduce algorithm is a gather operation that maps a space down to a subspace, e.g. performing an addition on an array of

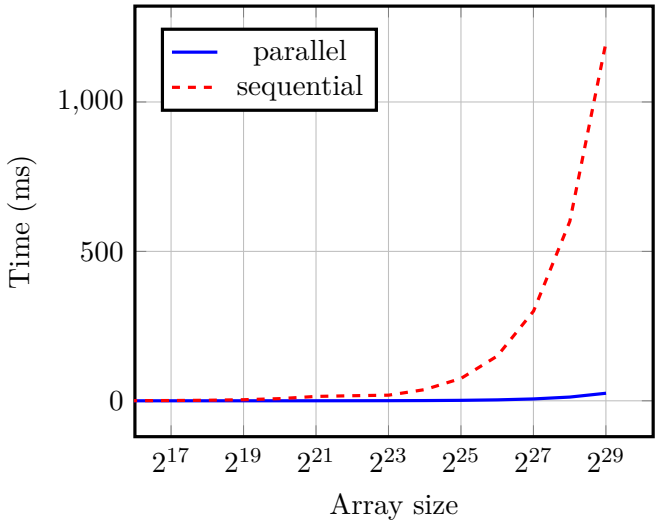


Figure 5.2: Runtime development of the map algorithms

numbers. The algorithm for performing parallel gathers are restricted to associative operations such that `(a op b) op c = a op (b op c)`.

Listing 5.3 shows such an operation in a serial fashion. The operation loops through all input elements and sums the result. The operation can be expressed such as, given $n =$ `ARRAY_SIZE`, workload of $\mathcal{O}(n)$ and step size of $\mathcal{O}(n)$.

```
1 void reduce(int *h_in, int h_out, int SIZE) {
2   for (int l = 0; l < SIZE; ++l)
3     h_out += h_in[l];
4 }
```

Listing 5.3: Serial reduce

The challenge with the reduce code is that we perform all operations on a single memory address. If the serial code is naively made parallel we would be faced with race conditions. Instead we can look at the associative property of out problem, such that

$$((\mathtt{a} + \mathtt{b}) + \mathtt{c}) + \mathtt{d} = \mathtt{A} + \mathtt{B},$$

where `A=a+b` and `B=c+d`. By utilizing this property we can perform the computation of `A` and `B` in parallel and thus halve the problem by allowing each thread to compute two numbers to one. Reducing the problem size by a factor of two at each step gives a step size of $\mathcal{O}(\log_2 n)$ and allows us to compute parts of the problem in parallel. We present an illustration in fig. 5.3 to try to give an intuition of how the parallel code will run. [6]

The idea is that each block in the image is performed by each kernel invocation. Listing 5.4 shows the reduce kernel. The needed indices are calculated, followed by the block's reduction in the loop. Before proceeding the code makes sure that all threads have finished. Finally, only the first thread writes the block's reduction to the output

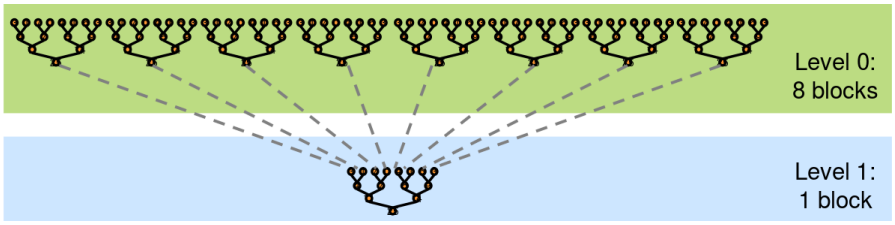


Figure 5.3: Reducing over multiple sweeps

array. The output is written to the block's index in the grid, which makes sure that each block's result is written to independent addresses.

```
__global__
void reduce_kernel(int *d_out, int *d_in, int SIZE) {
  int mid = threadIdx.x + blockIdx.x * blockDim.x,
      tid = threadIdx.x;

  for (int s = blockDim.x / 2; s > 0; s /= 2) {
    if ((tid < s) && ((mid + s) < SIZE))
      d_in[mid] += d_in[mid + s];
    __syncthreads();
  }

  if ((tid == 0) && (mid < SIZE))
    d_out[blockIdx.x] = d_in[mid];
}
```

Listing 5.4: Reduce kernel

The intermediate results are thus transferred to the next iteration of kernel invocations. We add a wrapper function, in listing 5.5, that runs these kernels iteratively. We introduce temporary arrays to hold the intermediate results, and update the grid size number of elements accordingly before invoking the kernel again. Finally, the code performs a final invocation of the kernel with the desired final output array to combine the results.

```
do {
  reduce_kernel<<<grid_size, BLOCK_SIZE>>>(d_tmp_out, d_tmp_in, size);
  // Updating intermediate arrays
  size  = grid_size;
  bytes = sizeof(int) * size;
  cudaMemcpy(d_tmp_in, d_tmp_out, bytes, cudaMemcpyDeviceToDevice);
  // Updating to reflect how many blocks we now want to compute on
  grid_size = size / BLOCK_SIZE + 1;
} while(size > BLOCK_SIZE);
// Computing remainder
reduce_kernel<<<1, size>>>(d_out, d_tmp_out, size);
```

Listing 5.5: The loop in the wrapper for the reduce kernel

5.3.1 Using Shared Memory

As the kernel relies on computation of a subtree it performs several read/writes to global memory. To reduce the amount of expensive transfers from global memory we propose

the usage of shared memory instead. If we have 1024 threads per block then we will do more than 3000 read and write operations to global memory. However, if we read the needed memory into shared memory and perform the loop on that memory, then we will do about 1000 read and write operations. Thus, we can achieve a theoretical 3x speed up. The number of read and writes are presented in table 5.4.

global memory	read	1024	512	256	···	1
	write	512	256	128	···	1
shared memory	read	1024				
	write	1				

Table 5.4: Global vs. Shared memory read and writes

We present the revised reduce kernel with shared memory in listing 5.6. We add the `sdata` array to contain the shared memory, and each thread copies the global data to that shared memory.

```
__global__
void reduce_kernel(int *d_out, int *d_in, int SIZE) {
  int mid = threadIdx.x + blockIdx.x * blockDim.x,
      tid = threadIdx.x;
  if (mid >= SIZE) return;

  extern __shared__ int sdata[]; // allocate shared memory
  sdata[tid] = d_in[mid];        // each thread loads global to shared memory
  __syncthreads();               // make sure all threads are done

  for (int s = blockDim.x / 2; s > 0; s /= 2) {
    if ((tid < s) && ((mid + s) < SIZE))
      sdata[tid] += sdata[tid + s]; // perform operations on shared memory
    __syncthreads();
  }

  if ((tid == 0) && (mid < SIZE))
    d_out[blockIdx.x] = sdata[0];   // copy shared back to global memory
}
```

Listing 5.6: Reduce kernel using shared memory

```
unsigned int SMEM = BLOCK_SIZE * sizeof(int);
reduce_kernel<<<grid_size, BLOCK_SIZE, SMEM>>>(d_tmp_out, d_tmp_in, size);
```

Listing 5.7: Updated call to reduce kernel after use of shared memory

Furthermore, we must update the kernel call to include the amount of shared memory to be allocated. This is illustrated in listing 5.7, where the third argument in the triple chevrons is the amount of shared memory to allocate.

In fig. 5.5 we present a graph of the runtime for the reduce algorithms. The GPGPU gets an advantage when the array size to sum is large. This is most likely due to overhead and memory transfers. The GPGPU is superior to the CPU when it has enough work to spread out to its many SMs.

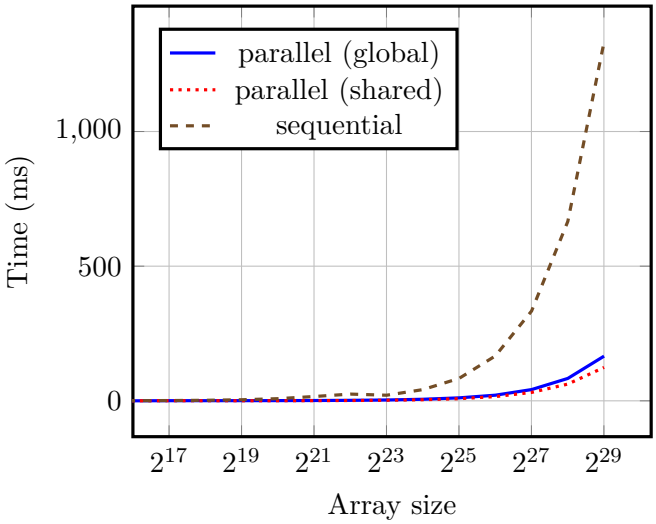


Figure 5.5: Runtime development of three reduce algorithms

Using Amdahl's law we can conclude that this problem is not as parallelisable as the mapping function from the previous section and should thus achieve a lesser speed up. Table 5.6 present the performance with $n = 2^{29}$. The CPU performs the reduce in $1332.38ms$ where the GPGPU performs the global reduce in 165.92, a 8.03x speed up, and shared reduce in 124.07, 10.74x speed up.

device	algorithm	time in ms	speed-up	bandwidth usage	usage percentage
CPU	serial	1332.38			
GPGPU	global	165.92	x8.03	38.83 GB/s	13.46%
	shared	124.07	x10.74	34.62 GB/s	6.00%

Table 5.6: Global vs. Shared memory read and writes

Reads in the global memory model are twice the total array amount as it also reads/writes for every step in the combining tree. As the size of the tree and amount of work is logarithmic with respect to the input the total amount of reads and writes are twice the input size. Reads/writes in the shared memory model does not follow the same logarithmic reduction in global memory reads/writes as it only communicates with global memory at block termination and thus only performs one write. As we used a block size of 1024 in the experiments we will not consider the additional transfers at the combining tree layers beyond the initial input array.

We mentioned that the shared memory version of the reduce would theoretically get approximately a 3x speed up compared to the regular version. This is not the case as illustrated in the plot and by the speed ups. This is due to the fact that we do not utilize the full potential of the local memory of the device. [13]

5.4 Scan

The third basic algorithm is the scan algorithm. The scan algorithm is able to produce the cumulative operation of all previous array ellements given location. Either exclusive or inclusive such as illustrated in table 5.7. To compute the scan the algorithm takes an array, a binary and associative operator, and an identity element. If the operator is add, the scan algorithm computes the running sum of the input. The identity element is 0 for addition as shown in the ouput (excl.) from table 5.7.

input	1	2	3	4
operator			+	
output (incl.)	1	3	6	10
output (excl.)	0	1	3	6

Table 5.7: Sum scan example

A serial implementation of an inclusive sum scan is presented in listing 5.8. The serial implementation follows a dynamic programming scheme where it utilizes the previously computed results to generate the next element in the output.

If the scan was to be exclusive the inclusive result would need to be shifted by one and have its first element as the identity element.

```
1 void scan(int *h_int, int *h_out, int SIZE) {
2   h_out[0] = h_in[0];
3   for (int l = 1; l < SIZE; ++l)
4     h_out[l] = h_out[l-1] + h_in[l];
5 }
```

Listing 5.8: Serial scan

5.4.1 Hillis and Steele Inclusive Scan

In this section we have provided an implementation and analysis of the Hillis and Steele inclusive scan. The algorithm is illustrated in fig. 5.8.

The algorithm utilizes the same type of dynammic programming scheme from the serial implementation though it is not as efficient.

The algorithm uses `steps` (not to be confused with the steps used in the algorithmic analysis) starting with `step=0`. On the ith `step`, each element adds itself to the neighbour 2^i to the left. Given an array where $n =$ `ARRAY_SIZE` and a reduction of 2^i at every level of the algorithm the tree gets a height of $\log_2 n$ and a workload of $\mathcal{O}(n \log_2 n)$. Thus, the step size is $\mathcal{O}(\log_2 n)$.

As of implementation details we need synchronization between each iteration of $\mathcal{O}(\log_2 n)$ scan tree. To implement this synchronization we have the kernel only performing a single operation per element in the array. To handle the height of the array we use a kernel wrapper to secure synchronization between blocks.

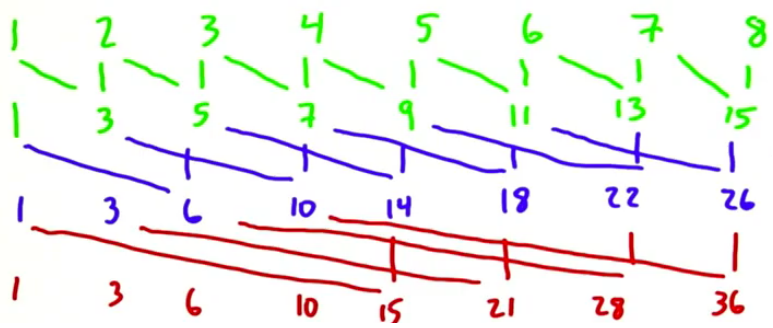


Figure 5.8: Hillis and Steele inclusive sum scan illustration

We present the kernel for this implementation in listing 5.9. Each thread calculates its position in the grid and block, and returns if it is out of bounds. Otherwise, we make sure to add the correct element in the input or 0 if it is out of bounds.

```
__global__
void scan_kernel(int *d_out, int *d_in, int step, int SIZE) {
  int mid = threadIdx.x + blockDim.x * blockIdx.x;
  if (mid >= SIZE) return;

  int toAdd  = ( ((mid - step) < 0) ? 0 : d_in[mid - step] );
  d_out[mid] = d_in[mid] + toAdd;
}
```

Listing 5.9: Hillis and Steele scan kernel

The wrapper is shown in listing 5.10. The wrapper secures synchronization between the different steps in the algorithm and controls the steps to be taken by the kernels by doubling the step size after each layer. We use a temporary variable holder so we do not store values in the input array.

```
void scan_kernel_wrapper(int *d_out, const int *d_in, int SIZE,
                         unsigned int BYTES, int BLOCK_SIZE) {
  for (int step = 1; step < SIZE; step *= 2) {
    scan_kernel<<<GRID_SIZE,BLOCK_SIZE>>>(d_out, d_tmp, step, SIZE);
    cudaMemcpy(d_tmp, d_out, BYTES, cudaMemcpyDeviceToDevice);
  }
}
```

Listing 5.10: Hillis and Steele scan kernel wrapper

device	algorithm	time in ms	speed-up	bandwidth usage	usage percentage
CPU	serial	4.42			
GPGPU	Hillis & Steele	1.57	x2.81	72.13 GB/s	25.01%

Table 5.9: Serial vs. Hillis and Steele

Figure 5.10 presents the scan algorithm's runtime development. Noticeable is that the Hillis and Steele scan does not perform significantly better than the serial at high values.

As the algorithm grows in logarithmic complexity the workload become approximately 28 larger at $n = 2^{29}$.

Given the higher asymptotic runtime we cannot assume that the Hillis and Steele algorithm will scale efficiently even though it is parallelisable. However, at smaller problems we found that the Hillis and Steele algorithm performed superior to the serial algorithm such as shown with $n = 2^{19}$ in table 5.9. We further elaborated the asymptotic relationship of the Hillis and Steele scan in section 4.3.1.

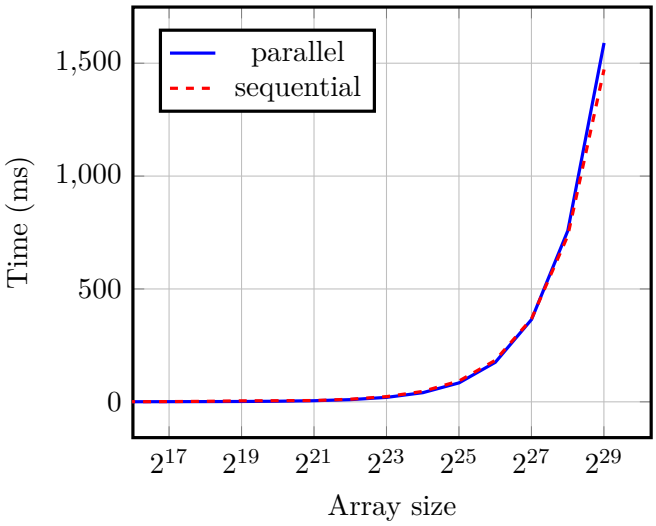


Figure 5.10: Runtime development of the scan algorithms

5.5 Histogram

The fourth and last basic algorithm is the histogram algorithm. The histogram algorithm maps a series of elements given a function to an index in a histogram with $k = \|\texttt{histogram}\|$ indexes, also known as the histogram's bins. When an element is mapped to the index the given value at the given index is incremented by one.

The mapping function in our implementation is the modulus algorithm such that e mod k with $e \in$ `input` with the `input` $\subset \mathbb{N}, 0 \in \mathbb{N}$ monotonically increasing, $[0, 1, 2, \ldots, n] \subset$ `input`. Such that the input maps uniformly to the bins of the histogram.

A serial example is presented in listing 5.11 where the input is `h_in` and `h_out` is the histogram. We will have k bins in the output. As the serial algorithm is a for loop going over every element in the input array we conclude that the workload is $\mathcal{O}(n)$ and step size $\mathcal{O}(n)$.

```
1 void histogram(int *h_in, int *h_out, int IN_SIZE, int k) {
2   for (int l = 0; l < IN_SIZE; l++)
3     h_out[(h_in[l] % k)]++;
4 }
```

Listing 5.11: Serial histogram

In section 3.3 we presented a naive implementation of the histogram kernel. To avoid race conditions we use the `atomicAdd()` function to perform an atomic update on the given bin. We present the parallel implementation in listing 5.12 with the input is `d_in` and `d_out` is the histogram.

```
__global__
void histo_kernel(int *d_bins, int *d_in, int BIN_SIZE, int SIZE) {
  int mid = threadIdx.x + blockDim.x * blockIdx.x;
  if (mid >= SIZE) return; // checking for out-of-bounds

  int bin = d_in[mid] % BIN_SIZE;
  atomicAdd(&(d_bins[bin]), 1);
}
```

Listing 5.12: Simple parallel histogram implementation

The test performed in this histogram is slightly different from the test performed in on the other basic algorithms. In this test we found that the interesting part of the histogram algorithm with atomic add is how the number of steps, the serial part, of the problem changes with the size of the histogram increasing given a monotonically increasing input. In such case we analyse the number of steps to be $\mathcal{O}(n/k)$, which is visible from our graph in fig. 5.11.

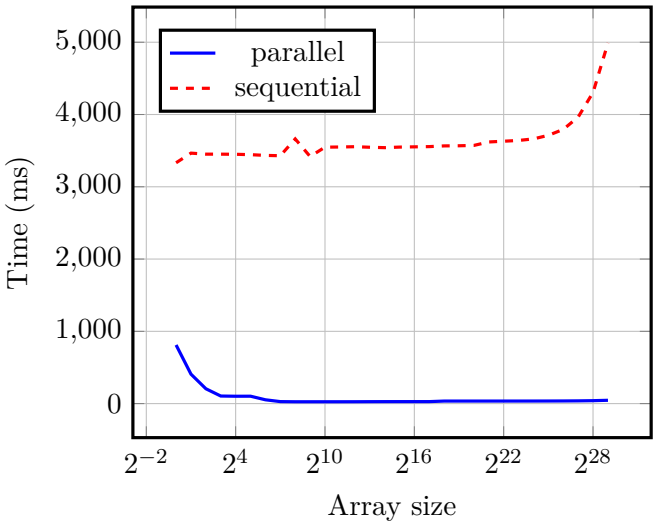


Figure 5.11: Histogram algorithm

5.6 Compact

This section presents the algorithm compact (also known as filter). The idea is to take some input and only return the items that obey some predicate, e.g. whether or not the

least significant bit (LSB) is 0. [13] The algorithm can be divided into three steps

1. Calculate predicate array
2. Calculate scatter addresses from predicate array (using scan from section 5.4)
3. Map desired items to output, given the scatter addresses (using map from section 5.2)

Listing 5.13 presents the predicate computation computation, of whether or not the LSB is 0, and saves the result to the `predicate` array.

```
predicate[idx] = (int)((input[idx] & 1) == 0);
```

Listing 5.13: LSB equal to 0 – save items' result to predicate array.

The next step is to calculate the scatter addresses from the predicate array, i.e. the addresses to which the elements in the input array must be mapped to. This can be computed with an exclusive sum scan over the `predicate` array. Table 5.12 illustrates a trivial example. The top row presents the input indices, the next row the elements in the respective index, next the LSB predicate for that item and lastly the scatter address based on the predicates.

`idx`	0	1	2	3	4	5
input`[idx]`	4	5	6	7	8	9
predicate`[idx]`	1	0	1	0	1	0
scatter address	0	1	1	2	2	2

Table 5.12: Predicate and scatter address output given the input

From the scatter addresses the elements with `predicate=1` are moved to the scatter address in the output array index of the bottom row. The contents of that array is thus the values where the LSB is 0. This array will be of length 3 as the reduction of the predicate array yields the value 3. The resulting output array would equal `[4, 6, 8]`.

5.7 Radix Sort

This section presents a parallelised algorithm of the radix sort benchmarked with a serial sort implementation from the `C++` library. The implementation of the Radix Sort algorithm can be found in appendix B. Radix Sort is a bit-wise comparison algorithm that iteratively, in the length of the bits of the integer, looks at the least significant bit (LSB) and sorts the values accordingly. [13]

In order to parallelise the sorting of the LSB in the radix sort we used the compact(section 5.6). Thus to parallelise the entire sorting we perform multiple compact operations as we move the elements in the array according to their LSB. In this section we outline a single iteration of radix sort in this section.

To sort the elements according to their LSB we perform two compact operations. The first compact operation maps the elements with LSB = 0 from the output array's zero index and up to

$$\texttt{zeros\_end\_idx} = \{|x| - 1 \;\Big|\; (x\&1) = 0, x \in \text{input}\},$$

as illustrated in fig. 5.13. The second compact operation maps the elements with LSB = 1 from the output arraysindex `zeros_end_idx` + 1 to index

$$\{|x| - 1\}$$

being the end position.

For each iteration of Radix Sort the LSB will shift left and compact sorting thus based on increasing importance of the bit representations in the element. Consider for instance the following

$$\text{iteration0} : 000000000000000[0]$$

$$\text{iteration1} : 00000000000000[0]0$$

$$\ldots$$

$$\text{iteration31} : [0]000000000000000$$

Figure 5.13: Example of how each digit is moved according to LSB

CHAPTER 6

Building a Fast Histogram Algorithm

In section 5.5 we present our parallel implementation of the histogram algorithm. The aim of the following sections is to present how we tried to boost the performance of the algorithm's running time. Before we introduce the histogram implementation we introduce the transpose algorithm. The next section present and describes what we did to try to boost the performance. The following section presents what further thoughts we had that might have given better performance.

6.1 Transpose

We present three algorithms for optimizing the transpose matrix operation in this section.

The serial code for a transpose operation is presented in listing 6.1. The code loops through each index of the input, `h_in[i,j]`, and copies the value to the transposed index in the output, `h_out[j,i]`. Given n = `width`, m = `height` this algorithm will have a workload of $\mathcal{O}(nm)$ and step size of $\mathcal{O}(nm)$.

```
1 void map(int *h_in, int *h_out, const int ROWS, const int COLUMNS) {
2 for(int row=0; row < ROWS; row++)
3   for(int column=0; column < COLUMNS; column++)
4     out[column + row*COLUMNS] = in[row + column*ROWS];
5 }
```

Listing 6.1: Serial transpose

As we can analyse the problem as a mapping operation, which we introduced in section 3.3, we know that mapping operations are independent from one another why the amount of steps can be reduced to $\mathcal{O}(1)$ and the problem thus becomes 100% parallelisible. In listing 6.2 we present a trivial elem-wise parallel implementation of the transpose algorithm.

```
1 __global__
2 void transpose_kernel(int * d_out, int * d_in,
3                       const int ROWS, const int COLUMNS){
4   int row = threadIdx.x + blockIdx.x * blockDim.x;
5   int column = threadIdx.y + blockIdx.y * blockDim.y;
6   if((row >= ROWS) || (column >= COLUMNS)) return;
7   d_out[column + row*COLUMNS] = d_in[row + column*ROWS];
8 }
```

Listing 6.2: Parallel transpose

The dilemma with the trivial element-wise transpose presented in listing 6.2 is that the kernel will read row-wise but write column-wise. The row-wise read will utilize coalesced reading whereas the writes will not be coalesced resulting in only one write per bus, wasting $48 - 4 = 44$ bytes. To utilize coalesced writes we present the tiled transpose exemplified in fig. 6.1 and implemented in listing 6.3. The tiled transpose uses shared memory to make coalesced writes to global memory, such as introduced in section 4.4.2.

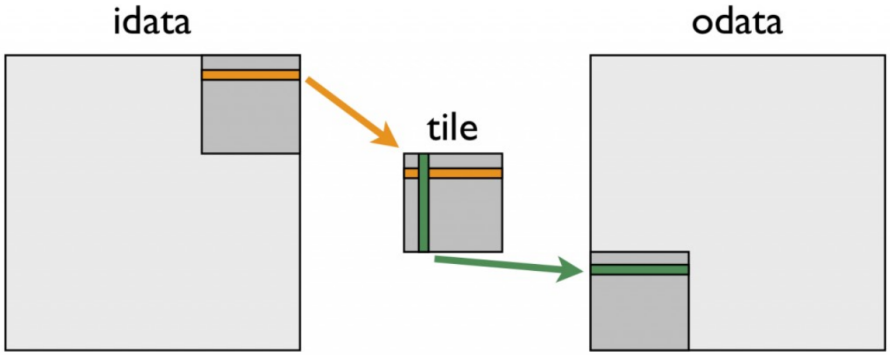


Figure 6.1: Tiled transpose example

```
1  __global__
2  void transpose_kernel_tiled(int * d_out, int * d_in,
3                              const int ROWS, const int COLUMNS){
4    __shared__ int tile[DIM][DIM];
5    int x = threadIdx.x + blockIdx.x * blockDim.x,
6        y = threadIdx.y + blockIdx.y * blockDim.y;
7    if((x >= COLUMNS) || (y >= ROWS)) return;
8    tile[threadIdx.y][threadIdx.x] = d_in[x + y*COLUMNS];
9    __syncthreads();
10   x = threadIdx.x + blockIdx.y * blockDim.y;
11   y = threadIdx.y + blockIdx.x * blockDim.x;
12   d_out[x + y*ROWS] = tile[threadIdx.x][threadIdx.y];
13 }
```

Listing 6.3: Tiled transpose implementation

The tiled solution, however, has memory bank conflicts such as described in section 4.4.3. To avert bank conflicts we convert line `4` to `tile[DIM][DIM+1]`. The speed-ups are presented in table 6.2 for a problem of size $n = 2^{14}$, $m = 2^{14}$.

It is to be noted that we could not get the tiled and tiled+bank to run a successful transpose if the input matrix was not a square and if the `BLOCK_SIZE` was not able to cover the matrix without overlapping blocks (e.g. threads out of bounds). We tried using online communities for this issue as described in section 1.2, but unfortunately did not get any responses.

device	algorithm	time in ms	speed-up	bandwidth usage	usage percentage
CPU	serial	6288.88			
	elem-wise	33.61	x187.11	63.89 GB/s	22.16%
GPGPU	tiled	24.93	x252.26	86.14 GB/s	29.87%
	tiled+bank	16.99	x370.15	126.40 GB/s	43.83%

Table 6.2: Global vs. Shared memory read and writes

6.2 Histogram with Coarse Bins

In this section we presents a parallel algorithm for histograms that does not suffer from the same serial bottlenecks as the atomic version from section 5.5.

The idea is to allow each block to compute their own histogram with atomic add and then use reduce to merge these histograms, which reduces steps from $\mathcal{O}(n)$ to $\mathcal{O}(\text{block\_size})$.

As having a personal histogram might be memory intense giving a large amount of bins. To avoid such we split the bins into coarse bins containing a range of bins. Each block thus works on a coarse bin instead of the total histogram. However, performing such coarse bin introduces a sorting of the input values according to their coarse bins.

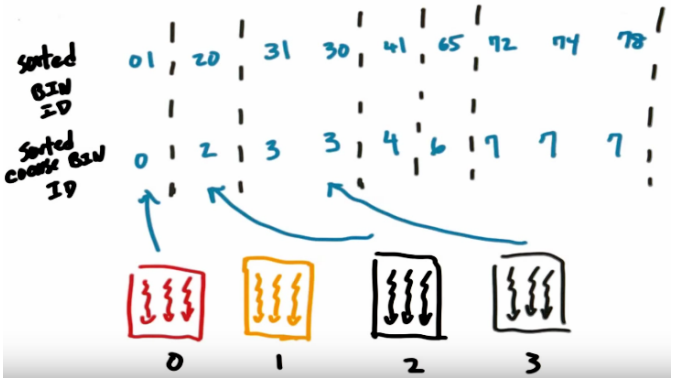


Figure 6.3: Intuition behind coarse bin and kernel invocations

The idea is illustrated in fig. 6.3, where we have the sorted bin values at the top and the sorted coarse bin values at the bottom.

For each coarse bin histogram a `grid_size` $\times$ the number of elements in a course bin, with `grid_size` the number of blocks necessary to collect all elements in the given coarse bin, is allocated as the `coarse_bin_grid`. For each coarse bin a kernel is launched containing threads to gather all the elements in the coarse bin. Each block in this kernel has its own private histogram in `coarse_bin_grid` in which only the given block performs updates. At end of kernel execution the `coarse_bin_grid` is left with bin values represented along the columns. In order to avoid cache misses and have coalesced access the transpose operation introduced in previous section is performed to have bin

values aligned in row-major order. Given all values for each bin being set in a row we perform reduce over the rows to gather the independent block's histograms for each bin leaving an array of bin values for each coarse bin. These are then merged resulting in the total histogram.

In our implementation we did not manage to create the full fast histogram solution as described above. We computed the coarse bin sorting and merely had each bin atomic add to the global memory without the private `coarse_bin_grid` to reduce over. From an algorithmic viewpoint this is no better than the atomic reduce histogram we were trying to improve, but it allowed us to test our coarse bin sorting. A more detailed description of the coarse bin implementation can be found below.

Given and input array, number of bins, `NUM_BINS`, and a number of coarse bins, `COARSE_SIZE`, we perform the following computation.

