NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY
TDT4200 – PARALLEL COMPUTING

Problem Set 2:
CUDA Intro

Per Magnus Veierland
permve@stud.ntnu.no

September 17, 2015

PART 1: THEORY

PROBLEM 1: PROBLEM 1, ARCHITECTURES & PROGRAMMING MODELS

a) **Briefly explain the differences between the following architectures:**
Keywords being: Homogeneous cores, heterogeneous cores, clusters, NUMA, threads.

i) *Nvidia Maxwell* is the latest generation Graphics Processor Unit (GPU) architecture from Nvidia. Their GPU architecture is highly parallel, with a chip such as the GM204 having 16 Streaming Multi-processors (SMs) split into 4 distinct Compute Unified Device Architecture (CUDA) processing blocks, each containing 32 homogeneous CUDA cores. In total the GM204 chip has 2048 CUDA processing cores.

Within each *processing block* (not to be mixed with *thread block*) there is a register file which is shared by the processing cores and is used for thread register storage and thread local storage. All processing blocks within an SM has one L1/texture cache as well as a shared memory. All SMs share a common L2 cache. There is no L3 cache. For each memory; the thread private memory, the SM shared memory, and the external global memory; there is equal access time for all users, so the architecture has uniform memory access.

Threads in a CUDA system are used to execute kernel programs. Each thread has its own program counter, registers and thread local storage. This data is stored

in the processing block's register file. Threads are organized into *thread blocks* by the application. The GPU global *GigaThread Engine* schedules thread blocks onto SMs. Within an SM threads are executed as part of warps. A *warp* is a collection of 32 threads which is executed simultaneously. The GM204 chip has 4 *warp schedulers* per SM which each can dispatch two instructions per warp per clock cycle. Since there is no register state within the processing cores, only in the register file, the SM can immediately switch between executing different warps without any context switch cost. When executing a warp every processing core must execute the same instruction. This is what makes the architecture Single Instruction Multiple Thread (SIMT); there is a single instruction being executed by multiple threads simultaneously.

ii) *ARM big.LITTLE* is a heterogeneous computer architecture consisting of a set of smaller ("little") processing cores combined with a set of larger ("big") processing cores. The architecture is primarily intended for embedded applications, especially smartphones, where power consumption is of great concern. Having smaller cores with less functionality which can used to handle lighter tasks saves the system from spending more power to achieve the same task with a bigger, more power demanding processing core. The memory access is uniform.

    The *ARM big.LITTLE* can be viewed and used as being composed of two clusters, the "high" cluster consisting of the "big" cores, and the "low cluster" consisting of the "little" cores. With clustered switching, only one of the clusters is used a time and the operating system can only view and use one cluster at a time. The processor cores share access to the L2 cache which is used to transfer state when switching execution between the two clusters.

iii) *Vilje @ NTNU* is a Silicon Graphics International (SGI) Altix ICE X system composed of a cluster of 1404 nodes connected by an Infiniband interconnect which is used for communication between nodes.

    Each node in the system consists of two Intel Xeon "Sandy Bridge" processors bridged with an Intel QuickPath interconnect. Both processors has a separate 16 GB memory which they can access directly. In addition, each processor can access the memory connected to the other processor through the Intel QuickPath interconnect. This makes it a Nonuniform memory access (NUMA) architecture. The advantage is that each processor can access their own memory faster they would be able to access a single shared memory used by both processors; while still being able to take advantage of shared-memory programming by having access to the other processor's memory as well.

    The Intel Xeon E5-2670 processors used for each node has eight homogeneous cores each for a total of 16 processing cores per node. Each processing core has a dedicated L1 (32 KB) and L2 (256 KB) cache. The L3 (20 MB) cache is shared between all cores in the processor.

    The processor cores supports Simultaneous Multi-threading (SMT), also known by the term *Hyper-threading* used by Intel, which allows a single physical pro-

cessor core to simultaneously execute two logical threads. This is possible because each processor core is able to hold the state for two threads simultaneously and weaves their execution through a shared instruction pipeline. This technique helps the processor core better utilize all of its functional units, leading to more efficient execution.

iv) *A Typical modern-day Central Processing Unit (CPU)* consists of one physical processor with 2-8 homogeneous cores and uniform memory access. Most often each core will support SMT also known as *Hyperthreading* for Intel processors which allows each core to execute two threads in one instruction pipeline. A modern CPU typically has 3 on-chip caches known as the L1, L2, and L3 cache. The processor cores are usually quite sophisticated and contains functionality for pipelining with branch prediction, speculative execution as well as predicting memory access and pre-fetching data from memory. A single CPU system, although it has multiple cores, is not usually viewed as a cluster.

b) **Explain Nvidia's SIMT addition to Flynn's Taxonomy, and how it is realized, if at all, in each of the architectures from a).**

Single Instruction Multiple Thread (SIMT) is used by Nvidia to describe an architecture where a single instruction stream is executed in parallel for a set of threads. This is realized by having "warps" of threads execute on a Streaming Multi-processor (SM) which has one instruction decoder for a large set of functional cores. This allows a single instruction to be executed in exact parallel for a set of threads.

The *Nvidia Maxwell* architecture implements SIMT through having a set of SMs, each with four warp schedulers. Each SM has a large number of cores which each warp scheduler uses to execute sets of threads.

None of the other architectures mentioned in a) implements SIMT. SMT / *Hyperthreading* is different as it involves multiple instruction streams and does not perform simultaneous computation.

c) **For each architecture from a), report which classification from Flynn's Taxonomy is best suited. If applicable, also state if the architecture fits Nvidia's SIMT-classification. Give the reasoning(s) behind your answers.**

- *Nvidia Maxwell* can be described as both SIMT and Multiple Instruction Multiple Data (MIMD). The SIMT description is used to describe how warps are executed on the GPU architecture, where a single instruction stream is shared by many threads with different data. A SIMT architecture by definition operates on multiple data streams. However since Maxwell systems has several SMs, each capable of executing multiple kernels in parallel with different instruction streams, the system as a whole can also be described as having multiple instruction streams, thus being a MIMD system as well.
- *ARM big.LITTLE* can be described as a MIMD system as it has multiple processing cores executing multiple instruction streams, and since it supports vector in-

structions allowing it to work on multiple data streams. It can not be described as a SIMT architecture.

- *Vilje @ NTNU* can be described as being a MIMD systems, both on a "low level" since it runs on processors with multiple cores supporting vector instructions – and on a "higher level" since it is a clustered system where different instruction streams are run on different processors with different data. It can not be described as a SIMT system.
- *A typical modern-day CPU* can be described as a MIMD system since it consists of multiple processing cores capable of executing multiple instruction streams in parallel on separate sets of data. It also supports vector instructions which would make each processor core capable of multiple data stream usage. It can not be described as a SIMT system.

PROBLEM 2: CUDA GPGPUS

a) **Explain the terms *Threads*, *Grids*, *Blocks*, and how they relate to one another (if at all).**

- In a CUDA architecture a *thread* represents one distinct execution of a kernel. A kernel is a general-purpose program running on the GPU and can be executed by a large number of threads simultaneously. Each thread has its own registers, program counter and private memory.
- A *block* is a collection of threads executing the same kernel on a single SM. Each block has a very fast local memory which the threads within the block can use to exchange data. Blocks can be executed in any order. All threads in a block will not necessary execute at the same time; see the description of warps in the answer to question *2d*. Threads in a block can synchronize their execution with a special barrier mechanism.

  The *number of threads per block* is specified at runtime when launching a kernel. As a convenience it is implemented as a 3-dimensional vector, where the threads for each block can be organized into 1, 2, or 3 dimensions according to what best describes the problem. The built-in variable `threadIdx` has a unique value available to each CUDA thread specifying its *X*-*Y*-*Z* index within the block. The built-in variable `blockDim` specifies the dimensions of the block.

  For Maxwell, the maximum number of threads per block is 1024, the maximum number of threads in the X-dimension for a block is $2^{32} - 1$, and the maximum number of threads in the Y- and Z-dimensions is 65535.
- When launching a kernel, both the *number of blocks* and the *number of threads per block* is specified at runtime. The set of blocks instantiated for a launched kernel is known as a *grid*.

  The *number of blocks* is also given as a 3-dimensional which can be used according to what best describes the problem. The built-in variable `blockIdx` provides

the *X-Y-Z* index of the block which a CUDA thread is a part of. The built-in variable `gridDim` specifies the dimensions of the associated grid.

For Maxwell, the maximum number of blocks in the X- and Y-dimension of a grid is 1024, and the maximum number of blocks in the Z-dimension of a grid is 64.

Figure 1 shows that a *block* is composed of a set of *threads*, that a *grid* is composed of a set of *blocks*, and that there is a separate *grid* per *kernel* executed on the *device*.

b) **Consider an algorithm whose input has size $2n$, output has size $5n$, and execution time is $5hn \cdot 7h \cdot \log_2(n)$ where $h = 1$ on the GPU and $h = 10$ on the CPU. The acCPU-GPU bus has a bandwidth of r. How big must n be before it is faster to run the dataset with the algorithm detailed on the GPU instead of the acCPU?**

When comparing the total cost of running the two algorithms it is assumed that the input comes from the host memory and the output is to be stored in the host memory.

For the acCPU the total runtime will simply be the execution time. However for the GPU the total runtime will be the GPU execution time plus the time it takes to transfer the input data from the host memory to the GPU and the output data from the GPU back to the host memory.

The following inequality shows for which values of n for which it will be faster to run the dataset with the GPU algorithm instead of the acCPU algorithm.

$$
\begin{aligned}
5 \cdot 10 \cdot n \cdot 7 \cdot 10 \cdot \log_2(n) &> 5 \cdot 1 \cdot n \cdot 7 \cdot 1 \cdot \log_2(n) + \frac{2n}{r} + \frac{5n}{r} \\
3500 \cdot n \cdot \log_2(n) &> 35 \cdot n \cdot \log_2(n) + \frac{2n}{r} + \frac{5n}{r} \\
3500 \cdot \log_2(n) &> 35 \cdot \log_2(n) + \frac{2}{r} + \frac{5}{r} \\
3500 \cdot \log_2(n) &> 35 \cdot \log_2(n) + \frac{2+5}{r} \\
3500 \cdot \log_2(n) \cdot r &> 35 \cdot \log_2(n) \cdot r + 2 + 5 \\
3500 \cdot \log_2(n) \cdot r - 35 \cdot \log_2(n) \cdot r &> 7 \\
\log_2(n) \cdot (3500r - 35r) &> 7 \\
\log_2(n) &> \frac{7}{3500r - 35r} \\
\log_2(n) &> \frac{1}{500r - 5r} \\
\log_2(n) &> \frac{1}{495r} \\
n &> 2^{\frac{1}{495r}}
\end{aligned}
$$

c) **Which of `kernel1()` and `kernel2()` will execute fastest, given that X and Y are *gridDim* and *blockDim* respectively, containing 3 integers with positive powers of 2 higher than 2^4?**

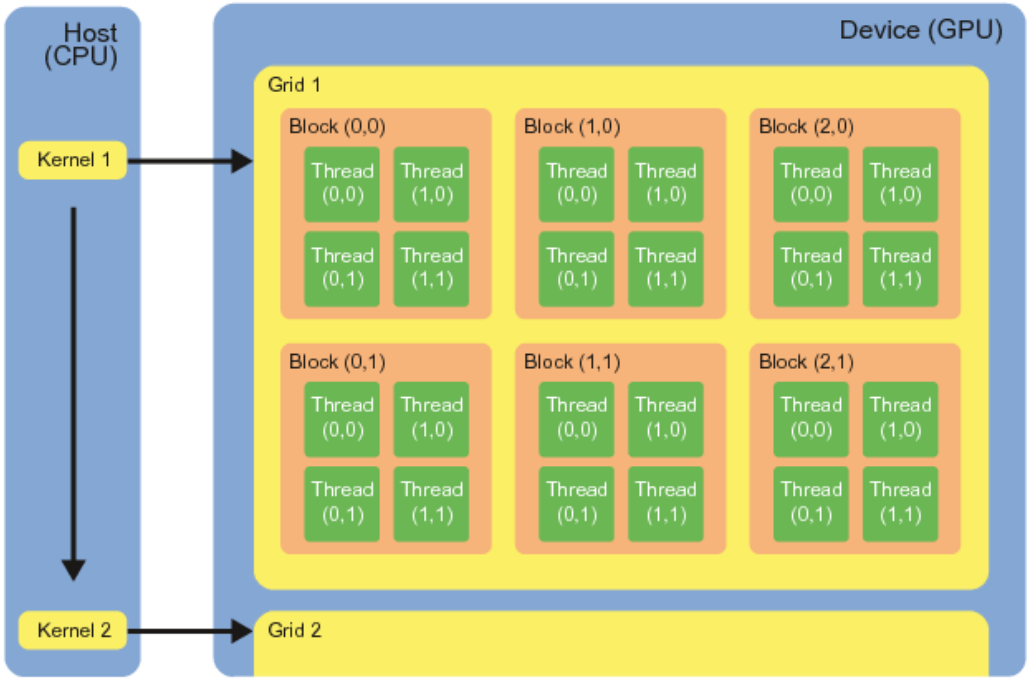

Figure 1: Organizational view showing how kernels, grids, blocks and threads are related in a CUDA GPU architecture. Source: "Performance potential for simulating spin models on GPU" by Martin Weigel 2011 (http://inspirehep.net/record/883697?ln=en).

Of the two kernels, `kernel2()` should execute the fastest. The reason is that the first kernel uses a branch which depends on the thread index which means that threads within a single warp will diverge, something is much more costly than the second kernel's case – where each warp within a block will follow the same code path.

d) **Explain each of the following terms, and how each should be utilized for maximum effect in a CUDA program:**

i) **Warps**

In a CUDA architecture the term *warp* is used to refer to a group of threads being executed simultaneously by an SM. Currently a warp consists of a up to 32 threads. All threads in a warp always executes the same instruction on different CUDA cores. When waiting for memory operations, barrier synchronizations or other dependencies a warp will not be eligible for dispatch and will only be dispatched once the awaited resource is available.

Because of their simultaneous execution it is very important to be aware of the effect of performing computation in warps. Warp divergence occurs when different threads execute different code paths. When this occurs within a warp it can severely reduce performance since only the threads on the relevant code path is performing productive computations while the others are wasted. For this reason it is highly desirable to get rid of branches where possible, or to ensure that different code paths are only taken on warp boundaries.

For maximum effect in a CUDA program, warps must be utilized, and it is desirably that the number of threads per block is always a multiple of 32 such that full warps can be run. It is also important to ensure that there will always be several warps eligible for execution for each SM. Whenever one warp becomes ineligible for dispatch other warps must be available to keep the processors busy. This continued execution of different warps is necessary to mask resource access delays. It is also important to note that the processor switching between different warps carries no extra cost such as is the case with a traditional CPU. In a traditional CPU system time is required to store and retrieve state information to perform context switching between threads. This cost does not exist when switching between warps.

ii) **Occupancy**

Occupancy is a technical term used with CUDA which refers to the ratio of active warps divided by the number of maximum active warps. The occupancy ratio is an important heuristic to achieve the best performance in a CUDA system. One of the main objectives to achieve maximum performance is to keep the memory interface saturated with transfers. Achieving this requires that there is always enough transactions in flight. By weaving the execution of many warps such that there is always available warps which can work while other warps await memory transactions the system can be fully utilized.

The Maxwell architecture supports up to 64 active warps per SM. This is more than the number of active thread blocks per SM, which for Maxwell is 32.

The occupancy ratio can be limited by factors such as register usage, shared memory usage and block size. Depending on the amount of registers used by each thread there may not be enough space to achieve the maximum number of active warps. Shared memory is also a limited resource and can limit the number of active warps in the same way. Block size can also be a limiting factor since there is a limited number of threads per block. If the number of blocks per SM is too low the number of threads will also be too low to achieve the maximum number of active warps.

Nvidia ships an occupancy calculator Excel spreadsheet as part of their SDK which can be used as a tool to estimate the occupancy ratio. CUDA 6.5 also includes a new *Occupancy API* which provides functions to estimate occupancy and other factors directly based on the actual program code.

iii) **Memory Coalescing**

Memory coalescing allows global memory accesses made by a group of threads to be grouped into a single transaction. If four threads within a warp each attempts to read a 32-bit integer from adjacent areas of global memory, the memory transaction can be grouped such that a 128-bit access can be performed instead, thus utilizing the whole memory bus for a memory access. Being aware of memory access patterns is important to achieve maximum performance in a CUDA system.

iv) **Local Memory**

Local memory does not refer to a specific physical memory but is an abstraction representing storage which is private to a thread instance. When there is inadequate register space available or if there for other reasons is not possible to keep a automatic variable in a thread in registers the contents are stored in local memory. It is important to note that the contents of the local memory is actually stored in global memory with the much greater access cost which this entails.

For performance reasons it is important to be aware that automatic thread variables such as large arrays or other structures which consume too much register space will spill to local memory. Since local memory is stored in global memory this will dramatically increase the time to access the data, even if it may appear to be local data. Restructuring variable usage may be necessary depending on the situation.

v) **Shared Memory**

Shared memory is a fast physical memory contained in each SM. When executing CUDA thread blocks it is guaranteed that all threads in a single thread block will be run on the same SM with access to the same shared memory. This makes shared memory the fastest mechanism for sharing and cooperatively computing data between threads. For cases where a block of threads make some set of computations on one set of data retrieved from global memory it can be much faster

to transfer the data to shared memory, perform all computations, and then transfer the data back to global memory – compared to making the transfer from global memory to registers and back for every computation.

CUDA also offers special atomic operations which can be used both against global memory as well as shared memory which can be used for fast concurrency interactions between threads.

PART 2: CODE

PROBLEM 1: CUDA INTRO

a) **In the CUDA file `lenna.cu` implement a kernel which performs the same job as the executable `cpu_version` does. The additionally required setup of memory and variables, freeing of the same, and transfers wrt. to the CUDA kernel are also required.**

The "Lenna" task is to invert each pixel in a 512×512 pixel RGB-image. This task does not involve any composition of sub-results, each byte in the image needs only to be inverted once without any consideration of other pixels.

The first realization we can make is to see that the image can be seen as array of data with a size of $512 \cdot 512 \cdot 3$ bytes = 786432 bytes and that the data does not need to be processed as individual pixels.

When building a CUDA program from a serial program one must consider how to parallelize the program efficiently. This requires taking into account the available hardware. The hardware provided with each workstation in the Tulip lab is a *GeForce GTX 750 Ti* graphics card with a Maxwell GM107 GPU. The GM107 GPU has 5 SMs, each with 4 distinct blocks each having an instruction buffers, warp scheduler and 32 CUDA cores. In total the GPU has 640 CUDA cores.

The first idea to efficiently use the hardware was to have the largest possible number of fixed warps in execution. Using 20 warps of 32 threads all warp schedulers and CUDA cores of the hardware should be kept active at the same time. The data is processed by 20 warps for 153 iterations, plus having the 12 first warps perform 1 extra iteration. Splitting the branching across a warp boundary should avoid warp divergence for the last iteration. The GPU memory bus width is 128 bits and each thread will process 64 consecutive bits from memory at a time for optimal coalescence.

$$(153 \frac{\text{iterations}}{\text{thread}} \cdot 20 \text{ warps} \cdot 32 \frac{\text{threads}}{\text{warp}} + 1 \frac{\text{iterations}}{\text{thread}} \cdot 12 \text{ warps} \cdot 32 \frac{\text{threads}}{\text{warp}}) \cdot 8 \text{ bytes} = 786432 \text{ bytes}$$

Averaged over 50 runs, the kernel execution time for the first approach is 40.19 us. An initial call to `cudaFree(0)` is used at the start of the program to eliminate CUDA context initialization costs from the timing measurements.

One problem with the first approach is that the occupancy is very low. When using the *CUDA Occupancy Calculator* from Nvidia the estimated occupancy of each SM was only 17%, given the following values:

- Compute capability: 2.1
- Shared memory size: 49152 bytes
- Threads per block: 32
- Registers per thread: 20
- Shared memory per block: 0

To improve upon the first attempt, a second program is written to accomplish a higher occupancy rate. Conveniently, 386 blocks with 256 threads each perfectly divides the number of bytes we wish to process by 8:

$$384 \text{ blocks} \cdot 256 \frac{\text{threads}}{\text{block}} \cdot 8 \frac{\text{bytes}}{\text{thread}} = 786432 \text{ bytes}$$

This means that each thread only needs to calculate the complement of a single 64-bit integer. Reapplying the *CUDA Occupancy Calculator* with the updated values gives an estimated occupancy of each SM of 100%. The averaged running time over 50 runs for the second kernel is 14.71 us.

- Compute capability: 2.1
- Shared memory size: 49152 bytes
- Threads per block: 256
- Registers per thread: 6
- Shared memory per block: 0

According to Nvidia the GM107 GPU has a theoretical memory bandwidth of 86.4 GB/s (Note! $86.4 \cdot 10^6$, not $86.4 \cdot 1024^3$). The effective bandwidth utilized by the second kernel can be calculated as:

$$BW_{\text{effective}} = \frac{R_B + W_B}{t_K} = \frac{2 \cdot (512 * 512 * 3 \text{ bytes})}{14.71 \text{ us}} = 106.92 \text{ GB/s}$$

Where R_B is the number of bytes read from memory, W_B is the number of bytes written to memory, and t_K is the kernel execution time. The reason that this calculation exceeds the theoretical maximum bandwidth is most likely that data waiting to be written to global memory is still in cache when the kernel's execution is complete. A hypothetical kernel achieving maximum bandwidth usage should spend 18.20 us for the given amount of data.

Reference: http://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/

In one additional experiment, using a dummy kernel to help warm the system helped reduce the `invert_pixels` runtime from 14.71 us to 13.9 us.

Reference: https://cudaspace.wordpress.com/2013/04/04/difference-on-creating-a-cuda-context-for-timing-a-kernel-warmups/

b) **Implement a `make cuda` makefile rule which compiles (but does not execute) the CUDA executable `gpu_version`.**

A `cuda` rule has been added to the provided `Makefile`.

c) **Time the transfers of data to and from device, and report the percentage of total program run-time the transfers require. How would you suggest to improve this percentage?**

After measuring the program running time it is evident that the majority of the program is spent on decoding the input image, creating the CUDA context, and encoding the output image, see Table 1. The measurements are a result of averaging 50 program runs. Compared to the entire program, only 0.12% is spent on copying the input image to the device and 0.11% is spent on copying the output image from the device.

When discounting the input image decoding, the CUDA context initialization and the output image encoding from the consideration, the copying of the input image to the device takes 29% of the runtime, and copying the output image from the device takes 26% of the runtime.

Reducing the impact of copying between the host and the device on the program's runtime can be accomplished in several ways:

- Through the use of page-locked ("pinned") memory higher bandwidth can be achieved between the host and the target. By default, the memory allocated in host programs is pageable. The GPU cannot access pageable host memory directly and must perform copies from the pageable host memory via a temporary non-pageable memory area on the host before copying it to the GPU. By changing the host program to use page-locked memory via functions such as `cudaMallocHost`, the performance of copying can be improved by a factor of two.
- With larger amounts of data the program can utilize streaming to improve performance significantly. Using streaming, asynchronous copy operations and kernel execution is bound to a stream flow which can be used to segment the program execution. If performing computations on a large dataset, streams can be used to fluidly segment the data such that one segment can be computed on the GPU while the data for the next segment is being copied.

In general, the portion of the program spent on copying data can be reduced directly by increasing the computation executed by the GPU. If the GPU executed more expensive operations on the same data a larger portion of the program time will be spent on computation.

PROBLEM 2: PINKFLOYD INTRO

A working first draft for a program capable of rendering a set of anti-aliased lines on a canvas is presented. Building a good renderer is a large task and there are many large and long avenues for improvement. The program built shows basic usage of the CUDA texture and stream functionality.

Percent of program	Total time (us)	Description	Function
50.8860%	78228.54	Encode output image	`lodepng::encode`
37.3701%	57450.24	Create CUDA context	`cudaFree(0)`
11.3247%	17409.84	Decode input image	`lodepng::decode`
0.1241%	192.88	Copy image to device	`cudaMemcpyHostToDevice`
0.1089%	167.38	Copy image from device	`cudaMemcpyDeviceToHost`
0.1016%	156.19	Allocate device memory	`cudaMalloc`
0.0683%	104.93	Free device memory	`cudaFree`
0.0096%	14.71	Invert pixels kernel	`invert_pixels`

Table 1: Lenna program 384 blocks / 256 threads – Runtime distribution

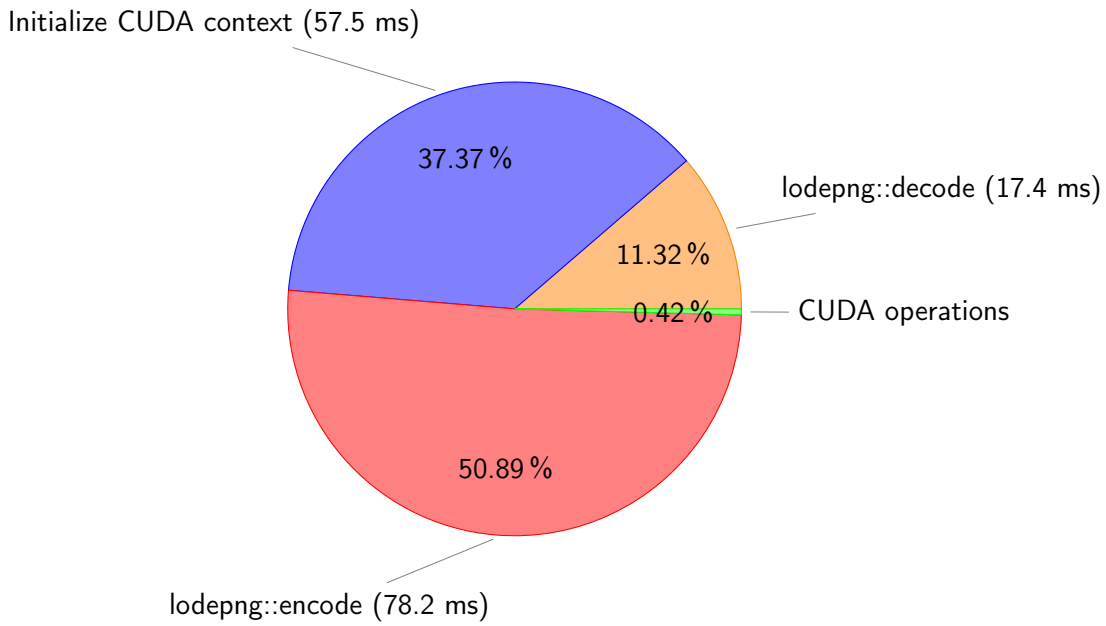

Figure 2: Lenna program 384 blocks / 256 threads – Total program runtime distribution

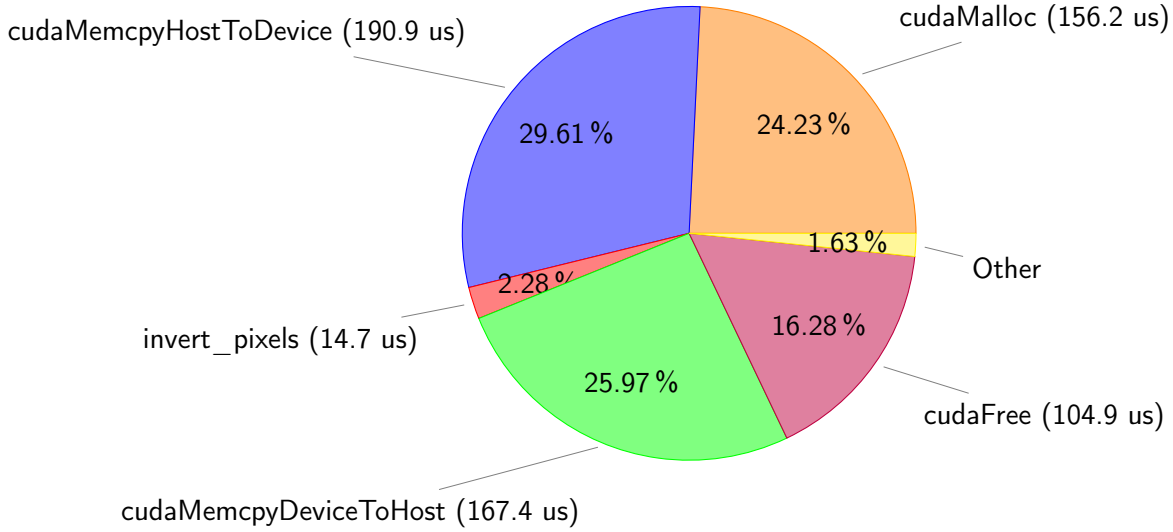

Figure 3: Lenna program 384 blocks / 256 threads – CUDA runtime distribution

Before the lines can be rendered, the 4 equations describing each edge in the line is computed using the `calculate_line_equations` kernel. This kernel is ran with one instance per line and performs a fairly pure computation without branching. After this kernel has completed, a set of streams are created. Each stream is responsible for performing all computations necessary for one part of the image. The stream allows the `draw_line` kernel to be executed for each line in order. Each kernel launch computes all pixels for an area of the final image for the given line.
When rendering anti-aliased lines a function is used to perform the edge filtering. For every pixel which is part of a line, two calculations are needed to determine the resulting alpha value of the pixel. The edge filter is a Gaussian function which is costly to compute, so a lookup table is used to achieve this efficiently.
The Gaussian lookup table is implemented using the CUDA texture feature. Through the use of a normalized 1-dimensional array with clamping, a floating point number can be used to perform lookups in the table. For each pixel in a line, the distance to the closest lateral and longitudinal edge are both calculated as a ratio of half the width of the line. This distance is expressed as a floating point number and can be used directly when looking up the resulting intensity in the table.
When drawing multiple colliding primitives, some strategy must be chosen to find the resulting colors. A popular strategy used with drawing is to use alpha compositing which depends on an alpha value for each pixel to describe its transparency. With alpha compositing there is a common way to divide the ways of composing two elements to be drawn called “Porter Duff”. Ideally, at least two “Porter Duff” modes would be available to draw the Pink Floyd image well. The assignment specifies to use additive composition such that the white lines can be drawn by using lines of different colors which add up to a white color. However, with additive composition it is hard to draw the right section of the image well. The implemen-

tation provided uses the *Over* composition mode to allow one component to be drawn over another correctly. This allows the right area of the image to be drawn well, and as a "cheat" the white lines are simply drawn by using white colors. One natural extension when writing a renderer would be to offer multiple/all "Porter Duff" modes to offer proper drawing.
Reference: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter22.html

Figure 4: Pink Floyd 1000x1000 pixel rendering showing anti-aliasing