Optimizing the CPU-GPU Interactions to Improve Performance

University of Maryland Independent Research

Electrical and Computer Engineering (Fall 2016)

Emanuelle Crespi

TABLE OF CONTENTS:

Optimizing the CPU-GPU Interactions to Improve Performance

Abstract:

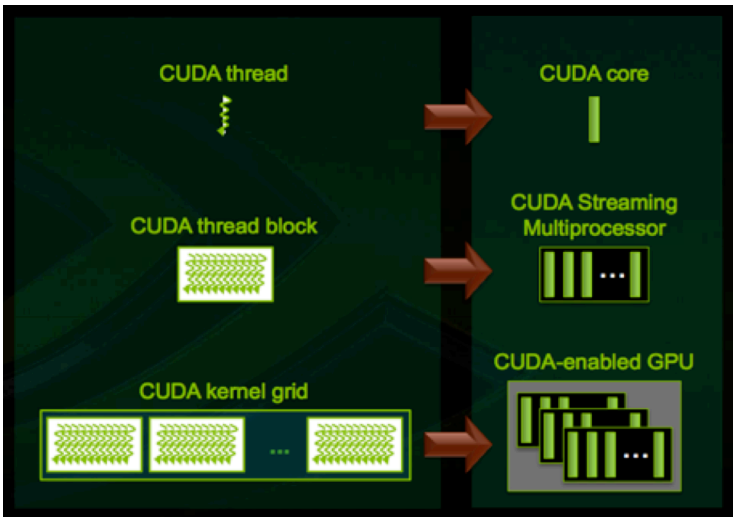
Little has been done to make the most capable use of the GPGPU in a computer architecture with a common memory region between the CPU and GPU, leaving the CPU to handle most of the general workload. With NVIDIA's Jetson TK1 as the host device, this research observes a correlation between speed up and power dissipation when general-purpose tasks have been processed amongst the multiprocessor cores of the GPU. The methodology describes the process used in our experiment to collect data relevant to both execution time and current draw of the device during runtime of the test applications.

My partner, Tolga Keskinoglu, and I focus on two aspects of this research; one on power draw of the device during execution, and the other on the speedup acquired during runtime. The conclusions discuss the overhead of runtime, with it's relation to power draw on the Jetson TK1 when asynchronous events have been scheduled to run on the GPU. A particular strategy, "race to finish", is discussed as a way of handling heavy general purpose tasks very quickly at the cost of higher power dissipation for a shorter amount of time. It seems that this may be a useful approach for embedded operating systems to implement in future designs, improving overall performance in the long run.

Introduction:

NVIDIA's CUDA API

The graphics-processing unit (GPU) has been immensely useful for handling a large workload of tasks in parallel to improve CPU efficiency. This has been done by devoting each unit of the motherboard to its own memory pool for processing as the data is sent from the CPU to the GPU and the result is passed back to the CPU. More computer architectures have included a general-purpose graphics-processing unit (GPGPU) for handling simpler tasks such as matrix multiplication to save the CPU some time. NVIDIA's embedded chip design, the Tegra K1, shares a common memory region between CPU and GPU and allows for explicit GPU programming for parallelizable applications.



Thread blocks can cooperate within SMs and can share results with each other for multiprocessing [6].

These types of applications require a unique understanding of the CUDA (Compute Unified Device Architecture) parallel computing API. The basis for this API is to make use of the stream multiprocessors (SM) stored within CUDA capable GPUs and take advantage of the GPU pipeline by establishing an optimal number of the blocks and threads to a parallelizable task.

Objective

Given that parallelizable computationally extensive code allows for the possibility of faster execution time, the CPU-GPU interaction can be further optimized to acquire a significant speedup for general-purpose tasks. This research pays particular attention to the Tegra K1 processor by observing the Jetson TK1 as an embedded device that makes efficient use of the GPGPU. The focus is to examine the performance of the board to see how speedup and power draw is affected during runtime of computationally extensive algorithms. The conclusions drawn from the data gathered will help in deciding particular circumstances where the embedded GPU should be used for general-purpose computing.

Methodology:

NOTE:

We began this research with the assumption that most of the equipment and workspace would be easily accessible to us as undergraduate affiliates to the Electrical and Computer Engineering department at University of Maryland. Unfortunately, finding accommodations for this space proved to be significantly difficult and we ended up improvising our setup for an ideal workspace. We eventually settled for an SSH connection, allowing access to the terminal on the device through Tolga Keskinoglu's home network.

The intention was to develop code and cross-compile onto the device using NVIDIA Nsight™ software via remote connection, but we ended up moving on from this idea due to time constraints and compatibility issues with the Jetson TK1 32-bit architecture. It seems most of this software is more agreeable with the Jetson TX1 64-bit architecture. Nonetheless the final setup for this required a significant amount of time to allow us to safely access the device remotely over the Internet.

The very first step required installation of the appropriate packages onto the Jetson TK1. The operating system of choice is the pre-installed Linux for Tegra (L4T) Ubuntu version 14.04. Installation of CUDA 6.5 is necessary for running and compiling CUDA capable code on the Jetson TK1 device. To become more familiar with the source code, the NVIDIA programmer's manuals (available in references section) have been used as a solid reference for understanding CUDA programming semantics and logical flow whenever necessary.

The equipment used for this entire process included the following…

- Jetson TK1 board
- HDMI Monitor and cable
- Mouse, keyboard, and USB hub
- 2.1mm barrel jack (to connect from supply to board)
- Agilent™ Dual Display DC Supply w/ Probes
- Breadboard (for connecting the probes w/ the power supply and the board)
- Stopwatch and Video camera (for tracking time and current readings)

This setup for this research depended highly on having complete access to the Jetson TK1 board with it's capabilities as a CUDA device. CUDA capabilities allow us to schedule function calls to execute in parallel on the many processors on the GPU. It is important to stay true to the standard of NVIDIA programmer manual and identify these as kernel calls, as they are labeled __global__ type functions in the code section. The scheduling is done in the compilation phase where we compile the GPU parallelizable

source code with '*NVCC*' rather than using the GNU compiler 'GCC' that is commonly used for serial applications to run on the CPU.
All of the code developed throughout the project is provided in the appendix section of this paper with an explanation of the purpose and intention of the program. The serial tasks to be considered as good test cases were chosen to fit all of the three following criteria. The serial application should be...

1. Parallelizable
2. Representative of real-world general-purpose tasks
3. Scalable so that the process could run for long enough to gather accurate data.

The first step of the development involved programming C code that would be a compatible candidate for parallelization on the GPU. A setup of some data structure in serial, mostly arrays, followed by a function call passing a pointer to that data fits this requirement. This kept the operations very similar to one another with respect to execution on either the CPU or GPU. The modifications to parallel code involved setting up space on both the CPU and GPU by using the C library '*malloc(...)*' and the CUDA library '*cudaMalloc(....)*' respectively. Upon populating the memory with some arbitrary data, it is possible to process in one of two ways, synchronous and asynchronous.

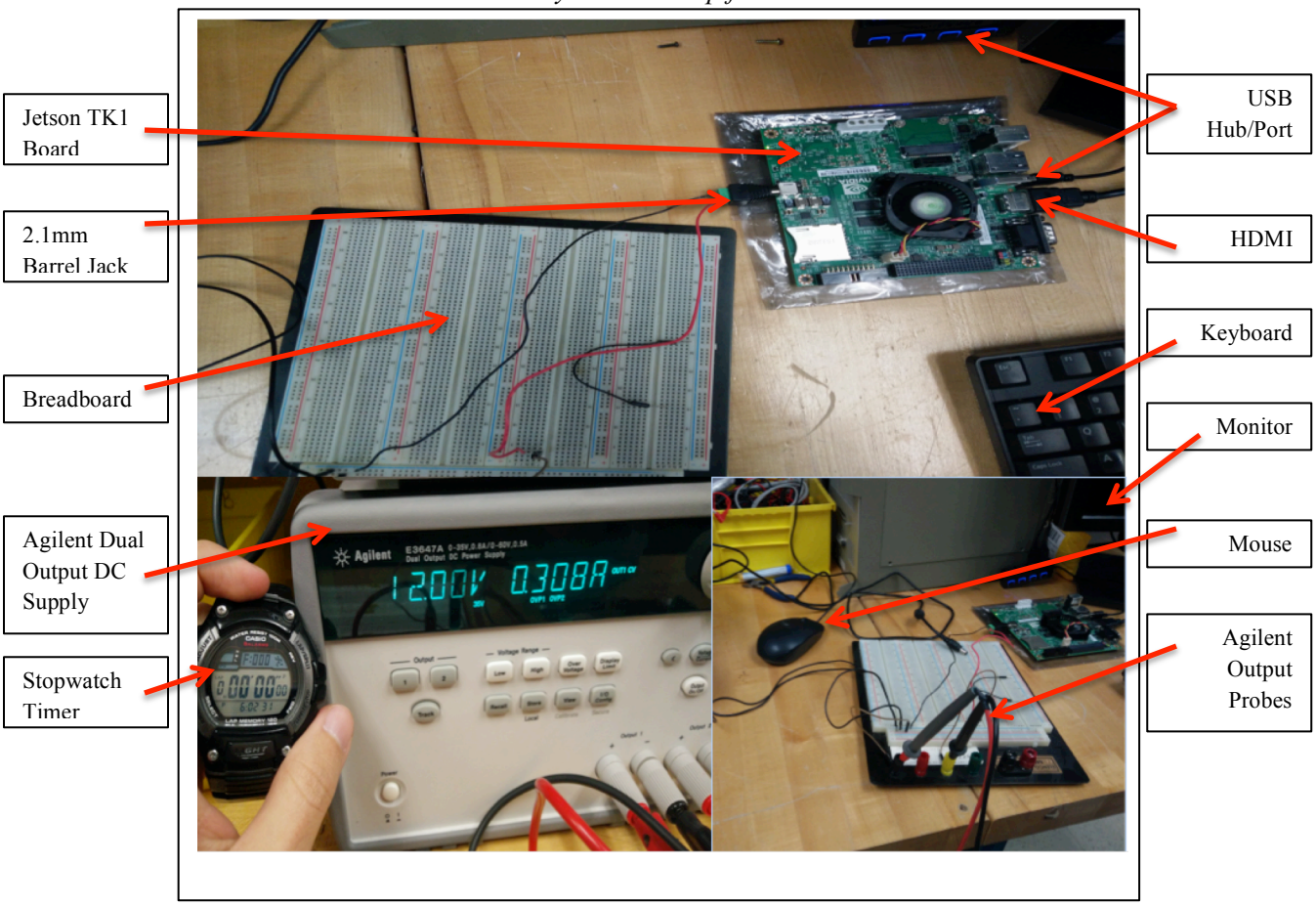
The synchronous tests ran without any participation of the GPU towards a result, with C code running on the CPU while the GPU remains idle. The next phase of the test parallelizes this code by using *cudaMalloc(...)* to allocate space on the device. When the vectored data has been populated on the host (CPU) the data is copied onto the device (GPU) using '*cudaMemcpy(....)*' to send data over the bus to the device. This is followed by a kernel call on the GPU for the cores to execute in parallel. Parallelization of the C code ended up as a series of multiple trial and error runs until a successful result could be identified. For certain applications, it became crucial to optimize memory sharing amongst the blocks and threads of the GPU and separate them into streams, adding synchronous behavior between those streams as needed.

NOTE:
It is also important to mention that calls to cudaMemcpy contribute quite a bit of overhead with regard to execution time for a large amount of vectored data. The performance tests acknowledge this by timing the execution of the task itself rather than timing the entire runtime of the program as a whole.

Testing serial code on the CPU for runtime is possible when using the time.h library available in the GNU. The timer prints the execution time of the task in milliseconds, using s*tart(...)* and *stop(...)* functions as a simple yet accurate way to time the task function. It is not ideal to track the runtime for asynchronous events using the time.h library. This library only tracks the runtime of the serial code, so it becomes complicated trying to keep track of the runtime for each kernel call running asynchronous to the serial tasks. Luckily, the CUDA library gives the programmer access to an event structure, which allows access to timing of kernel calls on the device. This became the standard for timing the parallel code and comparing the result to its serial counterpart.

The final phase of this research involves taking these testing applications and performing a "side channel based" analysis on the power consumption during execution of these applications. The Jetson TK1 runs at approximately 12V, however the current can fluctuate depending on the power dissipation of the board. This is modeled by Ohm's law $P = I * V$. Considering the board to have internal resistance R with current $I = V/R$ implies that the internal resistance of the Jetson TK1 must be changing. This is what is affecting the power consumption of the device and allows one to model the change in current with respect to time. A power supply and a 2.1mm barrel jack were used to power the Jetson TK1 at 12V while the current is measured (in Amperes) during code execution. The current is measured using an Agilent™ power supply, displaying both the voltage and current being supplied to the board.

Power Analysis Lab Setup for Jetson TK1



LIMITATIONS:

There are some assumptions and limitations to be mentioned for both phases of data collection in this experiment. We assume the device timers used for the test code are accurate enough for us to average over 4 successful trials.

Data in the appendix has been gathered through a setup using a camera, power supply display and a stopwatch. The camera records the power supply's display with the corresponding time on the stopwatch. Live execution time has been timed on a stopwatch so that the data could be gathered in 500 millisecond steps. Human error plays a significant role in this process, but was unavoidable due to the inability of the power supply to dump output data electronically. This is why we chose largely scalable code to minimize this error as much as possible. The time for a human to perceive some change printed on screen during checkpoints of the execution phase is assumed to give a deviation of 0.5 seconds. Improvements to this process can be made based on the prior assumption, and finding some way to dump both the current output and the corresponding time in a file (possibly with a Ny-DAC) would allow for more accurate readings.

Results:

Four algorithms were tested on the Jetson TK1 to observe speedup of sequential (CPU) executable code to its parallel (GPU) counterpart. The following results correspond to tests 1-4 listed below.

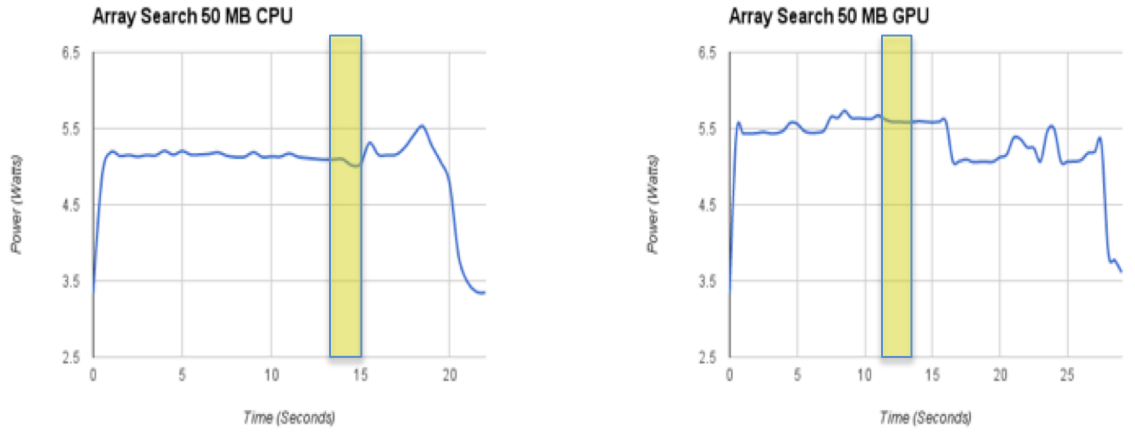
1) Array search for a particular character value
2) Testing a large number to see if it is prime
3) Vector Addition of large data arrays
4) Hashing a large vector of data to a smaller one

The final phase, involving power analysis during runtime, indicates behavior that is particularly interesting when the GPU is active on this embedded platform. All of the relevant source code/data has been included in the appendices with a brief explanation in the header and explicit comments describing the logical flow of the source code.

Array Search

We first consider a linear method for performing a search on a large array of data where the processing has been done on both CPU and GPU for comparison. The title of this test is array search. Two versions of this test were run. One has been run on a 50MB file and the other on a 100MB file. