ANALYZING GENERAL-PURPOSE COMPUTING PERFORMANCE ON GPU

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Fanfu Meng

December 2015

COMMITTEE MEMBERSHIP

TITLE:	Analyzing General-Purpose Computing Performance on GPU
AUTHOR:	Fanfu Meng
DATE SUBMITTED:	December 2015
COMMITEE CHAIR:	John Y. Oliver, Ph.D. Associate Professor of Electrical Engineering, Director of CPE Program
COMMITEE MEMBER:	Bridget Benson, Ph.D. Assistant Professor of Electrical Engineering
COMMITEE MEMBER:	Tina Smilkstein, Ph.D. Associate Professor of Electrical Engineering

ABSTRACT

Analyzing General-Purpose Computing Performance on GPU

Fanfu Meng

Graphic Processing Unit (GPU) has become one of the most important components in modern computer systems. GPUs have evolved from a single -purpose graphic rendering hardware to a powerful processor that is capable of handling many different kinds of computing tasks. However, GPUs don’t perform well on every application, and it takes a lot of design effort to get good performance on a GPU.

This thesis aims to investigate the relative performance of a GPU vs. CPU. Design effort is held minimum for both CPU implementations and GPU implementations. Matrix multiplication, Advance Encryption Standard (AES) and 32-bit Cyclic Redundancy Check (CRC32) are implemented on both a CPU and GPU. Input data size is varied to test the performance of the CPU and the GPU. The GPU generally has better performance than the CPU for matrix multiplication and AES because of the applications' good instruction and data parallelism. CRC has very poor parallelism, so the CPU performs better. For very small data inputs, the CPU generally outperformed the GPU because of GPU memory transfer overhead.

ACKNOWLEDGMENTS

I would like to thank my thesis committee for helping me with the thesis and my parents for supporting me all these years.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1. Introduction

1.1 Introduction

The Graphic Processing Unit (GPU) was introduced in the 1970s to perform graphic generating tasks such as line drawing [1]. People started to use computers for their daily life as the processing power of the computers gained. Entertainment industries such as movie and gaming became the main driving force in the development of GPU technologies. The software development tools of GPU programming developed along with the GPU hardware. APIs like OpenCL and DirectX brought flexibility into GPU computing through heterogeneous programming models.

The popularity of General-Purpose GPU increased greatly with the introduction of NVIDIA Compute Unified Device Architecture (CUDA). CUDA is a parallel computing platform and programming model invented by NVIDIA [2]. CUDA enables NVIDIA GPUs for general-purpose computing by allowing programmers to directly access to the GPU's virtual instruction set and parallel computational elements [3]. GPUs have since become one of the dominant forces in High-Performance Computing (HPC) area for crunching extremely large amounts of data quickly.

Today, GPUs have made their way into mobile devices. The leading System-on-Chip (SoC) manufactures put a lot of effort into optimizing the performance and power consumption of GPU in order to meet mobile devices' demands for computing power and limited power budget. Thanks to silicon fabrication technologies, an increasing number

of GPU cores are integrated into SoCs to boost GPU performance. Figure 1 and Figure 2 show the die photos of Apple A6 and Apple A8X SoCs. A6 has a dual-core CPU and a triple-core GPU. A8X has a triple-core CPU and an 8-core GPU. The two SoCs are only two years apart, but the number of GPU cores in the package has almost increased 3x.

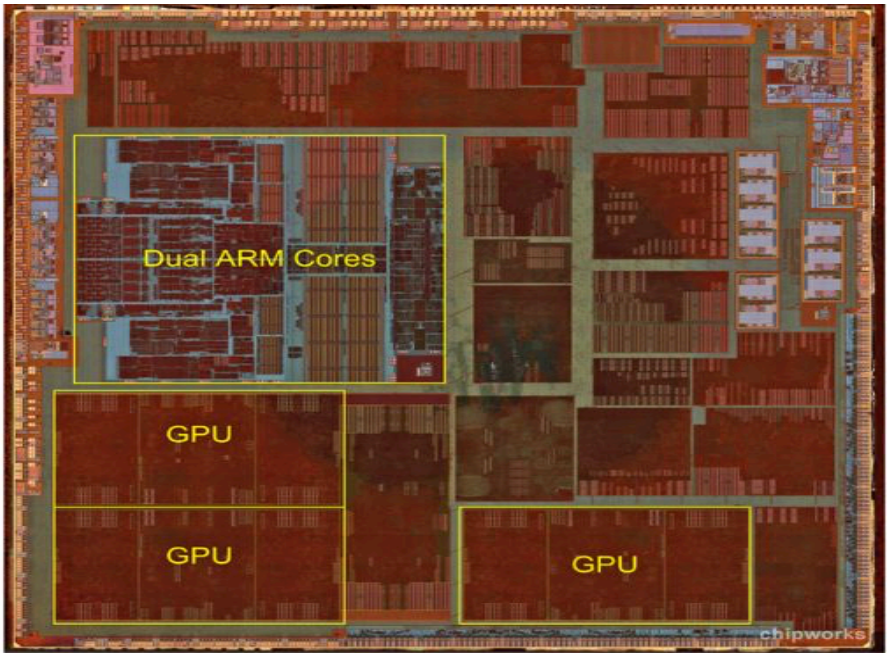

Figure 1. Die Photo of Apple A6 SoC [4]

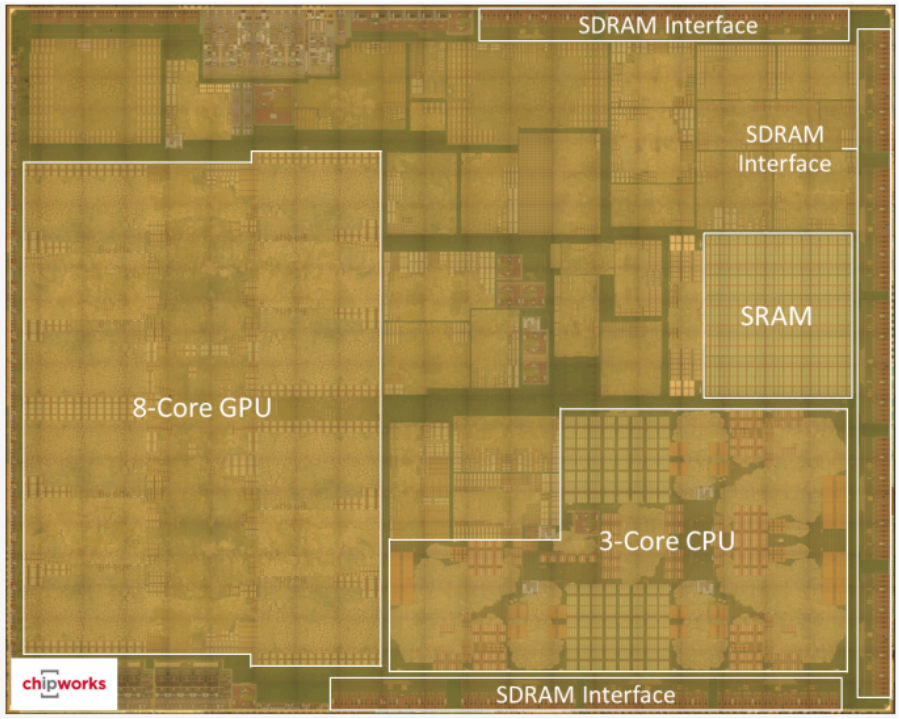

Figure 2. Die Photo of Apple A8X [5]

In many applications, the parallel computing capabilities give GPUs tremendous advantage over CPU, which is a sequential processing based device. However, GPU processing is not good for every application. The CPU will potentially outperform the GPU when there is little parallelism in an application.

This thesis aims to investigate the relative performance of a GPU vs. CPU when minimum design effort is held for both CPU implementations and GPU implementations. Matrix multiplication, Advance Encryption Standard (AES) and 32-bit Cyclic Redundancy Check (CRC32) are implemented on both CPU and GPU. Input data size is varied to test the performance of the CPU and the GPU. The GPU generally has better had better performance than the CPU for matrix multiplication and AES, because these applications have a high amount of

instruction and data parallelism. CRC has a very low amount of parallelism, so the CPU performs better for most of the data sizes. For very small data inputs, the CPU generally outperformed the GPU because of GPU memory transfer overhead.

Detailed background information will be provided in Chapter 2, followed by implementations in Chapter 3, performance analysis in Chapter 4, conclusions in Chapter 5 and future work in Chapter 6.

Chapter 2. Background

The hardware and the programming model used in this thesis are introduced in this Chapter. In the hardware section, the development platform and GPU architecture are discussed. In the programming model section, basics of CUDA programming and CUDA memory types are discussed. Related work is in the end of this chapter.

2.1 Hardware

2.1.1 Development Platform

The development platform used for this thesis is a 32-bit version NVIDIA Jetson TK1 Embedded Development Kit (See Figure 3). The reason for using this mobile device platform is that GPU is becoming more and more important in mobile devices (smart phones, tablets, etc.) as mentioned in Chapter 1, and the GPU of Tegra K1 is one of the top performers on the market. The platform is armed with a Tegra K1 SoC. Linux Unbuntu 14.04 is the operating system on this platform. CUDA 6.5 SDK is installed to compile the code and analyze the performance.

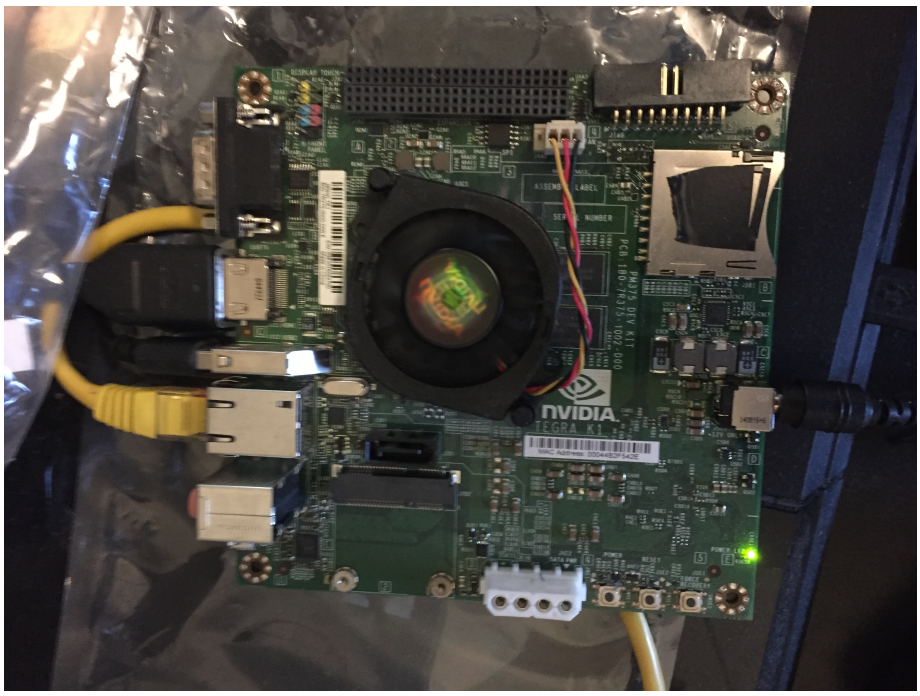

Figure 3.NVIDIA Jetson TK1 Development Board

As shown in Figure 4, the SoC has 5 ARM Cortex A15 cores with one being the battery saving core. The CPU is fabricated with the 28nm process. The maximum clock speed is 2.3 GHz. The Kepler GPU has 192 CUDA cores running at 960MHz.

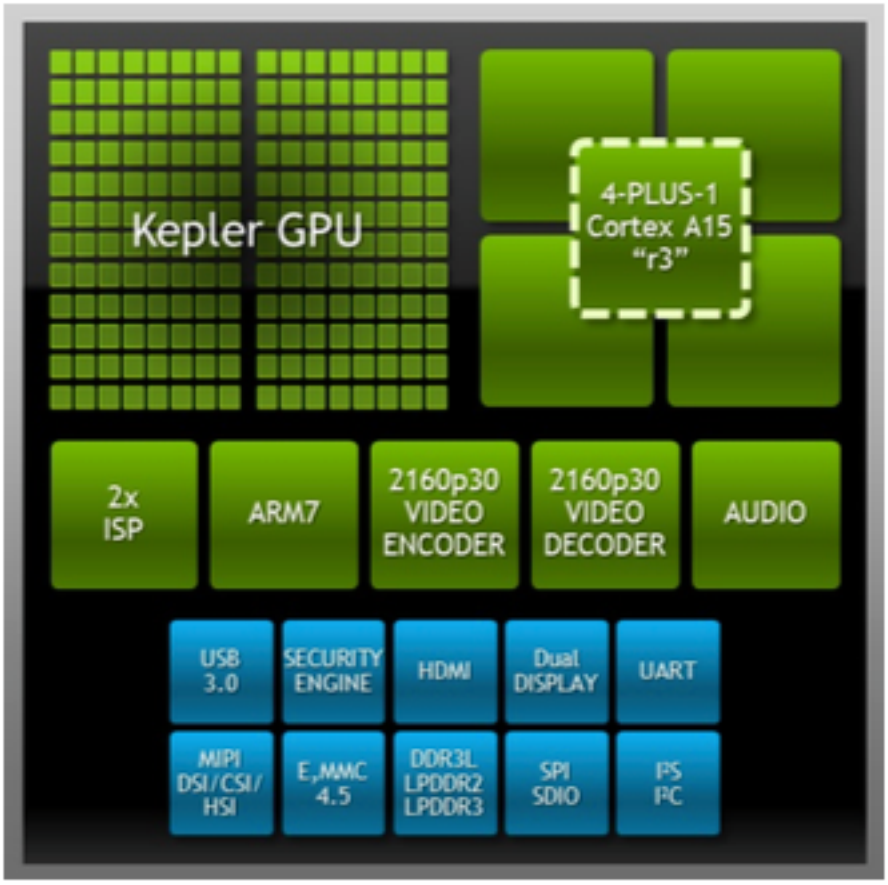

Figure 4. Tegra K1 Block Diagram [6]

2.1.2 GPU Architecture

The architecture of the Kepler GPU in Tegra K1 is virtually identical to the Kepler GPU architecture used in high-end systems [6]. In the big picture, the GPU consists of many small processors and has its own data storage hardware. This section introduces these processors, the memory architecture and hardware parallelism of the GPU.