1. Compute bin for each value
2. Compute coarse bin for value
3. Sort values w.r.t. the computed coarse bins
4. Find starting position for each coarse bin
5. For each coarse bin — count each bin in given range

We compute the bin for each respective value as presented in listing 6.4, where `d_in` is the array of input values and `d_out` is a give inputs respective bin number.

```
d_out[mid] = d_in[mid] % NUM_BINS;
```

Listing 6.4: compute each value's bin

Then the respective bin numbers are divided into coarse bins as presented in listing 6.5, where the `d_in` is the array of bins, i.e. the `d_out` from listing 6.4.

```
d_out[mid] = d_in[mid] / COARSE_SIZE;
```

Listing 6.5: compute each value's coarse bin

This gives an array of values, their respective bins, and their coarse bins. The next step is to sort all the values w.r.t. the coarse bins in ascending order, i.e. coarse bin 0 is first, followed by coarse bin 1, etc. We use the radix sort implementation we developed earlier (with slight modifications to suit our needs) to perform the sorting based on the values in the list of coarse bins.

After the values have been sorted it is possible to find the starting positions of each coarse bin. This gives us a starting value for each coarse bin from which it is possible to calculate how many values fall into each coarse bin. We find the positions of the bins as presented in listing 6.6 by running through each value and checking if the value before it was the same value. If the previous value is not the same then we have a starting position for the next coarse bin.

```
if (d_in[mid] != d_in[mid-1])
  d_out[d_in[mid]] = mid;
```

Listing 6.6: find the start positions of each coarse bin

6.2.1 Profiling and Analysis

We tested and profiled our new implementation by using the `nvprof` command-line tools to get a file that could be imported into the nvidia visual profiler. This is desirable because nvidia visual profiler is able to run a guided analysis of the program and give us advice on bottlenecks to optimize. We computed an executable with `BIN_SIZE` = 100, $n = 2^{22}$ and `COARSE_SIZE` = 10, which we ran with

```
nvprof --analysis-metrics -o output-file.nvprof ./executable-to-test
```

which gave us a file that contained all the metrics needed for the guided analysis. This file was opened with the nvidia visual profiler and the program performed analysis of GPGPU usage. Figure 6.4 shows the output given by the nvidia visual profiler's initial GPU usage analysis. It informs that the algorithm has low kernel concurrency and low compute utilisation. The low compute utilisation is mainly due to the lack of heavy lifting in our algorithm as most of our operations are memory intensive rather than computationally intensive. As for low kernel concurrency our approach to solve the coarse bins is serial and not performed in parallel, which we could optimise upon by introducing streams.

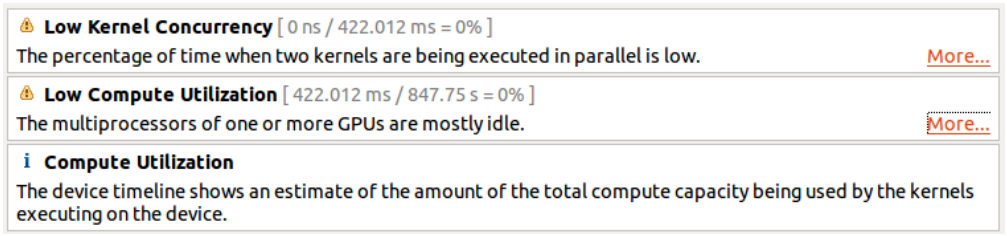


Figure 6.4: nvidia Visual Profiler analysis

6.2.2 Adding Streams

It is possible to make kernel executions concurrent. However, this only makes sense if the kernels are independent of each other. So, if one or more kernels are not dependent of the output from the other kernels they can be executed concurrently. In CUDA kernels are run concurrently with the use of streams.

Thus, from the visual profiler's analysis we tried to add streams to the kernels that were independent of the execution of the rest of the kernels. This were the kernels that were

in charge of counting and incrementing the final result of the histogram algorithm. This modification to the code is presented in listing 6.7. We instantiate an array of streams and for each iteration of the loop we create a stream and invoked the kernel with it. The full implementation is presented in appendix C.

```
1  // instantiate array of streams
2  cudaStream_t streams[COARSE_SIZE];
3
4  for (unsigned int i = 0; i < COARSE_SIZE; i++) {
5    // create the stream for each kernel call
6    cudaStreamCreate(&streams[i]);
7
8    // set up range for local coarse bin
9    local_bin_start = h_positions[i];
10   local_bin_end   = (i == COARSE_SIZE-1) ? NUM_ELEMS : h_positions[i+1];
11
12   // calculate local grid size
13   amount = local_bin_end - local_bin_start;
14   grid_size = amount / BLOCK_SIZE.x + 1;
15
16   // make sure there is at least one value in coarse bin
17   if (amount > 0)
18     coarse_histogram_count<<<grid_size, BLOCK_SIZE, 0, streams[i]>>>(
           d_histogram, d_bins, local_bin_start, local_bin_end);
19 }
```

Listing 6.7: Using streams to invoke kernels concurrently

The resulting output did not boost the performance significantly. We found that the sorting of the numbers used approximately 80% of the execution time. This means that the bottleneck was the sorting and not the histogram's atomic increments. The reason for the sorting being so slow could be due to using the Hillis & Steele scan for the compact in our radix sort implementation. If we used the Blelloch scan instead, as debated in section 4.3.1, we might have been able to reduce the time spent on sorting.

CHAPTER 7

Conclusion

In this project we described the fundamental difference between CPUs and GPGPUs. We described the hardware and software specifications of working on a GPGPU in the CUDA environment. We proposed challenges with optimising parallel code to run on a GPGPU. We implemented a series of trivial and more advanced algorithms such as sort and an optimised version of histogram. We used theory and debugging tools to analyse runtime and bottlenecks of the our implementations.

APPENDIX A

Tesla K40 Specifications

It is possible to fetch information about the installed GPUs by using a program called `deviceQuery`. To locate the program, we print the contents of the `$PATH` variable, and find the folder where CUDA was installed. We find the exact program by running

```
find /appl/cuda/6.5/ -type f -name "deviceQuery"
```

Listing A.1 presents the output given for our device, by running

```
/appl/cuda/6.5/samples/1_Utilities/deviceQuery/deviceQuery
```

```
Device 0: "Tesla K40c"
  CUDA Driver Version / Runtime Version          7.5 / 6.5
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 11520 MBytes (12079136768 bytes)
  (15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores
  GPU Clock rate:                                745 MHz (0.75 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D
      =(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           2 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
         simultaneously) >
```

Listing A.1: Information on the Tesla K40 GPU

APPENDIX B

Radix Sort Implementation

The Radix Sort code in listing B.1 is a cut down version of the full implementation. We have removed function prototypes and documentation comments so it takes up less space. The entire code base for this implementation is public at https://github.com/obeyed/dtu-hpc-course/tree/master/code/radix_sort. We use a helper function `checkCudaErrors()` to check if response is not `cudaSuccess` – it is defined in `utils.h`.

Listing B.1: Radix Sort implementation

```
// Populates array with 1/0 depending on Least Significant Bit is set.
__global__
void predicate_kernel(unsigned int* const d_predicate,
                      const unsigned int* const d_val_src,
                      const size_t NUM_ELEMS,
                      const unsigned int i) {
  const unsigned int mid = threadIdx.x + blockIdx.x * blockDim.x;
  if (mid >= NUM_ELEMS) return;

  d_predicate[mid] = (int)(((d_val_src[mid] & (1 << i)) >> i) == 0);
}

// Performs one iteration of Hillis and Steele scan.
__global__
void inclusive_sum_scan_kernel(unsigned int* const d_out,
                               const unsigned int* const d_in,
                               const int step,
                               const size_t NUM_ELEMS) {
  const int mid = threadIdx.x + blockIdx.x * blockDim.x;
  if (mid >= NUM_ELEMS) return;
  int toAdd = (((mid - step) < 0) ? 0 : d_in[mid - step]);
  d_out[mid] = d_in[mid] + toAdd;
}

// Shifts all elements to the right. Sets first index to 0.
__global__
void right_shift_array_kernel(unsigned int* const d_out,
                      const unsigned int* const d_in,
                      const size_t NUM_ELEMS) {
  const unsigned int mid = threadIdx.x + blockIdx.x * blockDim.x;
  if (mid >= NUM_ELEMS) return;
  d_out[mid] = (mid == 0) ? 0 : d_in[mid - 1];
}

// Toggle array with values 1 and 0.
__global__
void toggle_predicate_kernel(unsigned int* const d_out,
                             const unsigned int* const d_predicate,
                             const size_t NUM_ELEMS) {
  const unsigned int mid = threadIdx.x + blockIdx.x * blockDim.x;
  if (mid >= NUM_ELEMS) return;
```

```
42   d_out[mid] = ((d_predicate[mid]) ? 0 : 1);
43 }
44
45 // Adds an offset to the given array's values.
46 __global__
47 void add_splitter_map_kernel(unsigned int* const d_out,
48                              const unsigned int* const shift,
49                              const size_t NUM_ELEMS) {
50   const unsigned int mid = threadIdx.x + blockIdx.x * blockDim.x;
51   if (mid >= NUM_ELEMS) return;
52   d_out[mid] += shift[0];
53 }
54
55 // Computes the sum of elements in d_in
56 __global__
57 void reduce_kernel(unsigned int* const d_out,
58                    unsigned int* const d_in,
59                    const size_t NUM_ELEMS) {
60   unsigned int pos = blockIdx.x * blockDim.x + threadIdx.x;
61   unsigned int tid = threadIdx.x;
62
63   for (unsigned int s = blockDim.x / 2; s > 0; s >>=1) {
64     if ((tid < s) && ((pos + s) < NUM_ELEMS))
65       d_in[pos] = d_in[pos] + d_in[pos + s];
66     __syncthreads();
67   }
68
69   // only thread 0 writes result, as thread
70   if ((tid == 0) && (pos < NUM_ELEMS))
71     d_out[blockIdx.x] = d_in[pos];
72 }
73
74 // Maps values from d_in to d_out according to scatter addresses in d_sum_scan_0 or
       d_sum_scan_1.
75 __global__
76 void map_kernel(unsigned int* const d_out,
77                 const unsigned int* const d_in,
78                 const unsigned int* const d_predicate,
79                 const unsigned int* const d_sum_scan_0,
80                 const unsigned int* const d_sum_scan_1,
81                 const size_t NUM_ELEMS) {
82   const unsigned int mid = threadIdx.x + blockIdx.x * blockDim.x;
83   if (mid >= NUM_ELEMS) return;
84   const unsigned int pos = ((d_predicate[mid]) ? d_sum_scan_0[mid] : d_sum_scan_1[mid])
         ;
85   d_out[pos] = d_in[mid];
86 }
87
88 // Calls reduce kernel to compute reduction.
89 void reduce_wrapper(unsigned int* const d_out,
90                     unsigned int* const d_in,
91                     size_t num_elems,
92                     int block_size) {
93   unsigned int grid_size = num_elems / block_size + 1;
94
95   unsigned int* d_tmp;
96   checkCudaErrors(cudaMalloc(&d_tmp, sizeof(unsigned int) * grid_size));
97   checkCudaErrors(cudaMemset(d_tmp, 0, sizeof(unsigned int) * grid_size));
98
99   unsigned int prev_grid_size;
100  unsigned int remainder = 0;
101  // recursively solving, will run approximately log base block_size times.
```

```
102   do {
103     reduce_kernel<<<grid_size, block_size>>>(d_tmp, d_in, num_elems);
104
105     remainder = num_elems % block_size;
106     num_elems  = num_elems / block_size + remainder;
107
108     // updating input to intermediate
109     checkCudaErrors(cudaMemcpy(d_in, d_tmp, sizeof(int) * grid_size,
            cudaMemcpyDeviceToDevice));
110
111     // Updating grid_size to reflect how many blocks we now want to compute on
112     prev_grid_size = grid_size;
113     grid_size = num_elems / block_size + 1;
114
115     // updating intermediate
116     checkCudaErrors(cudaFree(d_tmp));
117     checkCudaErrors(cudaMalloc(&d_tmp, sizeof(int) * grid_size));
118   } while(num_elems > block_size);
119
120   // computing rest
121   reduce_kernel<<<1, num_elems>>>(d_out, d_in, prev_grid_size);
122 }
123
124 // Computes an exclusive sum scan of scatter addresses for the given predicate array.
125 void exclusive_sum_scan(unsigned int* const d_out,
126                         const unsigned int* const d_predicate,
127                         unsigned int* const d_predicate_tmp,
128                         unsigned int* const d_sum_scan,
129                         const unsigned int ARRAY_BYTES,
130                         const size_t NUM_ELEMS,
131                         const int GRID_SIZE,
132                         const int BLOCK_SIZE) {
133   // copy predicate values to new array
134   checkCudaErrors(cudaMemcpy(d_predicate_tmp, d_predicate, ARRAY_BYTES,
          cudaMemcpyDeviceToDevice));
135   // set all elements to zero
136   checkCudaErrors(cudaMemset(d_sum_scan, 0, ARRAY_BYTES));
137
138   // sum scan call
139   for (unsigned int step = 1; step < NUM_ELEMS; step *= 2) {
140     inclusive_sum_scan_kernel<<<GRID_SIZE,BLOCK_SIZE>>>(d_sum_scan, d_predicate_tmp,
          step, NUM_ELEMS);
141     cudaDeviceSynchronize(); checkCudaErrors(cudaGetLastError());
142     checkCudaErrors(cudaMemcpy(d_predicate_tmp, d_sum_scan, ARRAY_BYTES,
          cudaMemcpyDeviceToDevice));
143   }
144
145   // shift to get exclusive scan
146   checkCudaErrors(cudaMemcpy(d_out, d_sum_scan, ARRAY_BYTES, cudaMemcpyDeviceToDevice))
        ;
147   right_shift_array_kernel<<<GRID_SIZE,BLOCK_SIZE>>>(d_out, d_sum_scan, NUM_ELEMS);
148 }
149
150 // Computes an exclusive sum scan of scatter addresses for the given predicate array.
151 unsigned int* radix_sort(unsigned int* h_input,
152                          const size_t NUM_ELEMS) {
153   const int BLOCK_SIZE  = 512;
154   const int GRID_SIZE   = NUM_ELEMS / BLOCK_SIZE + 1;
155   const unsigned int ARRAY_BYTES = sizeof(unsigned int) * NUM_ELEMS;
156   const unsigned int BITS_PER_BYTE = 8;
157
158   // host memory
```

```
159   unsigned int* const h_output = new unsigned int[NUM_ELEMS];
160
161   // device memory
162   unsigned int *d_val_src, *d_predicate, *d_sum_scan, *d_predicate_tmp, *d_sum_scan_0,
          *d_sum_scan_1, *d_predicate_toggle, *d_reduce, *d_map;
163   checkCudaErrors(cudaMalloc((void **) &d_val_src,          ARRAY_BYTES));
164   checkCudaErrors(cudaMalloc((void **) &d_map,              ARRAY_BYTES));
165   checkCudaErrors(cudaMalloc((void **) &d_predicate,        ARRAY_BYTES));
166   checkCudaErrors(cudaMalloc((void **) &d_predicate_tmp,    ARRAY_BYTES));
167   checkCudaErrors(cudaMalloc((void **) &d_predicate_toggle, ARRAY_BYTES));
168   checkCudaErrors(cudaMalloc((void **) &d_sum_scan,         ARRAY_BYTES));
169   checkCudaErrors(cudaMalloc((void **) &d_sum_scan_0,       ARRAY_BYTES));
170   checkCudaErrors(cudaMalloc((void **) &d_sum_scan_1,       ARRAY_BYTES));
171   checkCudaErrors(cudaMalloc((void **) &d_reduce, sizeof(unsigned int)));
172
173   // copy host array to device
174   checkCudaErrors(cudaMemcpy(d_val_src, h_input, ARRAY_BYTES, cudaMemcpyHostToDevice));
175
176   for (unsigned int i = 0; i < (BITS_PER_BYTE * sizeof(unsigned int)); i++) {
177     // predicate is that LSB is 0
178     predicate_kernel<<<GRID_SIZE,BLOCK_SIZE>>>(d_predicate, d_val_src, NUM_ELEMS, i);
179
180     // calculate scatter addresses from predicates
181     exclusive_sum_scan(d_sum_scan_0, d_predicate, d_predicate_tmp, d_sum_scan,
          ARRAY_BYTES, NUM_ELEMS, GRID_SIZE, BLOCK_SIZE);
182
183     // copy contents of predicate, so we do not change its content
184     checkCudaErrors(cudaMemcpy(d_predicate_tmp, d_predicate, ARRAY_BYTES,
          cudaMemcpyDeviceToDevice));
185
186     // calculate how many elements had predicate equal to 1
187     reduce_wrapper(d_reduce, d_predicate_tmp, NUM_ELEMS, BLOCK_SIZE);
188
189     // toggle predicate values, so we can compute scatter addresses for toggled
          predicates
190     toggle_predicate_kernel<<<GRID_SIZE, BLOCK_SIZE>>>(d_predicate_toggle, d_predicate,
           NUM_ELEMS);
191     // so we now have addresses for elements where LSB is equal to 1
192     exclusive_sum_scan(d_sum_scan_1, d_predicate_toggle, d_predicate_tmp, d_sum_scan,
          ARRAY_BYTES, NUM_ELEMS, GRID_SIZE, BLOCK_SIZE);
193     // shift scatter addresses according to amount of elements that had LSB equal to 0
194     add_splitter_map_kernel<<<GRID_SIZE, BLOCK_SIZE>>>(d_sum_scan_1, d_reduce,
          NUM_ELEMS);
195
196     // move elements accordingly
197     map_kernel<<<GRID_SIZE,BLOCK_SIZE>>>(d_map, d_val_src, d_predicate, d_sum_scan_0,
          d_sum_scan_1, NUM_ELEMS);
198
199     // swap pointers, instead of moving elements
200     std::swap(d_val_src, d_map);
201   }
202   // copy contents back
203   checkCudaErrors(cudaMemcpy(h_output,d_val_src,ARRAY_BYTES,cudaMemcpyDeviceToHost));
204   return h_output;
205 }
```

APPENDIX C

Coarse Histogram Implementation

This appendix presents our implementation of the coarse histogram implementation.

Listing C.1: Coarse Histogram implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include "utils.h"
#include "radix_sort.h"

// CONSTANTS
const unsigned int NUM_ELEMS   = 1 << 22;
const unsigned int NUM_BINS    = 100;
const unsigned int ARRAY_BYTES = sizeof(unsigned int) * NUM_ELEMS;
const unsigned int BIN_BYTES   = sizeof(unsigned int) * NUM_BINS;

const dim3 BLOCK_SIZE(1 << 7);
const dim3 GRID_SIZE(NUM_ELEMS / BLOCK_SIZE.x + 1);

const unsigned int COARSE_SIZE = NUM_BINS / 10;
const unsigned int COARSE_BYTES = sizeof(unsigned int) * COARSE_SIZE;
const unsigned int MAX_NUMS = 1000;


__global__
void parallel_reference_calc(unsigned int* const d_out,
                             const unsigned int* const d_in) {
  for (unsigned int l = 0; l < NUM_ELEMS; ++l)
    d_out[(d_in[l] % NUM_BINS)]++;
}


__global__
void coarse_histogram_count(unsigned int* const d_out,
                        const unsigned int* const d_bins,
                        const unsigned int l_start,
                        const int l_end) {
  if (l_end < 0) return; // means that no values are in coarsed bin

  const unsigned int l_pos = l_start + threadIdx.x + blockIdx.x * blockDim.x;
  if (l_pos < l_start || l_pos >= l_end) return;

  const unsigned int bin = d_bins[l_pos];
  // read some into shared memory
  // atomic adds
  // write to global memory
```

```
44    atomicAdd(&(d_out[bin]), 1);
45  }
46
47  __global__
48  void compute_coarse_bin_mapping(const unsigned int* const d_in,
49                                  unsigned int* const d_out) {
50    const unsigned int mid = threadIdx.x + blockIdx.x * blockDim.x;
51    if (mid >= NUM_ELEMS) return;
52
53    d_out[mid] = d_in[mid] / COARSE_SIZE;
54  }
55
56  __global__
57  void compute_bin_mapping(const unsigned int* const d_in,
58                           unsigned int* const d_out) {
59    const unsigned int mid = threadIdx.x + blockIdx.x * blockDim.x;
60    if (mid >= NUM_ELEMS) return;
61
62    d_out[mid] = d_in[mid] % NUM_BINS;
63  }
64
65  __global__
66  void find_positions_mapping_kernel(unsigned int* const d_out,
67                                     const unsigned int* const d_in) {
68    const unsigned int mid = threadIdx.x + blockIdx.x * blockDim.x;
69    if ((mid >= NUM_ELEMS) || (mid == 0)) return;
70
71    if (d_in[mid] != d_in[mid-1])
72      d_out[d_in[mid]] = mid;
73  }
74
75  void init_rand(unsigned int* const h_in) {
76    /* initialize random seed: */
77    srand(time(NULL));
78
79    /* generate values between 0 and 999: */
80    for (int i = 0; i < NUM_ELEMS; i++)
81      h_in[i] = rand() % MAX_NUMS;
82  }
83
84  void sort(unsigned int*& h_coarse_bins,
85            unsigned int*& h_bins,
86            unsigned int*& h_values) {
87    const unsigned int NUM_ARRAYS = 3;
88    // set up pointers
89    unsigned int** to_be_sorted = new unsigned int*[NUM_ARRAYS];
90    to_be_sorted[0] = h_coarse_bins;
91    to_be_sorted[1] = h_bins;
92    to_be_sorted[2] = h_values;
93
94    unsigned int** sorted = radix_sort(to_be_sorted, NUM_ARRAYS, NUM_ELEMS);
95
96    // update pointers
97    h_coarse_bins = sorted[0];
98    h_bins = sorted[1];
99    h_values = sorted[2];
100 }
101
102 void init_memory(unsigned int*& h_values,
103                  unsigned int*& h_bins,
104                  unsigned int*& h_coarse_bins,
105                  unsigned int*& h_histogram,
```

```
106                 unsigned int*& h_positions,
107                 unsigned int*& h_reference_histo,
108                 unsigned int*& d_values,
109                 unsigned int*& d_bins,
110                 unsigned int*& d_coarse_bins,
111                 unsigned int*& d_positions,
112                 unsigned int*& d_histogram) {
113   // host
114   h_values          = new unsigned int[NUM_ELEMS];
115   h_bins            = new unsigned int[NUM_ELEMS];
116   h_coarse_bins     = new unsigned int[NUM_ELEMS];
117   h_histogram       = new unsigned int[NUM_BINS];
118   h_reference_histo = new unsigned int[NUM_BINS];
119   h_positions       = new unsigned int[COARSE_SIZE];
120   // device
121   checkCudaErrors(cudaMalloc((void **) &d_values,       ARRAY_BYTES));
122   checkCudaErrors(cudaMalloc((void **) &d_bins,         ARRAY_BYTES));
123   checkCudaErrors(cudaMalloc((void **) &d_coarse_bins,  ARRAY_BYTES));
124   checkCudaErrors(cudaMalloc((void **) &d_positions,    ARRAY_BYTES));
125   checkCudaErrors(cudaMalloc((void **) &d_histogram,    BIN_BYTES));
126 }
127
128 bool compare_results(const unsigned int* const ref,
129                      const unsigned int* const gpu) {
130   for (unsigned int i = 0; i < NUM_BINS; i++)
131     if (gpu[i] != ref[i]) return false;
132
133   return true;
134 }
135
136
137 void streamed_coarse_atomic_bin_calc(unsigned int*& d_values,
138                                      unsigned int*& h_values,
139                                      unsigned int*& d_bins,
140                                      unsigned int*& h_bins,
141                                      unsigned int*& d_coarse_bins,
142                                      unsigned int*& h_coarse_bins,
143                                      unsigned int*& d_positions,
144                                      unsigned int*& h_positions,
145                                      unsigned int*& d_histogram,
146                                      unsigned int*& h_histogram) {
147   // initialise streams
148   cudaStream_t streams[COARSE_SIZE];
149
150   // compute bin id
151   compute_bin_mapping<<<GRID_SIZE, BLOCK_SIZE>>>(d_values, d_bins);
152
153   // compute coarse bin id
154   compute_coarse_bin_mapping<<<GRID_SIZE, BLOCK_SIZE>>>(d_bins, d_coarse_bins);
155   // move memory to host
156   checkCudaErrors(cudaMemcpy(h_coarse_bins, d_coarse_bins, ARRAY_BYTES,
          cudaMemcpyDeviceToHost));
157
158   // move memory to host
159   checkCudaErrors(cudaMemcpy(h_bins, d_bins, ARRAY_BYTES, cudaMemcpyDeviceToHost));
160   // sort
161   sort(h_coarse_bins, h_bins, h_values);
162   checkCudaErrors(cudaMemcpy(d_coarse_bins, h_coarse_bins,  ARRAY_BYTES,
          cudaMemcpyHostToDevice));
163   checkCudaErrors(cudaMemcpy(d_bins,        h_bins,         ARRAY_BYTES,
          cudaMemcpyHostToDevice));
164   checkCudaErrors(cudaMemcpy(d_values,      h_values,       ARRAY_BYTES,
```

```
      cudaMemcpyHostToDevice));
165
166   // find starting position for each coarsed bin
167   checkCudaErrors(cudaMemset(d_positions, 0, COARSE_BYTES));
168   checkCudaErrors(cudaMemcpy(h_positions, d_positions, COARSE_BYTES,
      cudaMemcpyDeviceToHost));
169
170   // find positions of separators
171   find_positions_mapping_kernel<<<GRID_SIZE, BLOCK_SIZE>>>(d_positions, d_coarse_bins);
172   checkCudaErrors(cudaMemcpy(h_positions, d_positions, COARSE_BYTES,
      cudaMemcpyDeviceToHost));
173
174   // we have entire bin_grid
175   // only access relevant elements in kernel
176   // based on bin_size and bin_start
177   checkCudaErrors(cudaMemset(d_histogram, 0, BIN_BYTES));
178
179   unsigned int local_bin_start, local_bin_end, grid_size;
180   int amount;
181   for (unsigned int i = 0; i < COARSE_SIZE; i++) {
182     // create stream
183     cudaStreamCreate(&streams[i]);
184     // make some local bins
185     local_bin_start = h_positions[i];
186     local_bin_end   = (i == COARSE_SIZE-1) ? NUM_ELEMS : h_positions[i+1];
187     amount = local_bin_end - local_bin_start;
188     // calculate local grid size
189     grid_size = amount / BLOCK_SIZE.x + 1;
190
191     if (amount > 0)
192       coarse_histogram_count<<<grid_size, BLOCK_SIZE, 0, streams[i]>>>(d_histogram,
          d_bins, local_bin_start, local_bin_end);
193   }
194
195   // make sure device is cleared
196   cudaDeviceSynchronize();
197   checkCudaErrors(cudaMemcpy(h_histogram, d_histogram, BIN_BYTES,
      cudaMemcpyDeviceToHost));
198 }
199
200
201 int main(int argc, char **argv) {
202   // host pointers
203   unsigned int* h_values, * h_bins, * h_coarse_bins, * h_histogram, * h_positions, *
      h_reference_histo;
204   // device pointers
205   unsigned int* d_values, * d_bins, * d_coarse_bins, * d_positions, * d_histogram;
206   // set up memory
207   init_memory(h_values, h_bins, h_coarse_bins, h_histogram, h_positions,
      h_reference_histo,
208           d_values, d_bins, d_coarse_bins, d_positions, d_histogram);
209
210   // initialise random values
211   init_rand(h_values);
212
213   // copy host memory to device
214   checkCudaErrors(cudaMemcpy(d_values, h_values,  ARRAY_BYTES, cudaMemcpyHostToDevice))
      ;
215
216   // reset output array
217   checkCudaErrors(cudaMemset(d_histogram, 0, BIN_BYTES));
218   // parallel reference test
```

```
219   parallel_reference_calc<<<1,1>>>(d_histogram, d_values);
220   checkCudaErrors(cudaMemcpy(h_reference_histo, d_histogram,  BIN_BYTES,
          cudaMemcpyDeviceToHost));
221
222   streamed_coarse_atomic_bin_calc(d_values, h_values, d_bins, h_bins, d_coarse_bins,
          h_coarse_bins,
223                                   d_positions, h_positions, d_histogram, h_histogram);
224   printf("STREAMED COARSE ATOMIC BIN (%s)\n",
225       (compare_results(h_reference_histo, h_histogram) ? "Success" : "Failed"));
226
227   cudaFree(d_values); cudaFree(d_positions);
228   cudaFree(d_coarse_bins); cudaFree(d_bins); cudaFree(d_histogram);
229
230   cudaDeviceReset();
231
232   return 0;
233 }
```

Bibliography

[1] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer Systems: A Programmer's Perspective*, volume 2. Prentice Hall Upper Saddle River, 2003.

[2] CL Philip Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences*, 275:314–347, 2014.

[3] Rob Farber. *CUDA application design and development*. Elsevier, 2011.

[4] Nvidia Fermi. Nvidia's next generation cuda compute architecture: Fermi. *Nvidia Whitepaper*, 2009.

[5] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[6] Mark Harris. Optimizing Parallel Reduction in CUDA. `https://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf`. Viewed January 2016.

[7] Mark Harris. CUDA Pro Tip: nvprof is Your Handy Universal GPU Profiler. `http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/`, October 2013. Viewed January 2016.

[8] Nvidia. Tesla k40 gpu active accelerator. *Board Specification*, 2013.

[9] NVIDIA. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, September 2015. Viewed January 2016.

[10] NVIDIA. Profiler User's Guide. `http://docs.nvidia.com/cuda/profiler-users-guide/`, September 2015. Viewed January 2016.

[11] Robert R Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, 1997.

[12] Aater Suleman. Parallel programming: Amdahl's law or gustafson's law. `http://www.futurechips.org/thoughts-for-researchers/parallel-programming-amdahls-law-gustafsons-law.html`, June 2011. viewed January 7th, 2016.

[13] Udacity, John Owens, and David Luebke. CS-344: Introduction to Parallel Programming with CUDA. https://www.udacity.com/course/intro-to-parallel-programming--cs344. Viewed January 2016.