Streaming Multiprocessor (SMX) and Streaming Processor(SP) of the GPU

Figure 5 illustrates the SMXes and SPs in a desktop GPU. Desktop GPUs usually have multiple SMXes. There is one streaming multiprocessor (SMX) in the GPU on Tegra K1. The SMX consists of many streaming processors (SPs). For the GPU on Tegra K1, there are 192 SPs in the SMX. The term streaming processor is equivalent to CUDA core. To be consistent, this thesis uses streaming processor or SP. The number of SPs in the GPU on Tegra K1 is more than the number of SPs in many entry-level desktop GPU of just a few years ago [6]. NVIDIA GeForce GTS 250 had 128 cores as the entry-level desktop GPU in 2009.

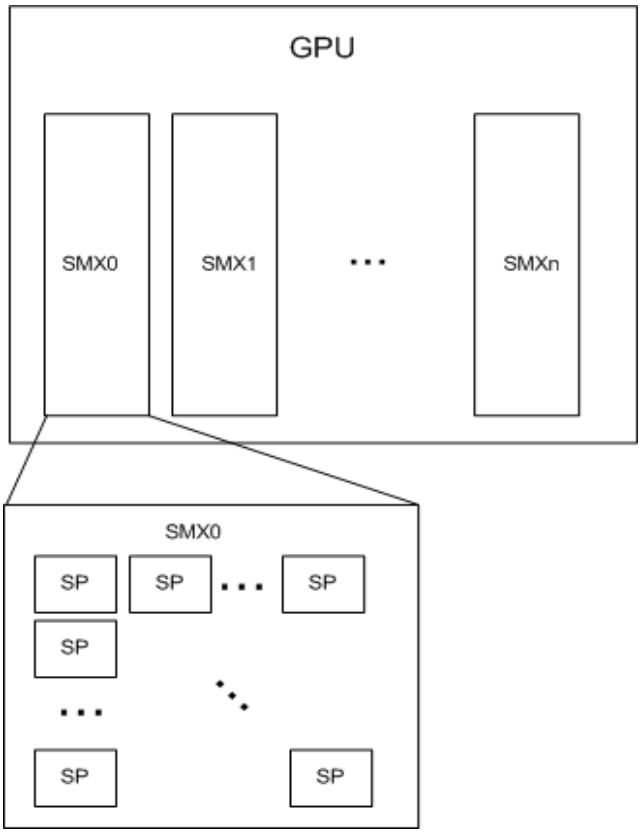

Figure 5. SMXes and SPs in a Desktop GPU

GPU Memory Architecture

The system memory architecture is shown in Figure 6. As shown in Figure 6, the GPU does not have direct access to the system DRAM. The CPU manages memory transactions between the CPU and the GPU. The GPU allocates memory space and copies data from host memory to its own DRAM before processing, and transfers data back to system DRAM after processing. Data transfers between system DRAM and GPU DRAM creates GPU memory transfer overhead.

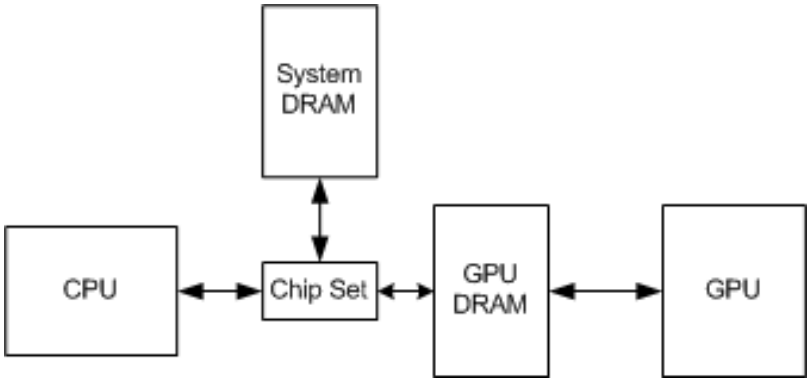

Figure 6. System Memory Architecture

GPU DRAM is off-chip. There are also on-chip data storage hardware in the GPU, cache and shared memory. The GPU implementation approach in this thesis is to fully utilize shared memory, and cache is seldom used. Thus, only shared memory is discussed here.

Shared memory is a piece of high performance on-chip data storage hardware. It acts like a user controllable cache. Programmers can write instructions in software to copy data from DRAM to shared memory

and vice versa. The hardware doesn't evict data at GPU kernel runtime, but the data can be overwritten by software instructions. The concept of GPU kernel will be discussed later in CUDA Programming Model section. Each SMX has its dedicated shared memory that can be accessed by all SPs in the SMX. Shared memory is banked memory. For the GPU on Tegra K1, the bank can be configured as 4-byte or 8-byte. The term shared memory is also used in CUDA programming model by NVIDIA, which will be explained in the programming model section.

Hardware Parallelism

Hardware parallelism is achieved through, as defined by NVIDIA, Single-Instruction-Multiple-Thread (SIMT). SIMT is similar to Single-Instruction-Multiple-Data (SIMD). However, programmers have to explicitly mask the data in a data set in SIMD if this data needs to be executed with different instructions. In SIMT, the masking is done by hardware.

Threads in the GPU are grouped into warps, which are defined by NVIDIA, to be executed simultaneously. A warp can have up to 32 threads. Warps are scheduled and pipelined by the warp scheduler. Figure 7 shows the warp pipelines. The figure shows the GPU executes the instructions in next warp while waiting for data for the current warp. The GPU on Tegra K1 has a quad warp scheduler, and each warp scheduler contains two instruction dispatch units.

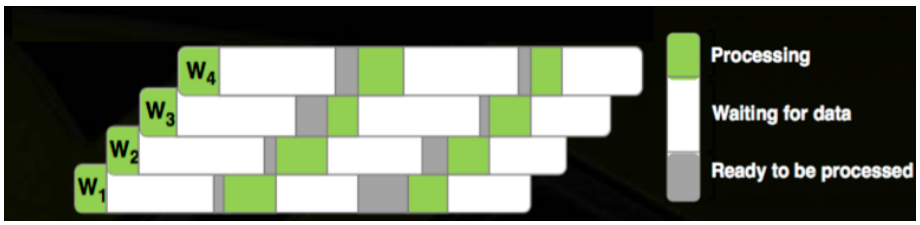

Figure 7. Warp Pipelines in the GPU [7]

2.2 CUDA Programming Model

CUDA provides a complete programming environment. GPU code is written in C/C++ or fortran with a variety of CUDA libraries. This section introduces basics in the CUDA programming model in C including GPU kernels, grids/blocks, types of memory, and the compiling process.

Kernel

A kernel is a function executed on the GPU as array of threads in parallel [7]. The kernel definition starts with __global__. The instructions in the kernel function are executed N times when N threads are launched in parallel [8]. The CUDA runtime library needs to be included in order for the compiler to identify the kernel. Figure 8 shows templates of a kernel and a function that calls the kernel. The number of parallel threads is defined by `grid_size` and `block_size`, which will be explained next.

```
__global__ void kernel (/*arguments*/)
{
     //user instructions
}

void kernel_call(/*arguments*/)
{
     //memory allocation and transfer
     //user instructions
     dim3 grid_size(x, y, z); //grid dimensions
     dim3 block_size(j, k, l); //block dimensions
     kernel(/*arguments*/)<<<grid_size, block_size>>>;
//launching the kernel. There are N = (j*k*l) *
//(x*y*z) threads in total
     //user instructions
     //memory de-allocation
}
```

Figure 8. Templates of a Kernel and a Kernel Call Function

Grids and Blocks

As shown in Figure 8, the kernel needs to be launched with a grid size and a block size. A grid is composed of all CUDA threads in an application. In the grid, threads are divided into thread blocks (see left side in Figure 9). One grid is mapped to one SMX. Multiple grids can run on multiple SMXes in parallel. As shown in Figure 9, blocks are distributed among the SPs to be executed in parallel once a grid is mapped to a SMX. Blocks and threads have their own IDs. The IDs do not have relationships to the physical locations of the SPs. The distribution is managed by hardware.

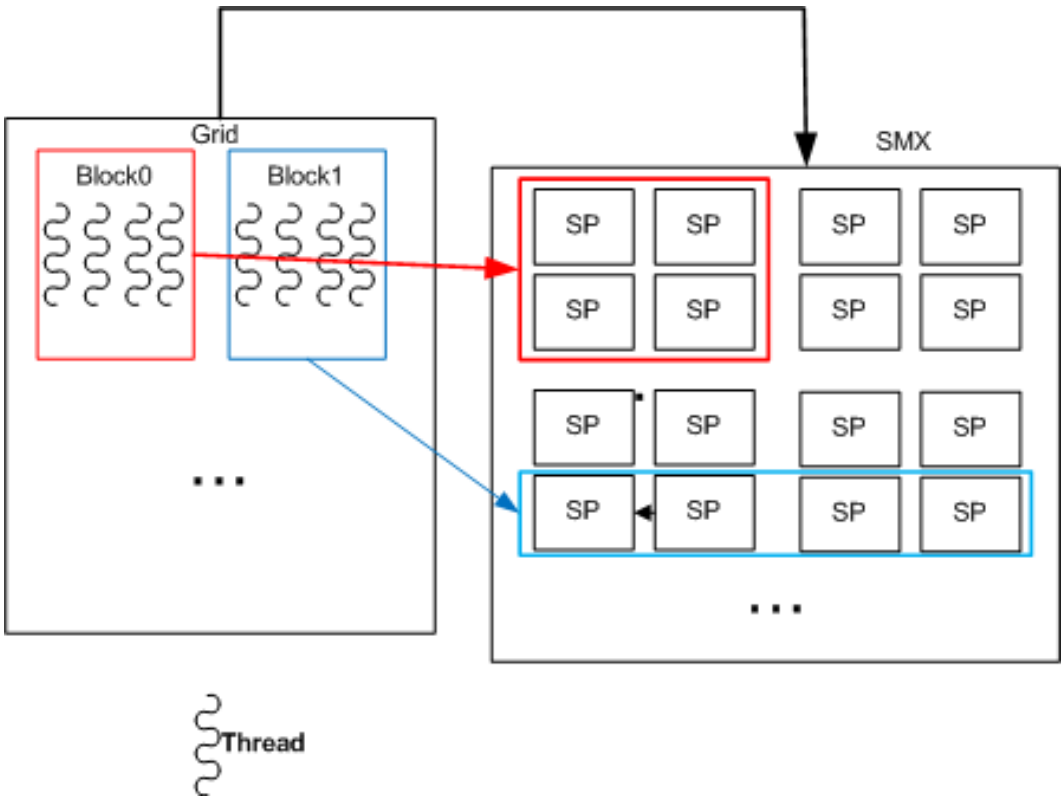

Figure 9. Grid and Block Mapping to Hardware

The dimensions of blocks need to be set when launching the kernel and cannot be changed while the kernel is being executed. There are limitations on the dimensions of grids and blocks. Different GPUs may have different limitations. The GPU on Tegra K1 has the following limitations:

```
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)
```

The information is obtained from running `deviceQuery` application provided in CUDA SDK.

Types of Memory

There are different types of memory in CUDA programming model. The largest and slowest memory is global memory. Global memory resides in the off-chip DRAM. Global memory is per-grid memory, as shown in Figure 10, and also accessible to the CPU. Shared memory (memory type) is a much smaller and faster memory compared to global memory. Shared memory (memory type) resides in the on-chip shared memory hardware. Shared memory is allocated per-block (see Figure 10). Shared memory can only be declared in a kernel function. The lifetime of shared memory is the lifetime of the kernel. The fastest memory is registers. Registers are private to an individual thread. Usually, programmers copy data from global memory into shared memory or registers for better performance.

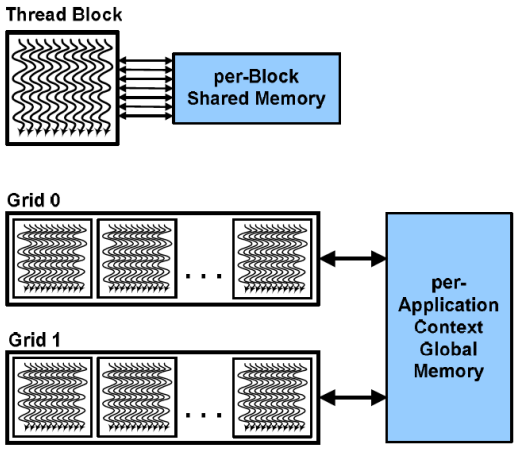

Figure 10. Relationships between Grids/Blocks and Memory [9]

CUDA Compiling Process

The process for compiling CPU code and GPU code is shown in Figure 11. GPU code compiled with NVCC, which is NVIDIA's GPU compiler, while CPU code is compiled with G++. Then the linker links the complied GPU object with the compiled CPU object into an executable. GPU code and CPU code are assigned to NVCC and G++ in the `make` file.

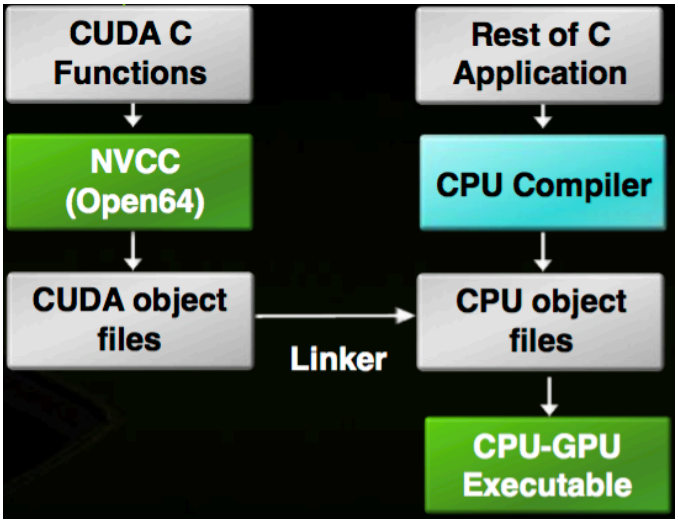

Figure 11. CUDA Compiling Process [7]

2.3 Related Work

Research performed by University of Virginia focuses on the impact of GPU overhead on the overall GPU performance. In the publication *Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer* [10], the authors addressed the significance of GPU overhead through a few applications. Figure 12 shows the results of implementing convolution on a Fermi architecture based GPU documented by the authors. It can be seen that the duration for data

transfer is much longer than the duration of kernel execution. The research concludes that GPU memory transfer overhead is generally too big to ignore.

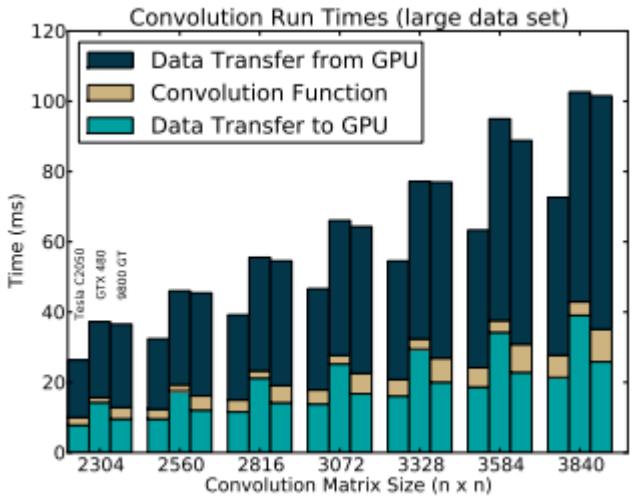

Figure 12. Results of Implementing Convolution on GPU [10]

My thesis takes a similar approach but extends the research with the CPU/GPU performance comparisons with GPU overhead accounted for. The memory transfer overhead is characterized on my specific platform so that the relationship between data size and memory transfer overhead is more obvious.

In *Parallel AES Algorithm for Fast Data Encryption on GPU* published by Changshu Institute of Technology, an implementation of AES on GPU is experimented [11]. The encryption process is as shown in Figure 13. This implementation shows good performance of the GPU implementation.

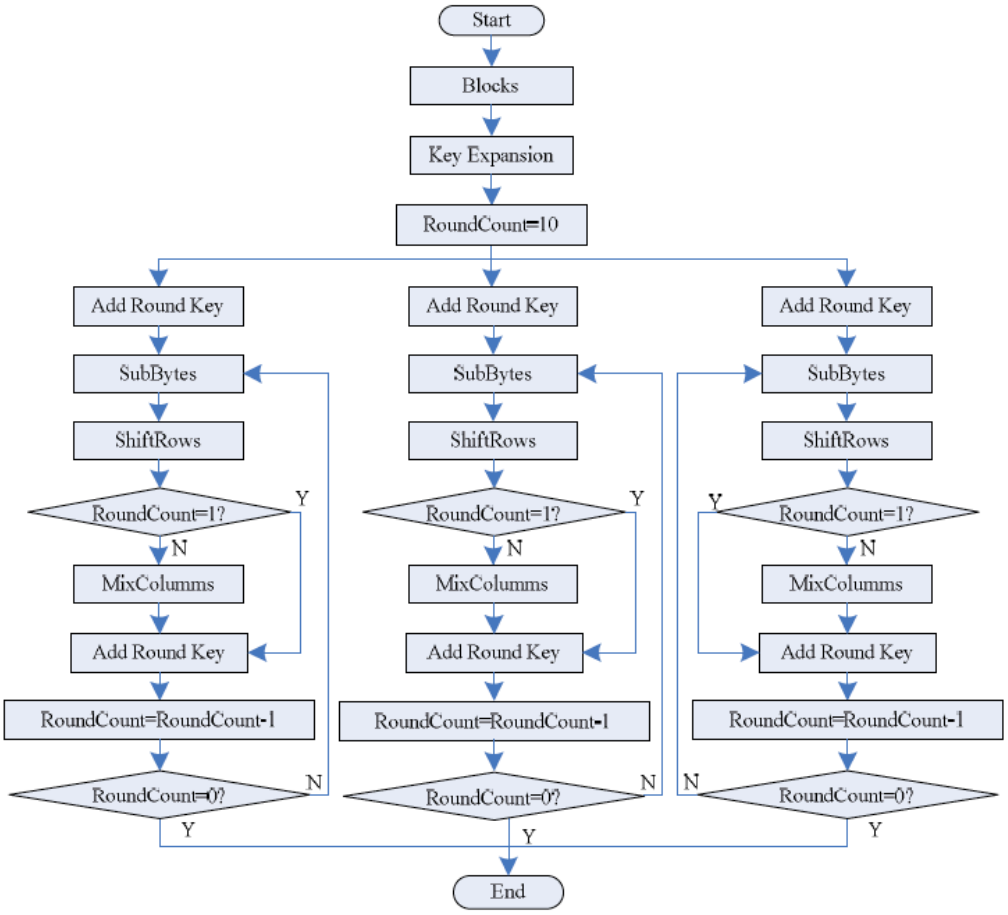

Figure 13. Parallel AES Implementation [11]

However, it is not fully utilizing instruction-level parallelism in AES since key expansion is done prior to the encryption as shown in Figure 13. The implementation of AES in this thesis does the key expansion in parallel along with encryption rounds.

In Improving *Performance of Matrix Multiplication and FFT on GPU* published by Peking University, performance of their GPU matrix multiplication implementation is analyzed in detail [12]. The implementation was optimized specifically for GPU, which utilizes warps and break the input matrices into sub-blocks for parallel processing to

maximize the throughput. It is fair to compare the GPU performance and the CPU performance, because only the GPU is optimized with a great amount of design effort.

The implementation in this thesis only utilized the data parallelism in matrix multiplication. No extra design effort was put into GPU implementation to boost GPU performance. Thus, this thesis provides a fairer CPU vs. GPU performance comparison.

Chapter 3. Implementations of the Applications

3.1 Introduction

There are three applications chosen for comparing the CPU performance and GPU performance: matrix multiplication, Advanced Encryption Standards (AES) and 32-bit Cyclic Redundancy Check (CRC32). These applications are widely used, and they can represent applications that have similar characteristics.

Matrix multiplications are widely used in mathematics and engineering. Matrix multiplication represents applications that are very easy to implement in GPU parallel programming and very likely to have good performance due to good data parallelism and simple instructions.

AES is a widely used encryption algorithm because of its robustness. AES represents algorithms with mixed serial and parallel structure. The encryption steps in AES have to be executed in serial, but key (cipher) expansion and encryption can run in parallel. There is good data parallelism within each serial step.

CRC32 is the common method for checking the integrity of data for data transmission. CRC32 represents algorithms that have poor parallelism. There is no instruction or data parallelism for calculating the CRC32 code for a single input message, but multiple input messages can be processed in parallel.

In this chapter, the CPU implementations and GPU implementations of matrix multiplication, Advanced Encryption Standard

(AES) and Cyclic Redundancy Check (CRC32) are explained. In the big picture, CPU implementations are direct translation of the theoretical algorithms, and the approach of GPU implementations is as following:

- Utilizing instruction-level parallelism and data parallelism in each application if there is any
- Utilizing shared memory
- Providing no optimizations are made for GPU warp pipelining

By following these rules, a programmer with basic CUDA programming knowledge can implement matrix multiplication, AES and CRC32 quickly.

3.2 Matrix Multiplication

Matrix multiplication computes the product of two matrices. The theoretical operation is shown in Figure 14, where matrix A and matrix B are the input matrices, and matrix C is the output matrix.

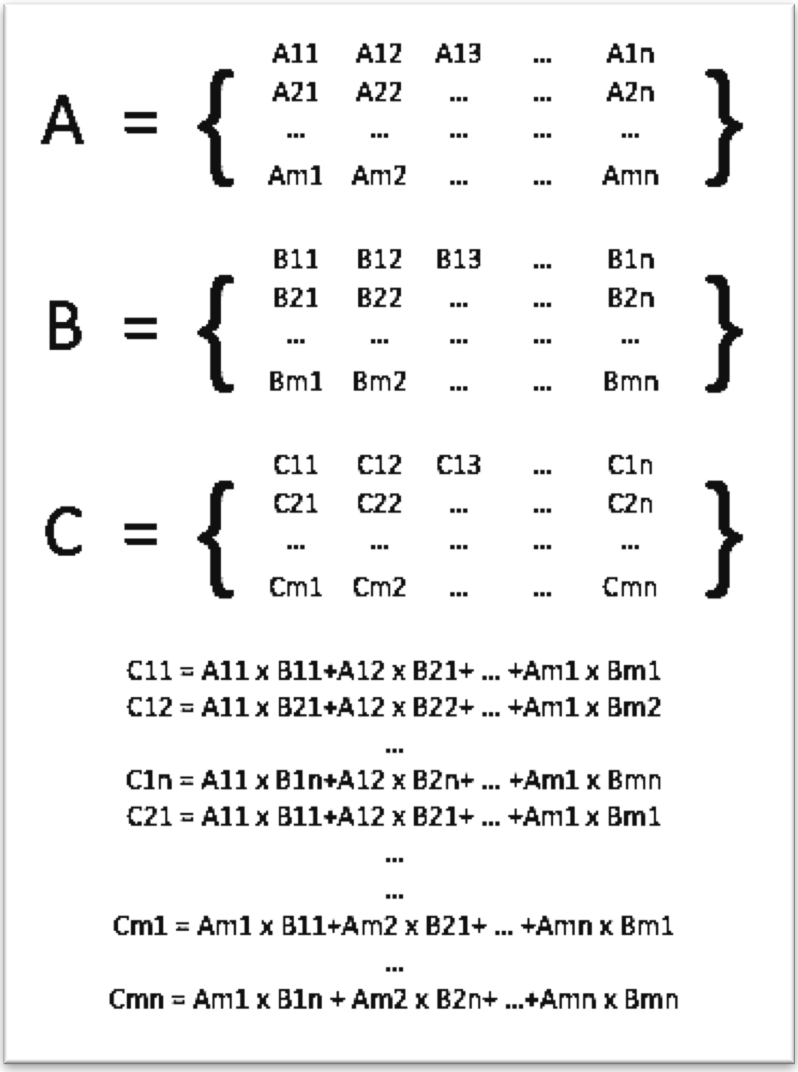

Figure 14. Matrix Multiplication

CPU Implementation

The CPU implementation is the most basic implementation. The pseudo code is as shown in Figure 15:

```
for i < A_heigh; row++
     for j < B_width; column++
          for k < A_width; column ++
               C[i][j]+=A[i][k]*B[k][j]
          end
     end
end
```

Figure 15. Matrix Multiplication Pseudo CPU code

In each iteration of the *for* loops, the code computes the products of two elements from matrix A and matrix B and adds the product to the result from previous iterations of the loops. For an n-by-n square matrix, to compute one element in matrix C, the CPU performs n multiply operations and n-1 sum operations. So, for the entire multiplication, there are n^3 multiply operations and n^2 x (n-1) sum operations.

GPU Implementation

The GPU utilizes the data parallelism in matrix multiplication. The kernel launches n^2 thread blocks for an n-by-n square matrix multiplication. Each block does the multiplication and summation for a row in input matrix A and a column in input matrix B (see Figure 16). Each thread in a block computes the product of one element in the row and one element in the column (see Figure 17).

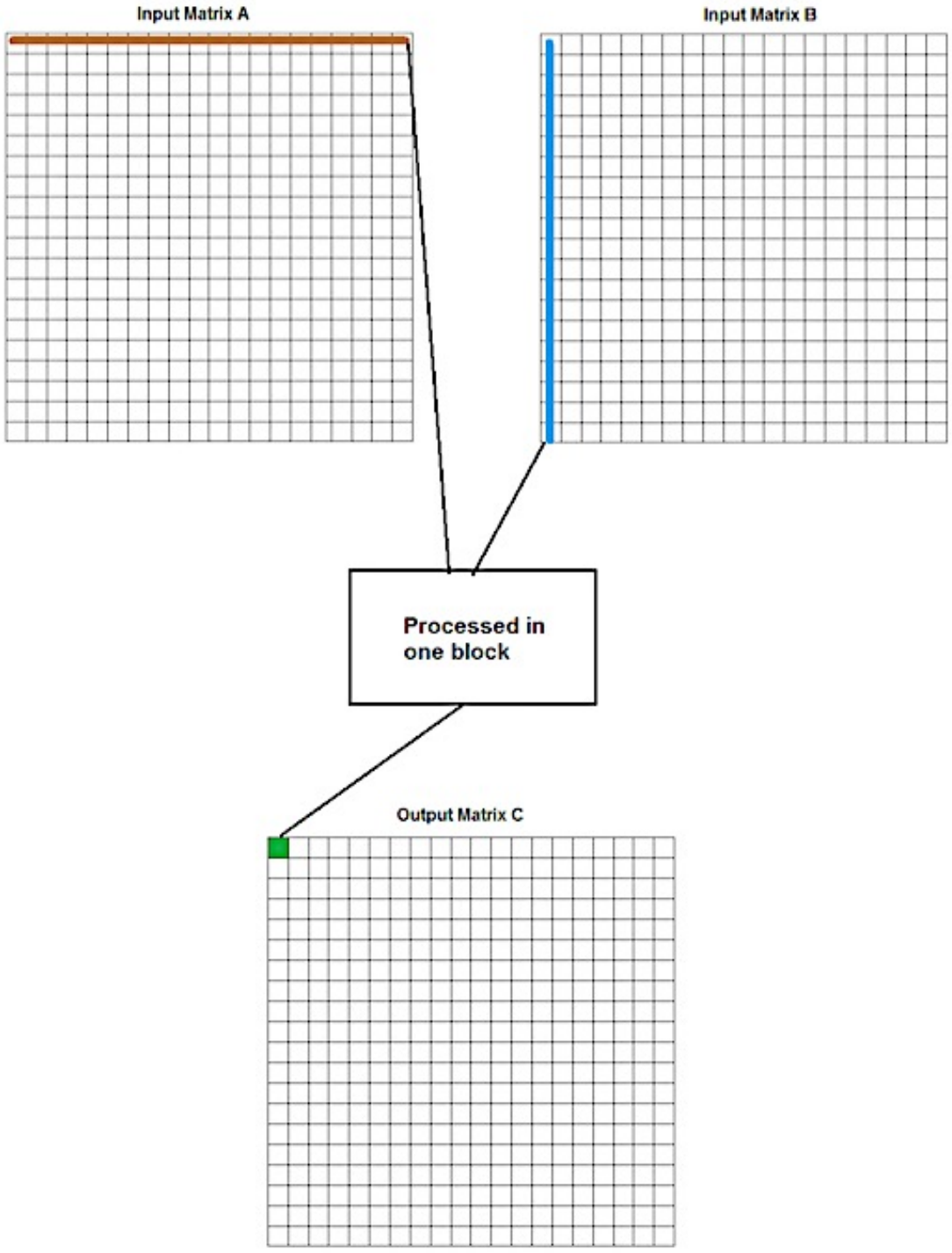

Figure 16. Matrix Multiplication GPU Implementation – Thread Block Operation

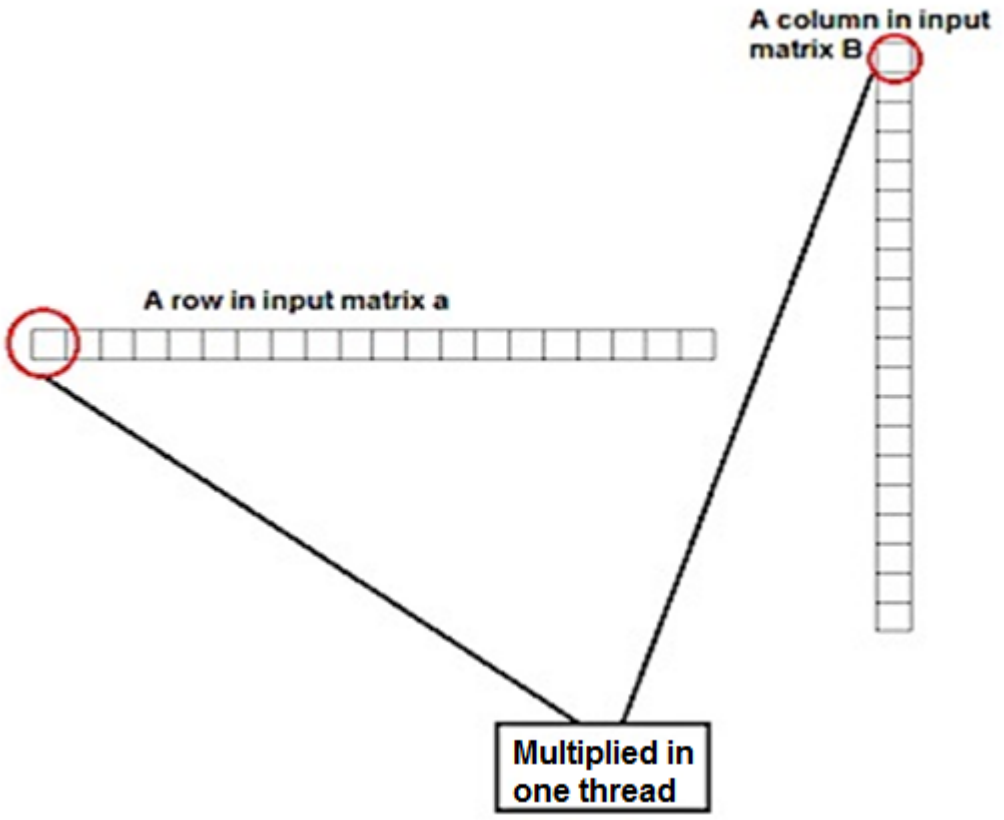

Figure 17. Matrix Multiplication GPU Implementation - GPU Thread Operation

The multiplication core in CPU code and GPU code are in Figure 18 side by side. It can be seen that the GPU is not more complicated than the CPU code. The only tricky part is the indexing with block ID and thread ID.

CPU

```
for(i=0;i<size;i++)
   {
       for(j=0;j<size;j++)
       {
           c[i][j]=0;
           for(int k=0;k<size;k++)
           {
               c[i][j]=c[i][j]+a[i][k]*b[k][j];
           }
       }
   }
```

GPU

```
//indexing
int a_index = size*(blockIdx.x/size)+threadIdx.x;
int b_index = (blockIdx.x%size)+threadIdx.x*size;

//global mem to shared mem
s_a[threadIdx.x] = a[a_index];
__syncthreads();
s_b[threadIdx.x] = b[b_index];
  syncthreads();
//multiplication
s_a[threadIdx.x] = s_a[threadIdx.x] * s_b[threadIdx.x];
__syncthreads();


//addition - parallel reduction
for(
    unsigned int s=1; s < blockDim.x; s *= 2) {
    if (threadIdx.x % (2*s) == 0)
    {
        s_a[threadIdx.x] += s_a[threadIdx.x + s];
    }
    __syncthreads();
}
```

Figure 18. Matrix Multiplication – CPU Code vs. GPU code

3.3 AES

The AES in this thesis is 128-bit AES, which means every 128 bits of data are encrypted with a 128-bit cipher. The AES algorithm deals two sets of data, the state and the key. The state is the actual message, and the key is the cipher. By utilizing the instruction parallelism in AES, state and keys are calculated independently. Keys follow the key schedule. In each round of encryption, keys are added to the state through XOR operation.

CPU Implementation

The CPU implementation calculates keys for the entire message of prior to the actual encryption. In the end, the 16 bytes of cipher expand to 176 bytes. The process of the key expansion core is shown in Figure 19.

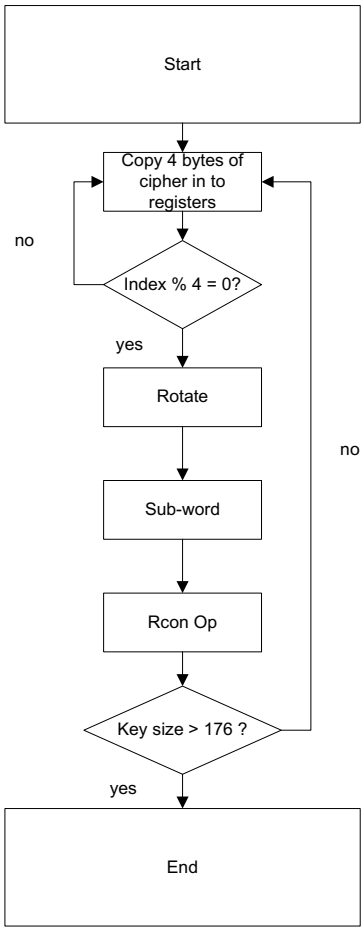

Figure 19. AES Key Expansion Process

The core processes of the key expansion are as follows:

- Rotate: rotate the 4-byte word to the left by 1. For example,

  K0 = K1, K1 = K2, K2 = K3 and K4 = K0.

- Sub-word: the previous round keys are substituted with values in s-box. s-box is a hard-coded lookup table. For example, key_next_round = s-box[key_previous_round].
- Rcon Op: Rcon is another hard-coded lookup table. There is one Rcon value for each iteration in the key expansion loop, Rcon[round_number/(128/(4x3))]. Then the value that comes out from the Rcon table and the previous key are processed through an XOR operation.

After the round key expansion, the encryption process starts. The process is shown in Figure 20.

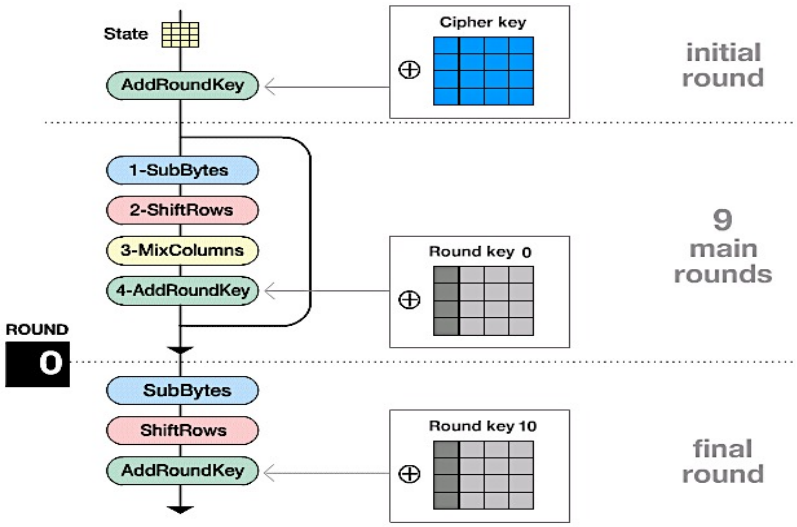

Figure 20. AES Encryption Process [13]

The core processes of the encryption are as follows:

- Sub-byte*:* It is the same as *Sub-word* step in the key expansion.
- Shift rows*:* It is the same as *Rotate* in the key expansion.

- Mix columns*:* Each column is treated as a polynomial over GF(2^8). Then the column is multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x) = 3x^3 + x^2 + x + 2$. The multiplication is performed with XOR since the math is done in Rijndael's Galois field.
- Add round keys*:* XOR each element in the state with the element in the round key that is in the same position in the 4-by-4 matrix

There is a total of 10 rounds of encryption for every 128 bits of data.

GPU Implementation

The GPU implementation utilizes instruction-level parallelism between the key expansion process and encryption process and data parallelism in each step in key expansion and encryption. As shown in the key expansion process and encryption process, the operations in each step depend on the results from the previous step. Thread synchronization is used to ensure the serial process. In each round of encryption, the process is implemented as shown in Figure 21, which indicates that the steps in key expansion and steps in encryption are processed in parallel.

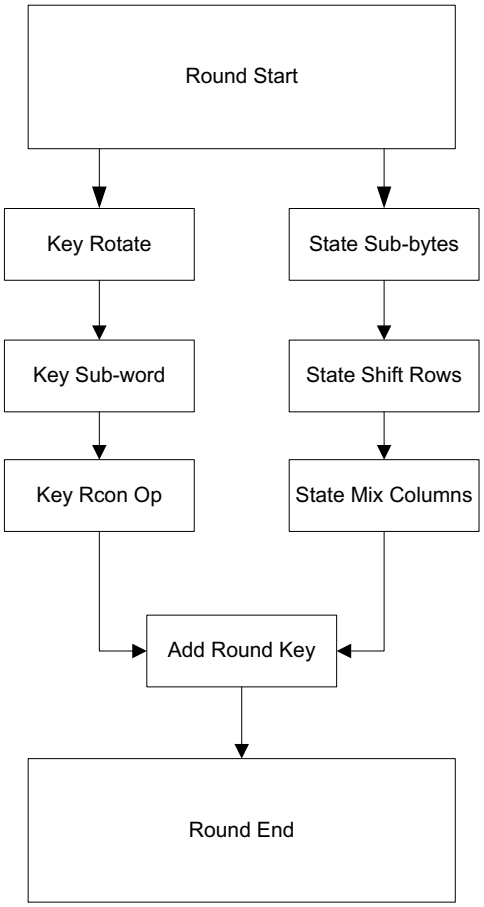

Figure 21. AES GPU Encryption

For example, Figure 22 shows the first step in the kernel:

```
//key rotate
if (threadIdx.x < 4){
    tk[threadIdx.x] = temp[threadIdx.x+1];

    if(threadIdx.x==3)tk[threadIdx.x] = temp[threadIdx.x-3];}
//sub bytes
else if (threadIdx.x>3){
    s_data[threadIdx.x-4] = s_sbox[s_data[threadIdx.x-4]];
}

__syncthreads ();
```

Figure 22. AES GPU Implementation Key Rotate and State Sub-bytes

Operations in key expansion and round encryption have good data parallelism within each step, which is suitable for GPU programming. The only tricky steps are rotation steps. CPU implementation requires a temporary register to hold the data in the first position while copying data from position 2 to position 1, position 3 to position 2 and position 4 to position 3. Instead of doing the serial process, 16 bytes of `char` registers are created to hold the values after rotation. To take advantage of the memory redundancy, the temporary registers are used instead of copying the values back to the original registers in the next step.

A comparison of *Shift Rows* step in the encryption between CPU code and GPU code is shown in Figure 23. The CPU code is longer, but it is simpler than the GPU code. In the GPU code, thread IDs are selected for designated tasks. From GPU programming's standpoint, the GPU code is not complicated.

```
void ShiftRows()
{
    unsigned char temp;

    // Rotate first row 1 columns to left
    temp=state[1][0];
    state[1][0]=state[1][1];
    state[1][1]=state[1][2];
    state[1][2]=state[1][3];
    state[1][3]=temp;

    // Rotate second row 2 columns to left
    temp=state[2][0];
    state[2][0]=state[2][2];
    state[2][2]=temp;

    temp=state[2][1];
    state[2][1]=state[2][3];
    state[2][3]=temp;

    // Rotate third row 3 columns to left
    temp=state[3][0];
    state[3][0]=state[3][3];
    state[3][3]=state[3][2];
    state[3][2]=state[3][1];
    state[3][1]=temp;

}
```

CPU

```
//shift rows
else if(threadIdx.x > 3 && threadIdx.x < 8)
    {sft_temp[threadIdx.x-4] = s_data[threadIdx.x-4];}
else if (threadIdx.x > 7 && threadIdx.x <12){
    if(threadIdx.x == 11)
        {sft_temp[threadIdx.x-4]=s_data[4];}
    else{sft_temp[threadIdx.x-4] = s_data[threadIdx.x-3];}}
else if (threadIdx.x>11 && threadIdx.x<16){
    if(threadIdx.x>13){sft_temp[threadIdx.x-4] = s_data[threadIdx.x-6];}
    else{sft_temp[threadIdx.x-4] = s_data[threadIdx.x-2];}}
else if (threadIdx.x>15 && threadIdx.x<20){
    if(threadIdx.x>16){sft_temp[threadIdx.x-4] = s_data[threadIdx.x-5];}
    else{sft_temp[threadIdx.x-4] = s_data[threadIdx.x-1];}}
__syncthreads ();
```

GPU

Figure 23. AES Implementation - Comparison Between CPU Code and GPU Code

3.4 32-bit Cyclic Redundancy Check (CRC32)

CRC32 has no instruction parallelism or data parallelism. There is a possibility to break up one message in several parts and combine the CRC results into one as the way zlib's *crc32_combine* does, but the complexity in the combining algorithm is too high for the GPU to gain any advantage over CPU. The combining method is not a practical way to implement on the CPU as well due to its high complexity. There are other faster lookup table oriented CRC32 implementations such as Intel proposed in " Fast CRC Computation for iSCSI Polynomial Using CRC32 Instruction"[14]. However, the lookup tables can be rather large that exceeds the size of GPU shared memory. In this case, threads have to constantly access the global memory on the GPU, which violates the design approach in this thesis. Accessing global memory greatly decreases the performance of the GPU.

CPU Implementation

The CPU implementation is based on Stephan Brumme's bitwise branch-free CRC32 [15]. The pseudo code is shown in Figure 24, where the `polynomial` is `0xEDB88320`.

```
while (length > 0)
     crc  = crc ^ data[index++]
     for cnt <8
          crc = (crc >> 1)^(-int(crc) & polynomial)
          cnt++
```

Figure 24. CRC CPU Implementation – CRC Pseudo Code

GPU Implementation

The same CRC function in the CPU code is defined for the GPU in the GPU code. The kernel calls this function after copying the data into shared memory. Each thread block runs a single thread to do the CRC process. Many thread blocks run in parallel to process multiple sets of data.

Chapter 4. Performance Analysis

4.1 Introduction

This chapter will show and analyze the performance of GPU implementations and CPU implementations of matrix multiplication, AES and CRC32. Other findings from the experiments are in the end of this chapter, which includes GPU memory transfer overhead characterization.

For each application, the input data size is varied to test the performance. For all CPU implementations and GPU implementations, the performance is measured as throughput. For GPU performance, two kinds of throughput are measured:

- GPU kernel throughput: GPU memory transfer overhead is not included in the throughput calculation. It is the theoretical performance of the GPU. GPU kernel throughput is calculated as:

$$GPU\ Kernel\ Throughput\ = \frac{Data\ Size}{GPU\ Kernel\ Execution\ Time}$$

- GPU overall throughput: GPU memory transfer overhead is included in the throughput calculation. It is the real performance of the GPU. GPU overall throughput is calculated as:

$$GPU\ Throughput = \frac{Data\ Size}{GPU\ Kernel\ Execution\ Time + GPU\ Memory\ Trnasfer\ Overhead}$$

GPU timers are set to get the GPU kernel execution time and GPU memory transfer overhead. GPU memory transfer overhead include time

for data moving from system DRAM to GPU DRAM and from GPU DRAM to system DRAM.

There is a CPU timer set to get CPU processing time as well. The CPU throughput is calculated as:

$$CPU\ Throughput = \frac{Data\ Size}{CPU\ Execution\ Time}$$

4.2 Matrix Multiplication Performance

To test the performance of the CPU implementation and GPU implementation, the CPU function and the GPU function take in two arrays that store values of two n-by-n matrices, and compute the product of the matrices. The input matrices are square matrices. The dimensions of the input matrices range from 8-by-8 to 64-by-64 increasing in powers of 2. The values of the input matrices are randomly generated 1-byte data.

The throughputs of the CPU implementation and the GPU implementation are shown in Figure 25. Y-axis is log10 scale. The higher the bar is, the better the performance is. The plot indicates that GPU kernel has higher throughput than the CPU for all input data sizes. GPU overall throughput is higher when the size is larger than or equal to 2kB (16 x 16 x 4 x 2 = 2kB, where 4 comes from that elements in the matrices are 4-byte data, and 2 comes from that there are two input matrices). For 8-by-8 input matrices, CPU throughput is 2.5x the GPU overall throughput. The GPU overall throughput increases when input data size increases.

For 64-by-64 input matrices, the GPU overall throughput is 100x the CPU throughput.

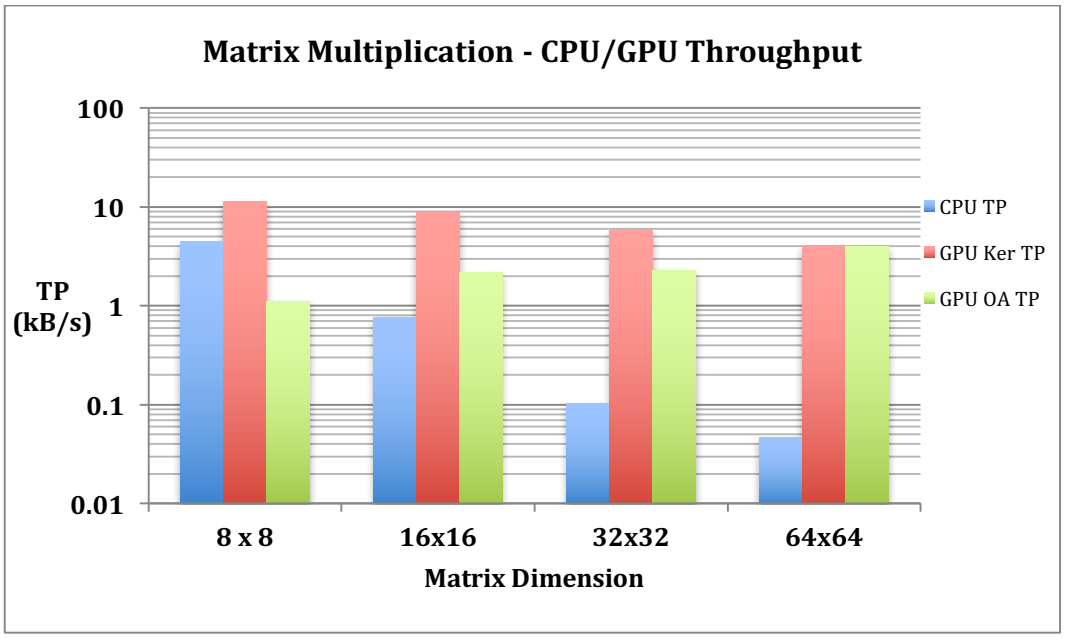

Figure 25. Matrix Multiplication – CPU/GPU Throughput

Memory access is the cause of the decreasing trend of the CPU throughput. One reason being the memory locality was not taken into consideration in the CPU implementation since the design effort was held minimum. Thus, the larger the data is, the poorer performance the CPU gets. The other reason is the cache capacity. The 32kB L1 cache cannot hold the data if the data size is too large. When the input matrices are 64 x 64, there will be 64 x 64 x 4 x 3 = 48kB data on the fly, where 4 is the size of the elements in the matrices, and 3 includes 2 input matrices and 1 output matrix.

The GPU kernel throughput has a decreasing trend as well. The limitation of parallel computing ability of the GPU is one of the reasons. The Kepler GPU on Tegra K1 can schedule and pipeline 64 warps

simultaneously, which translates to 2048 threads at a time. There are 4096 threads running to multiply two 64 x 64 matrices. The threads need to be scheduled twice to complete matrix multiplication. The implementation could be optimized to process data in smaller chunks at a time for better pipelining to get better performance.

The GPU memory transfer overhead has negative impact on the overall GPU throughput. As shown in Figure 25, when the dimension of the input matrices is 8-by-8, GPU memory transfer overhead causes the GPU overall throughput to be 10x lower than the GPU kernel overhead, which indicates that the memory transfer time is much longer than the kernel execution time. The GPU overall throughput is 3x lower than the CPU throughput for 8-by-8 matrices. As input data size grows, the GPU overall throughput increases gradually and gets closer and closer to the GPU kernel throughput, which indicates that the GPU execution time grows faster than the GPU memory transfer overhead when data size increases.

4.3 AES Performance

To test the performance of AES on the CPU and the GPU, the programs encrypt randomly generated messages. The message is composed of an array of randomly generated 1-byte data. The message size goes from 512kB to 16MB increasing in powers of 2.

The throughputs of the CPU implementation and the GPU implementation are shown in Figure 26. Higher bars indicate better

performance. Y-axis is in linear scale. The plot shows that the GPU kernel has higher throughput for all test cases. The overall GPU throughput is higher when the input message size is larger than or equal to 4MB. At 16MB, the GPU overall throughput is 1.5x the CPU throughput.

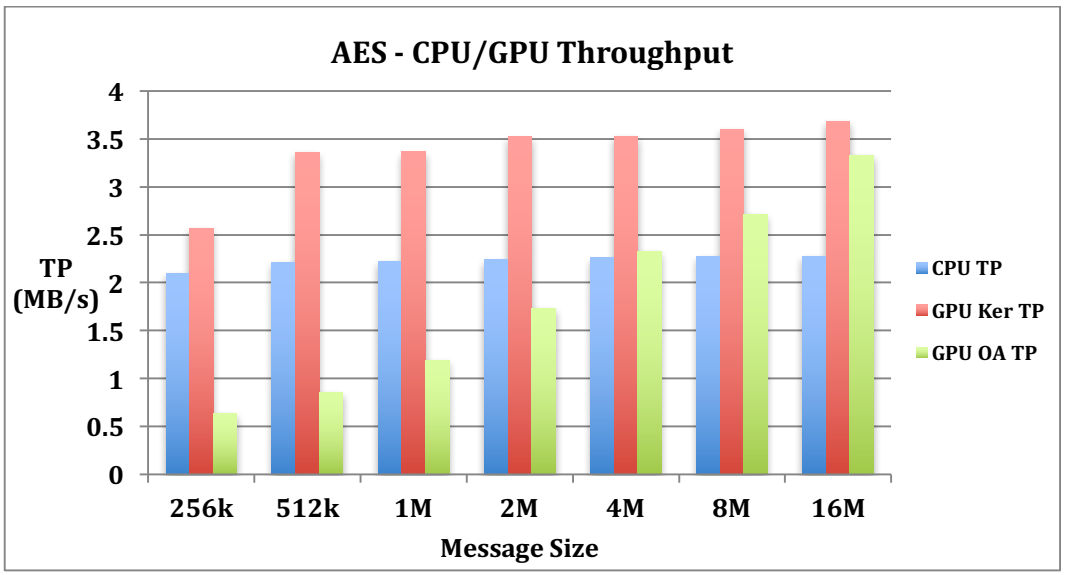

Figure 26. AES – CPU/GPU Throughput

The throughput of the CPU shows little scaling. This is because the CPU executes the same instructions for encrypting every 16 bytes of data. If the input message size increases N times, the execution time increases N times, which results in constant throughput in theory.

The GPU kernel performs better than the CPU, because the GPU implementation utilizes instruction-level parallelism and data parallelism are utilized. GPU kernel throughput increases when the input message size grows due to more parallel threads running in the GPU.

The overall GPU throughput increases gradually. The GPU overall throughput gets suppressed when the input message size is small due to

the GPU memory transfer overhead. When the input message size is 256kB, the GPU overall throughput is 4x lower than the GPU kernel throughput as shown in Figure 26. The negative impact of GPU transfer overhead is weaker for large input message sizes, because the growth in the kernel execution time is faster than the growth in the GPU memory transfer overhead.

4.4 CRC32 Performance

To test the performance of the CRC32 CPU implementation and the CRC32 GPU implementation, the programs calculate CRC32 code for a randomly generated message. The message is an array of randomly generated 8-bit data. The message sizes range from 16kB to 1MB increasing in powers of 2. The message is composed of many 1024-byte data segments. A CRC32 code is calculated for each data segment.

The throughputs of the CRC32 CPU implementation and the CRC32 GPU implementation are shown in Figure 27. Higher bars indicate higher performance. Y-axis is in log10 scale. The plot shows that the CPU throughput is 2x to 4x higher than the GPU kernel throughput and the GPU overall throughput depending on data size. The larger the input message size is, the higher the ratio between the CPU throughput and the GPU kernel/overall throughput.

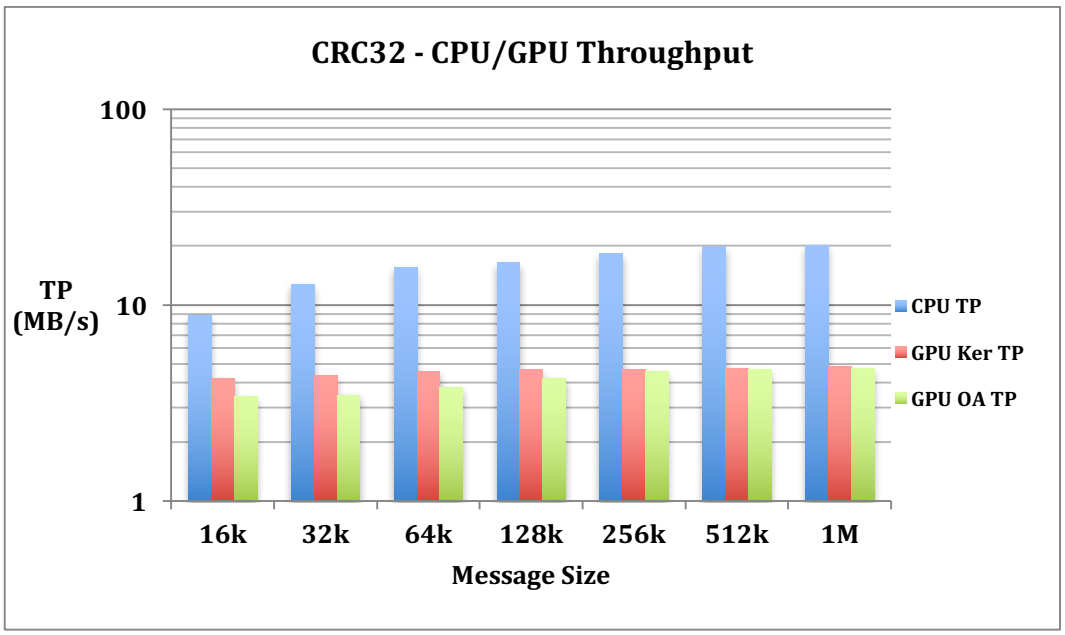

Figure 27. CRC32 – CPU/GPU Throughput

The throughput of the CPU increases gradually as the input message size gets bigger. The growth rate of the CPU throughput decreases, as the data size gets bigger, because the pipelines of the CPU get saturated.

The throughput of GPU kernel is lower than the CPU throughput in all test cases since there is no instruction-level parallelism and data parallelism in CRC32.

The GPU overall throughput is about the same as the GPU kernel throughput in most test cases. The kernel execution time is longer than the GPU memory transfer overhead, because CRC32 has poor instruction-parallelism and data parallelism.

4.5 Other Findings

Results of matrix multiplication and AES performance testing show that GPU memory transfer overhead generally can't be ignored when talking about GPU overall performance for small input data size. Thus, it is necessary to quantize GPU memory transfer overhead for the platform. An experiment was setup to measure the time duration for transferring data from system (host) DRAM to system (device) DRAM and from GPU DRAM to system DRAM with a GPU timer. Figure 28 shows the results of the characterization. Note that X-axis is in log10 scale.

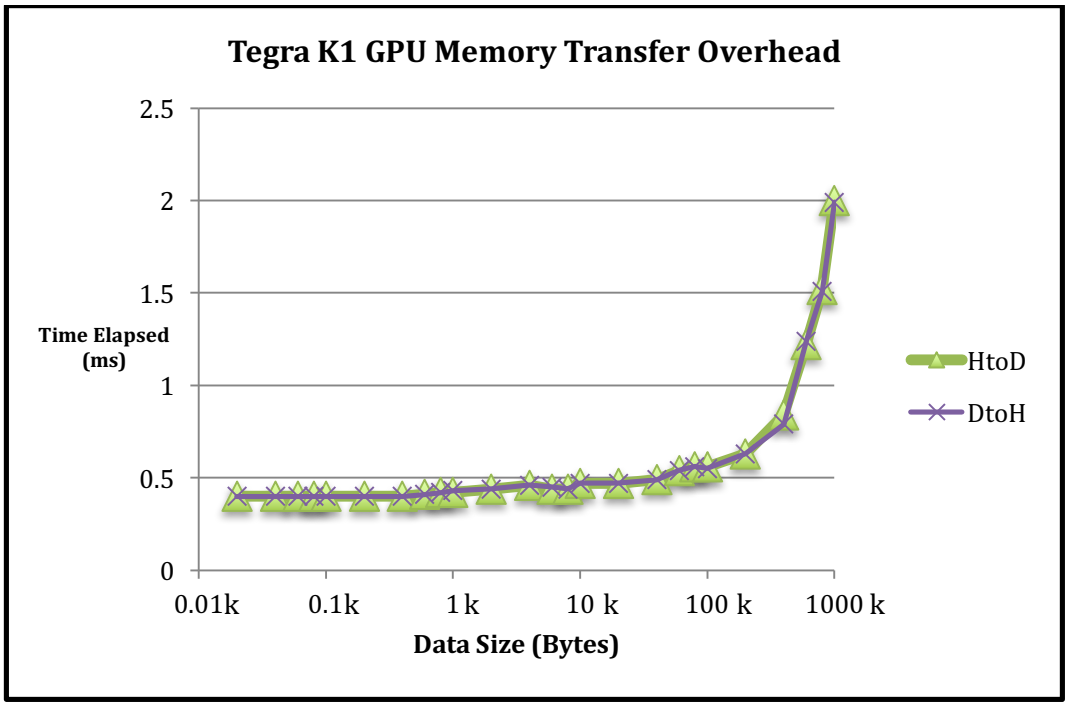

Figure 28. Tegra K1's GPU Memory Transfer Overhead Characterization

Figure 28 shows that the memory transfer time for under 100kB is close to 0.5ms. When the data size is larger than 100kB, the time starts to increase linearly.

4.6 Conclusion

The results from the experiments show that the GPU does not always have performance advantage over the CPU. For matrix multiplication and AES, the GPU is able to hide memory transfer overhead with parallel processing when the input data size is large, because the applications have good instruction-level parallelism or data parallelism. When the input data size is small, the GPU overall performance is suppressed by the GPU overhead. The GPU performs poorly in CRC32 because of the poor instruction-level parallelism and data parallelism in CRC32.

Chapter 5. Conclusion

This thesis has shown the performance of the GPU implementations and the GPU implementations of matrix multiplication, AES and CRC32 comparing to the CPU implementations when design effort is held minimum for both GPU implementations and CPU implementations. The performances of the implementations show that GPU memory transfer overhead has negative impact on the GPU overall throughput. Though the GPU kernel performs better than the CPU in the majority of the test cases in matrix multiplication and AES, GPU memory transfer overhead suppresses the GPU performance when GPU memory transfer overhead is considered. For matrix multiplication and AES, which have good instruction-level and data parallelism, the results recommend using the GPU instead of the CPU to get better performance when input data size is large. It is not recommended using GPU for CRC32 since CRC has poor parallelism.

GPU memory transfer overhead is quantized for NVIDIA Jetson K1 development platform. Knowing GPU memory transfer overhead can help programmers decide whether or not to use the GPU based on the input data size of certain applications.

Chapter 6. Future Work

This thesis analyzed the impact of GPU overhead on the GPU performance when the GPU acts like a slave device to the CPU. However, NVIDIA GPUs are able to be the master when communicating with external devices through NVIDIA GPUDirect. NVIDIA GPUDirect was introduced in 2013. The technology can eliminate GPU overhead in the following situations:

High Bandwidth, Low Latency Communication for GPU Accelerated Applications	
Accelerated Communication with Network & Storage Devices	• Direct access to CUDA memory for 3rd party devices • Eliminates unnecessary memory copies & CPU overhead • CUDA 3.1 and later
Peer-to-Peer Communication between GPUs	• Peer-to-Peer memory access, transfers & synchronization • Less code, higher programmer productivity • CUDA 4.0 and later
GPUDirect for Video	• Optimized pipeline for frame-based video devices • Low-latency communication with OpenGL, DirectX, or CUDA • CUDA 4.2 and later
RDMA	• Direct communication between GPUs across a cluster • Significantly increased MPISendRecv efficiency • CUDA 5.0 and later

Table 1. GPU Direct Accelerated Applications [16]

Applications shown in Table 5.1 involve interactions between GPU and other external devices. Traditionally, the CPU will collect the data from external devices through data bus. GPUDirect cuts off the CPU involvement and grants direct inter-system memory access for the GPU to minimize the memory transfer overhead. According to Nvidia, the memory transfer time can be reduced up to 45% [16]. Communicating with external devices introduces inter-system communication delay such as synchronization between the GPU system and the external device.

Characterizing the delay and analyzing the performance of GPUDirect enabled GPU can be a research topic.

Experiments in this thesis show that the CPU may have better performance than the GPU when input data size is small though the application has good instruction-level parallelism or data parallelism. Thus, a heterogeneous programming model can be applied to some applications. Applications can be designed to run on the CPU when the data size is small and on the GPU when the data size is large. However, the CPU does not stall when the GPU is running. It is possible that the GPU finishes processing while the CPU is busy, which leads to delay of transferring data from GPU DRAM to CPU DRAM. If the delay is not deterministic or too big, the heterogeneous approach may lead to poor performance. The delay caused by CPU-GPU synchronization when the CPU has a high amount of workload needs to be studied.

BIBLIOGRAPHY

[1] G. Singer, “The History of the Modern Graphics Processor,” TechSpot, 2013. [Online]. Available at: http://www.techspot.com/article/650-history-of-the-gpu/. [Accessed: Oct-2015].

[2] “Parallel Programming and Computing Platform | CUDA | NVIDIA|NVIDIA,” Parallel Programming and Computing Platform | CUDA | NVIDIA|NVIDIA. [Online]. Available at: http://www.nvidia.com/object/cuda_home_new.html. [Accessed: Feb-2015].

[3] F. Abi-Chahla, “Nvidia's CUDA: The End of the CPU? - Introduction,” Tom's Hardware, 2008. [Online]. Available at: http://www.tomshardware.com/reviews/nvidia-cuda-gpu,1954.html. [Accessed: Nov-2015].

[4] A. Lal Shimpi, B. Klug, and V. Gowri, “The iPhone 5 Review,” AnandTech, 2012. [Online]. Available at: http://www.anandtech.com/show/6330/the-iphone-5-review/4. [Accessed: 2015].

[5] R. Smith, “Apple A8X's GPU - GXA6850, Even Better Than I Thought,” AnandTech, Nov-2014. [Online]. Available at: http://www.anandtech.com/show/8716/apple-a8xs-gpu-gxa6850-even-better-than-i-thought. [Accessed: May-2015].

[6] Whitepaper NVIDIA Tegra K1 A New Era in Mobile Computing," nvidia.com, 2014. [Online]. Available at: whitepaper nvidia tegra k1 a new era in mobile computing. [Accessed: Jan-2015].

[7] C. Woolley, "CUDA Overview," http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/02-cuda-overview.pdf. [Online]. Available at: http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/02-cuda-overview.pdf. [Accessed: Jan-2015].

[8] "Programming Guide :: CUDA Toolkit Documentation," Programming Guide :: CUDA Toolkit Documentation, Jan-2015. [Online]. Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide. [Accessed: 2015].

[9] "Whitepaper NVIDIA's Next Generation CUDATM Compute Architecture: Fermi," nvidia.com, 2009. [Online]. Available at: https://www.nvidia.com/content/pdf/fermi_white_papers/nvidiafermicomputearchitecturewhitepaper.pdf. [Accessed: 2015].

[10] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," (Ieee Ispass) Ieee International Symposium On Performance Analysis Of Systems And Software.

[11] D. Le, J. Chang, X. Gou, A. Zhang, and C. Lu, "Parallel AES algorithm for fast Data Encryption on GPU," 2010 2nd International Conference on Computer Engineering and Technology.

[12] X. Cui, Y. Chen, and H. Mei, “Improving Performance of Matrix Multiplication and FFT on GPU,” 2009 15th International Conference on Parallel and Distributed Systems.

[13] J. Kubieziel, “Wie AES funktioniert,” kubieziel.de, 2008. [Online]. Available at: http://www.kubieziel.de/blog/archives/937-wie-aes-funktioniert.html. [Accessed: 2015].

[14] Monteiro, A. Dandache, A. M'sir, and B. Lepley, “A fast CRC implementation on FPGA using a pipelined architecture for the polynomial division,” ICECS 2001. 8th IEEE International Conference on Electronics, Circuits and Systems (Cat. No.01EX483).

[15] “Fast CRC32,” create.stephan-brumme.com, Oct-2011. [Online]. Available at: http://create.stephan-brumme.com/crc32/#branch-free. [Accessed: 2015].

[16] “NVIDIA GPUDirect Technology,” developer.download.nvidia.com, 2012. [Online]. Available at: http://developer.download.nvidia.com/devzone/devcenter/cuda/docs/gpudirect_technology_overview.pdf. [Accessed: 2015].

APPENDICES

APPENDIX A

Matrix Multiplication

main.cpp – page 1 of 2

```
//matrix multiplication
//main.cpp
//written by Fanfu Meng

#include<stdio.h>
#include<iostream>
#include<fstream>
#include <helper_cuda.h>
#include <helper_functions.h>
#include <cuda_runtime.h>
#include <stdlib.h>
#include "gpu_matrixMul.h"

#define size 64  //matrix dimension. matrices are square matrices
#define TestNum 50 //number of tests

//timer
static double seconds()
{

    #if defined(_WIN32) || defined(_WIN64)
        LARGE_INTEGER frequency, now;
        QueryPerformanceFrequency(&frequency);
        QueryPerformanceCounter  (&now);
        return now.QuadPart / double(frequency.QuadPart);
    #else
        timespec now;
        clock_gettime(CLOCK_REALTIME, &now);
        return now.tv_sec + now.tv_nsec / 1000000000.0;
    #endif
}
```

```
int main(int argc, char **argv)
{
    double startTime, duration;
    //a and b are input. c is the output
    int a[size][size], b[size][size], c[size][size];
    int i, j, k;


    //random number generator
    srand(2015);
    for(int x=0; x<size; x++)
    {
        for (int y=0; y<size; y++)
        {a[x][y]=int(rand()/100000);b[x][y]=int(rand()/100000);}

    }

    //cpu matrix multiplication

    for (int cnt = 0; cnt < TestNum; cnt++)
    {
        startTime = seconds();
        for(i=0;i<size;i++)
            {
                    for(j=0;j<size;j++)
                    {
                        c[i][j]=0;
                        for(k=0;k<size;k++)
                        {
                                //c[i][j]=c[i][j]+a[i][k]*b[k][j];
                        }
                    }
```

Matrix Mutiplication

main.cpp – page 2 of 2

```
            }

        duration  += (seconds() - startTime);
    }
    duration = duration/50;
    printf("\nCPU Throughput: %f kB/s\n", size/(duration*1000));



    //gpu
    int g_a[size*size], g_b[size*size], g_c[size*size];
    for (int x=0; x<size; x++)
    {
        for(int y=0; y<size; y++)
        {
            g_a[x*size+y] = a[x][y];
            g_b[x*size+y] = b[x][y];
        }
    }

    gpu_matrixMul(g_a,g_b,g_c,size);


}
```

gpu_matrixMul.h – page 1 of 1

```
//matrix multiplication
//gpu_matrixMul.h
//written by Fanfu Meng

#ifndef GPU_MATRIXMUL_H
#define GPU_MATRIXMUL_H
#include <stdint.h>
#include <math.h>

//gpu matrix multiplication function
extern "C" void gpu_matrixMul(int* a, int* b, int* c, unsigned int size);


#endif
```

gpu_matrixMul.cu – page 1 of 2

```
//matrix multiplication
//gpu_matrixMul.cu
//written by Fanfu Meng

#include <stdio.h>
#include <assert.h>
#include <helper_cuda.h>
#include <cuda_runtime.h>
#include <helper_functions.h>


__global__ void matrixMul_ker(int* a, int* b, int* c, int size)
{
    //declare shared mem
    __shared__ int s_a[64];
    __shared__ int s_b[64];

    //indexing
    int a_index = size*(blockIdx.x/size)+threadIdx.x;
    int b_index = (blockIdx.x%size)+threadIdx.x*size;

    //global mem to shared mem
    s_a[threadIdx.x] = a[a_index];
    __syncthreads();
    s_b[threadIdx.x] = b[b_index];
    __syncthreads();
     //multiplication
     s_a[threadIdx.x] = s_a[threadIdx.x] * s_b[threadIdx.x];
     __syncthreads();


    //addition - parallel reduction
    for(
        unsigned int s=1; s < blockDim.x; s *= 2) {
        if (threadIdx.x % (2*s) == 0)
        {
            s_a[threadIdx.x] += s_a[threadIdx.x + s];
        }
        __syncthreads();
    }


    //shared mem to global mem
    c[blockIdx.x] = s_a[0];

}

extern "C" void gpu_matrixMul(int* a, int* b, int* c, unsigned int size)
{

    //gpu global mem
    int* a_Buf;
    int* b_Buf;
    int* c_Buf;

    //thread config
    dim3 block(size, 1, 1);
    dim3 grid(size*size,1, 1);

    //gpu timer
    cudaEvent_t start, stop;
    float elapsedTime;
    float duration;
```

gpu_matrixMul.cu – page 2 of 2

```
    for (int cnt=0;cnt<50;cnt++) //50 tests
    {
        //gpu timer start - overall
        checkCudaErrors (cudaEventCreate (&start));
        checkCudaErrors (cudaEventCreate (&stop));
        checkCudaErrors (cudaEventRecord (start, 0));


        //gpu mem allocation
        checkCudaErrors(cudaMalloc((void **)&a_Buf, sizeof(int)*size*size));
        checkCudaErrors(cudaMalloc((void **)&b_Buf, sizeof(int)*size*size));
        checkCudaErrors(cudaMalloc((void **)&c_Buf, sizeof(int)*size*size));



      //host mem to device mem
      checkCudaErrors(cudaMemcpy(a_Buf, a, sizeof(int)*size*size, cudaMemcpyHostToDevice));
      checkCudaErrors(cudaMemcpy(b_Buf, b, sizeof(int)*size*size, cudaMemcpyHostToDevice));


      //timer start - kernel
      //checkCudaErrors (cudaEventCreate (&start));
      //checkCudaErrors (cudaEventCreate (&stop));
      //checkCudaErrors (cudaEventRecord (start, 0));

      //gpu kernel launch
      matrixMul_ker<<<grid, block>>>(a_Buf, b_Buf, c_Buf, size);

      //timer stop - kernel
      //checkCudaErrors (cudaEventRecord (stop, 0));
      //checkCudaErrors (cudaEventSynchronize (stop));
      //checkCudaErrors (cudaEventElapsedTime (&elapsedTime, start, stop));

      //device mem to host mem
      cudaMemcpy(c, c_Buf, sizeof(int)*size*size, cudaMemcpyDeviceToHost);


      //timer stop - overall
      checkCudaErrors (cudaEventRecord (stop, 0));
      checkCudaErrors (cudaEventSynchronize (stop));
      checkCudaErrors (cudaEventElapsedTime (&elapsedTime, start, stop));

      checkCudaErrors(cudaFree(a_Buf));
      checkCudaErrors(cudaFree(b_Buf));
      checkCudaErrors(cudaFree(c_Buf));
        duration += elapsedTime;
    }
    printf ("GPU Throughput: %f kB/s\n", size/(elapsedTime/50));


    cudaDeviceSynchronize();
    cudaDeviceReset();
}
```

main.cpp – page 1 of 6

```
//aes
//main.cpp
//written by Fanfu Meng
//based on Advanced Encryption Standard implementation in C By Niyaz PK
//license free and available at Downloaded from Website: www.hoozi.com


#include<stdio.h>
#include<iostream>
#include<fstream>
#include<stdlib.h>
#include "aes_gpu.h"

// The number of columns comprising a state in AES. This is a constant in AES. Value=4
#define Nb 4

// The number of rounds in AES Cipher. It is simply initiated to zero. The actual value is recieved in
the program.
int Nr=0;

// The number of 32 bit words in the key. It is simply initiated to zero. The actual value is recieved in
the program.
int Nk=0;

// in - it is the array that holds the plain text to be encrypted.
// out - it is the array that holds the key for encryption.
// state - the array that holds the intermediate results during encryption.
unsigned char in[16], out[16], state[4][4];
// The array that stores the round keys.
unsigned char RoundKey[240];

// The Key input to the AES Program
unsigned char Key[16];


unsigned char sbox[256] = {
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 };

unsigned char Rcon[255] = {
    0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,
    0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39,
    0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,
    0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,
    0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,
    0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,
    0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b,
    0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
    0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
```

main.cpp – page 2 of 6

```
    0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20,
    0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,
    0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,
    0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
    0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63,
    0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
    0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb  };


unsigned char getSBoxValue(int num)
{
    unsigned char sbox[256] = {
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 };
    return sbox[num];
}


void KeyExpansion()
{
    int i,j;
    unsigned char temp[4],k;

    // The first round key is the key itself.
    for(i=0;i<Nk;i++)
    {
        RoundKey[i*4]=Key[i*4];
        RoundKey[i*4+1]=Key[i*4+1];
        RoundKey[i*4+2]=Key[i*4+2];
        RoundKey[i*4+3]=Key[i*4+3];
    }


    // All other round keys are found from the previous round keys.

    while (i < (Nb * (Nr+1)))
    {

                    for(j=0;j<4;j++)
                    {
                        temp[j]=RoundKey[(i-1) * 4 + j];
                    }

                    if (i % Nk == 0)
                    {
                        // Function RotWord()

                      {
                          k = temp[0];
                          temp[0] = temp[1];
                          temp[1] = temp[2];
```

main.cpp – page 3 of 6

```
                        temp[2] = temp[3];
                        temp[3] = k;
                    }


                    // Function Subword()
                    {
                        temp[0]=getSBoxValue(temp[0]);
                        temp[1]=getSBoxValue(temp[1]);
                        temp[2]=getSBoxValue(temp[2]);
                        temp[3]=getSBoxValue(temp[3]);
                    }

                    temp[0] =  temp[0] ^ Rcon[i/Nk];

                }
                else if (Nk > 6 && i % Nk == 4)
                {

                    // Function Subword()
                    {
                        temp[0]=getSBoxValue(temp[0]);
                        temp[1]=getSBoxValue(temp[1]);
                        temp[2]=getSBoxValue(temp[2]);
                        temp[3]=getSBoxValue(temp[3]);
                    }
                }
                RoundKey[i*4+0] = RoundKey[(i-Nk)*4+0] ^ temp[0];
                RoundKey[i*4+1] = RoundKey[(i-Nk)*4+1] ^ temp[1];
                RoundKey[i*4+2] = RoundKey[(i-Nk)*4+2] ^ temp[2];
                RoundKey[i*4+3] = RoundKey[(i-Nk)*4+3] ^ temp[3];
                i++;
    }
}


void AddRoundKey(int round)
{
    int i,j;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            state[i][j] ^= RoundKey[round * Nb * 4 + i * Nb + j];

        }

    }


}
void SubBytes()
{
    int i,j;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            state[i][j] = getSBoxValue(state[i][j]);

        }
    }


}
```

main.cpp – page 4 of 6

```
void ShiftRows()
{
    unsigned char temp;

    // Rotate first row 1 columns to left
    temp=state[1][0];
    state[1][0]=state[1][1];
    state[1][1]=state[1][2];
    state[1][2]=state[1][3];
    state[1][3]=temp;

    // Rotate second row 2 columns to left
    temp=state[2][0];
    state[2][0]=state[2][2];
    state[2][2]=temp;

    temp=state[2][1];
    state[2][1]=state[2][3];
    state[2][3]=temp;

    // Rotate third row 3 columns to left
    temp=state[3][0];
    state[3][0]=state[3][3];
    state[3][3]=state[3][2];
    state[3][2]=state[3][1];
    state[3][1]=temp;

}

// xtime is a macro that finds the product of {02} and the argument to xtime modulo {1b}
#define xtime(x)   ((x<<1) ^ (((x>>7) & 1) * 0x1b))

// MixColumns function mixes the columns of the state matrix
void MixColumns()
{
    int i;
    unsigned char Tmp,Tm,t;
    for(i=0;i<4;i++)
    {
        t=state[0][i];
        Tmp = state[0][i] ^ state[1][i] ^ state[2][i] ^ state[3][i] ;
        Tm = state[0][i] ^ state[1][i] ; Tm = xtime(Tm); state[0][i] ^= Tm ^ Tmp ;
        Tm = state[1][i] ^ state[2][i] ; Tm = xtime(Tm); state[1][i] ^= Tm ^ Tmp ;
        Tm = state[2][i] ^ state[3][i] ; Tm = xtime(Tm); state[2][i] ^= Tm ^ Tmp ;
        Tm = state[3][i] ^ t ; Tm = xtime(Tm); state[3][i] ^= Tm ^ Tmp ;
    }
}

// Cipher is the main function that encrypts the PlainText.
void Cipher()
{
    int i,j,round=0;

    //Copy the input PlainText to state array.
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            state[i][j] = in[i*4 + j];

        }
    }


    AddRoundKey(0);
```

main.cpp – page 5 of 6

```
    // There will be Nr rounds.
    // The first Nr-1 rounds are identical.
    // These Nr-1 rounds are executed in the loop below.
    for(round=1;round<2;round++)//Nr;round++)
    {
        SubBytes();
        MixColumns();
        AddRoundKey(round);
    }

    // The last round is given below.
    // The MixColumns function is not here in the last round.
    SubBytes();
    ShiftRows();
    AddRoundKey(Nr);

    // The encryption process is over.
    // Copy the state array to output array.
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            out[i*4+j]=state[j][i];
        }
    }
}
static double seconds()
{
//#if defined(_MSC_VER) || defined(__CYGWIN__)
#if defined(_WIN32) || defined(_WIN64)
  LARGE_INTEGER frequency, now;
  QueryPerformanceFrequency(&frequency);
  QueryPerformanceCounter  (&now);
  return now.QuadPart / double(frequency.QuadPart);
#else
  timespec now;
  clock_gettime(CLOCK_REALTIME, &now);
  return now.tv_sec + now.tv_nsec / 1000000000.0;
#endif
}

const int data_sz = 16*1024*1024;
int main()
{
    unsigned char* val = new unsigned char [data_sz];
    unsigned char* k = new unsigned char [data_sz];
    unsigned char* output = new unsigned char [data_sz];

    double startTime, duration;


    Nr = 128;
    // Calculate Nk and Nr from the recieved value.
    Nk = Nr / 32;
    Nr = Nk + 6;



   srand(2009);
   for (int cnt=0; cnt<data_sz; cnt++)
   {val[cnt]=rand(); k[cnt]=rand();}




   // cpu aes
   for (int test_num=0; test_num <5; test_num++)
```

main.cpp – page 6 of 6

```
    {
        startTime = seconds();
        for (int itr=0; itr < data_sz/16; itr++)
        {
            for(int i=0; i<16; i++)
            {
                Key[i] = k[itr*16+i];
                in[i] = val[itr*16+i];
            }
            KeyExpansion();
            Cipher();
        }
        duration  += (seconds() - startTime);
    }

    printf("\nCPU Throughput: %.3fkB/s\n", data_sz/(duration*1000/5));


    aes_gpu(k, val, output, data_sz, sbox, Rcon);

    delete [] val;
    delete [] k;
    delete [] output;
}
```

aes_gpu.h – page 1 of 1

```
//aes
//aes_gpu.h
//written by Fanfu Meng
#include<stdio.h>
extern void aes_gpu(unsigned char* key, unsigned char* data, unsigned char* output, int data_size,
unsigned char* s_box, unsigned char* rcon);
```

aes_gpu.cu – page 1 of 4

```
//aes
//aes_gpu.cu
//written by Fanfu Meng
#include "aes_gpu.h"
#include <cuda_runtime.h>
#include <helper_cuda.h>
#include <helper_functions.h>

#define xtime(x)   ((x<<1) ^ (((x>>7) & 1) * 0x1b))


//aes kernel
__global__ void aes_ker(unsigned char* data, unsigned char* key, unsigned char* output, int data_size,
unsigned char* sbox, unsigned char* rcon)
{
    //shared memory
    __shared__ unsigned char s_sbox[256];
    __shared__ unsigned char s_Rcon[255];
    __shared__ unsigned char s_data[16];
    __shared__ unsigned char s_key[176];
    __shared__ unsigned char sft_temp[16];
    __shared__ unsigned char temp[4];
    __shared__ unsigned char tk[4];

    int tid = threadIdx.x;

     //load shared memory
     if (tid == 0)
     {
         for (int i=0; i<256; i++){
             s_Rcon[i] = sbox[i];
         }


     }
     else if(tid == 1){
         for (int i=0; i<255; i++){
             s_Rcon[i] = rcon[i];
         }
     }


     if(tid < 16){
         s_data[tid] = data[tid];
     }


     if (tid < 16){
         s_key[tid] = key[tid];
     }




    //add round key -- first round
    if(tid < 16){s_data[tid] ^= s_key[tid]; }
    __syncthreads ();


    //round 1 to 10
    for (int i=4; i<44; i++)
    {

        if (tid < 20){
            if (tid <4){
                temp[tid] = s_key[(i-1) * 4+tid];
```

aes_gpu.cu – page 2 of 4

```
                __syncthreads ();}


            if (i % 4 == 0){
                if (tid < 4){
                    tk[tid] = temp[tid+1];

                    if(tid==3)tk[tid] = temp[tid-3];}
                //sub bytes
                else if (tid>3){
                    s_data[tid-4] = s_sbox[s_data[tid-4]];
                }

                __syncthreads ();

                if(tid == 0){temp[0] =  temp[0] ^ s_Rcon[i/4];}
                //rotate
                else if(tid > 3 && tid < 8){sft_temp[tid-4] = s_data[tid-4];}
                else if (tid > 7 && tid <12){
                    if(tid == 11){sft_temp[tid-4]=s_data[4];}
                    else{sft_temp[tid-4] = s_data[tid-3];}}
                else if (tid>11 && tid<16){
                    if(tid>13){sft_temp[tid-4] = s_data[tid-6];}
                    else{sft_temp[tid-4] = s_data[tid-2];}}
                else if (tid>15 && tid<20){
                    if(tid>16){sft_temp[tid-4] = s_data[tid-5];}
                    else{sft_temp[tid-4] = s_data[tid-1];}}
                __syncthreads ();
                //mix collumn
                if (tid <4){
                    temp[tid] = s_sbox[tk[tid]];}
                else if(tid > 3 && tid < 8){
                    s_data[tid-4] ^= xtime(sft_temp[tid-4]^sft_temp[tid])^sft_temp[0]^sft_temp[4]^sft_temp
[8]^sft_temp[12];
                }
                else if(tid > 7 && tid < 12){
                    s_data[tid-4] ^= xtime(sft_temp[tid-4]^sft_temp[tid])^sft_temp[1]^sft_temp[5]^sft_temp
[9]^sft_temp[13];
                }
                else if(tid > 11 && tid < 16){
                    s_data[tid-4] ^= xtime(sft_temp[tid-4]^sft_temp[tid])^sft_temp[2]^sft_temp[6]^sft_temp
[10]^sft_temp[14];
                }
                else{
                    s_data[tid-4] ^= xtime(sft_temp[tid-4]^sft_temp[tid-16])^sft_temp[3]^sft_temp
[7]^sft_temp[11]^sft_temp[15];
                }


                __syncthreads ();


                if (tid <4){
                    s_key[i*4+tid] = s_key[(i-4)*4+tid] ^ temp[tid];}
                __syncthreads ();



            }
            if(tid < 16)
                {s_data[tid] ^= s_key[((i>>2))*16+tid-4]; }
                __syncthreads();

        }

    }

}
```

aes_gpu.cu – page 3 of 4

```
extern void aes_gpu(unsigned char* data, unsigned char* key, unsigned char* output, int data_size,
unsigned char* s_box, unsigned char* rcon)
{

    unsigned char* d_Buf;
    unsigned char* k_Buf;
    unsigned char* output_Buf;
    unsigned char* sbox_Buf;
    unsigned char* rcon_Buf;
    //gpu timer
    cudaEvent_t start, stop;
    float elapsedTime;

    //thread config
    dim3 grid(data_size/16, 1, 1);
    dim3 block(20, 1, 1);

    //gpu overall - start
    //checkCudaErrors (cudaEventCreate (&start));
    //checkCudaErrors (cudaEventCreate (&stop));
    //checkCudaErrors (cudaEventRecord (start, 0));


    //allocate device memory
    checkCudaErrors(cudaMalloc((void **)&d_Buf, sizeof(unsigned char)*data_size));
    checkCudaErrors(cudaMalloc((void **)&k_Buf, sizeof(unsigned char)*data_size));
    checkCudaErrors(cudaMalloc((void **)&output_Buf, sizeof(unsigned char)*data_size));
    checkCudaErrors(cudaMalloc((void **)&sbox_Buf, sizeof(unsigned char)*256));
    checkCudaErrors(cudaMalloc((void **)&rcon_Buf, sizeof(unsigned char)*255));
    //copy data from host to device
    checkCudaErrors(cudaMemcpy(d_Buf, data, sizeof(unsigned char)*data_size, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(k_Buf, key, sizeof(unsigned char)*data_size, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(sbox_Buf, s_box, sizeof(unsigned char)*256, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(rcon_Buf, rcon, sizeof(unsigned char)*255, cudaMemcpyHostToDevice));


    //gpu kernel - start
    checkCudaErrors (cudaEventCreate (&start));
    checkCudaErrors (cudaEventCreate (&stop));
    checkCudaErrors (cudaEventRecord (start, 0));

    //aes kernel launch
    aes_ker <<<grid, block>>>(d_Buf, k_Buf, output_Buf, data_size, sbox_Buf, rcon_Buf);


    //gpu kernel - stop
    checkCudaErrors (cudaEventRecord (stop, 0));
    checkCudaErrors (cudaEventSynchronize (stop));
    checkCudaErrors (cudaEventElapsedTime (&elapsedTime, start, stop));


    cudaMemcpy(output, output_Buf, sizeof(unsigned char)*data_size, cudaMemcpyDeviceToHost);

    //gpu overall - stop
    //checkCudaErrors (cudaEventRecord (stop, 0));
    //checkCudaErrors (cudaEventSynchronize (stop));
    //checkCudaErrors (cudaEventElapsedTime (&elapsedTime, start, stop));


    cudaMemcpy(output, output_Buf, sizeof(unsigned char)*data_size, cudaMemcpyDeviceToHost);

    //gpu overall - stop
    //checkCudaErrors (cudaEventRecord (stop, 0));
    //checkCudaErrors (cudaEventSynchronize (stop));
    //checkCudaErrors (cudaEventElapsedTime (&elapsedTime, start, stop));


    printf ("GPU Throughput : %f3kB/s\n", data_size/elapsedTime);

    checkCudaErrors(cudaFree(d_Buf));
    checkCudaErrors(cudaFree(k_Buf));
    checkCudaErrors(cudaFree(sbox_Buf));
    checkCudaErrors(cudaFree(rcon_Buf));
```

aes_gpu.cu – page 4 of 4

```
    checkCudaErrors(cudaFree(output_Buf));
    cudaDeviceSynchronize();
    cudaDeviceReset();
}
```

crc32.cpp – page 1 of 2

```
//crc32
//crc32.cpp
//written by Fanfu Meng
//based on bitwise branch-free implementation by Stephan Brumme
//see http://create.stephan-brumme.com/crc32/
//


#include <helper_cuda.h>
#include <helper_functions.h>
#include <cuda_runtime.h>
#include <stdlib.h>
#include "gpu_crc.h"


/// zlib's CRC32 polynomial
const uint32_t Polynomial = 0xEDB88320;


/// compute CRC32 (bitwise algorithm)
uint32_t crc32_bitwise(const void* data, size_t length, uint32_t previousCrc32 = 0)
{
  uint32_t crc = ~previousCrc32; // same as previousCrc32 ^ 0xFFFFFFFF
  const uint8_t* current = (const uint8_t*) data;

  while (length-- != 0)
  {

    crc ^= *current++;

    for (int j = 0; j < 8; j++)
    {
      // branch-free
      crc = (crc >> 1) ^ (-int32_t(crc & 1) & Polynomial);

    }

  }

  return ~crc;
}

//timer
static double seconds()
{
#if defined(_WIN32) || defined(_WIN64)
  LARGE_INTEGER frequency, now;
  QueryPerformanceFrequency(&frequency);
  QueryPerformanceCounter  (&now);
  return now.QuadPart / double(frequency.QuadPart);
#else
  timespec now;
  clock_gettime(CLOCK_REALTIME, &now);
  return now.tv_sec + now.tv_nsec / 1000000000.0;
#endif
}


///data size
const size_t NumBytes = 16*1024;


int main(int, char**)
{
      uint32_t* output = new uint32_t [NumBytes/1024];
```

crc32.cpp – page 2 of 2

```
      uint32_t randomNumber = 0x27121978;
      // initialize
      char* data = new char[NumBytes];
      //char data[NumBytes];
      for (size_t i = 0; i < NumBytes; i++)
      {
        data[i] = char(randomNumber & 0xFF);
        randomNumber = 1664525 * randomNumber + 1013904223;
      }

      // re-use variables
      double startTime, duration;
      uint32_t crc;

      for(int test_num=0; test_num < 50; test_num++){
          startTime = seconds();
          // bitwise
          for (int cnt=0; cnt < NumBytes/1024; cnt++){
            crc = crc32_bitwise(data+cnt*1024, 1024);
          }
          duration += (seconds() - startTime);
      }

      printf("CPU Throughput: %.3fkB/s\n", NumBytes/(duration/50)/1000);

      gpu_crc(data, output, NumBytes, NumBytes/1024);

      //delete[] data;
      delete[] output;
      return 0;
}
```

crc_gpu.h – page 1 of 1

```
//crc32
//gpu_crc.h
//written by Fanfu Meng
#ifndef GPU_CRC_H
#define GPU_CRC_H
#include <stdint.h>
#include <math.h>
typedef unsigned long ulong;

extern "C" void gpu_crc(char* data, uint32_t* output, int length, int o_length);

#endif
```

crc_gpu.cu – page 1 of 2

```
//crc
//gpu_crc.cu
//written by Fanfu Meng
//gpu_crc.cu


#include <helper_cuda.h>
#include "gpu_crc.h"
#include <stdint.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <cuda_runtime.h>
#include <helper_functions.h>


__device__ uint32_t gpu_crc32_bitwise(const void* data, size_t length, uint32_t previousCrc32 = 0)
{
  uint32_t crc = ~previousCrc32;
  const uint8_t* current = (const uint8_t*) data;
  uint32_t Polynomial = 0xEDB88320;
  while (length-- != 0)
  {

    crc ^= *current++;
     for (int j = 0; j < 8; j++)
     {
       crc = (crc >> 1) ^ (-int32_t(crc & 1) & Polynomial);
     }

   }
   return ~crc;
 }


 __global__ void crc32_ker(char* data, uint32_t* output, int length)
 {

     //shared memory
     __shared__ char s_data[1024];

     int bid = blockIdx.x;
     s_data[threadIdx.x] = data[bid*1024+threadIdx.x];
     //load shared memory
     //for (int i=0; i<1024; i++){
     //  s_data[i] = data[i];
     //}

     __syncthreads();

     if(threadIdx.x==0)
     {   output[bid] = gpu_crc32_bitwise(s_data, 1024); }
     //output[bid] = gpu_crc32_bitwise(s_data, 1024);
    __syncthreads();

}

extern "C" void gpu_crc(char* data, uint32_t* output, int d_length, int o_length)
{
    char* d_Buf;
    uint32_t* o_Buf;

    //thread config
    dim3 grid(d_length/1024, 1, 1);
    dim3 block(1024,1,1);
```

crc_gpu.cu – page 2 of 2

```
    //timer
    cudaEvent_t start, stop;
    float elapsedTime;
    float duration=0;

    for (int test_num=0; test_num<50; test_num++){
        //timer - overall start
        //checkCudaErrors (cudaEventCreate (&start));
        //checkCudaErrors (cudaEventCreate (&stop));
        //checkCudaErrors (cudaEventRecord (start, 0));

        //memory allocation
        checkCudaErrors(cudaMalloc((void **)&d_Buf, sizeof(char)*d_length));
        checkCudaErrors(cudaMalloc((void **)&o_Buf, sizeof(uint32_t)*o_length));

        //memory transfer
        checkCudaErrors(cudaMemcpy(d_Buf, data, sizeof(char)*d_length, cudaMemcpyHostToDevice));

        //timer - kernel start
        checkCudaErrors (cudaEventCreate (&start));
        checkCudaErrors (cudaEventCreate (&stop));
        checkCudaErrors (cudaEventRecord (start, 0));

        //kernel launch
        crc32_ker<<<grid, block>>>(d_Buf, o_Buf, 1024);

       //timer - kernel stop
        checkCudaErrors (cudaEventRecord (stop, 0));
        checkCudaErrors (cudaEventSynchronize (stop));
        checkCudaErrors (cudaEventElapsedTime (&elapsedTime, start, stop));

        //memory transfer
        cudaMemcpy(output, o_Buf, sizeof(uint32_t)*o_length, cudaMemcpyDeviceToHost);

        //timer - overall stop
        //checkCudaErrors (cudaEventRecord (stop, 0));
        //checkCudaErrors (cudaEventSynchronize (stop));
        //checkCudaErrors (cudaEventElapsedTime (&elapsedTime, start, stop));
        duration += elapsedTime;
    }
    printf("GPU Throughput: %.3fkB/s\n", d_length/(duration/50));

    checkCudaErrors(cudaFree(d_Buf));
    checkCudaErrors(cudaFree(o_Buf));
    cudaDeviceSynchronize();
    cudaDeviceReset();

}
```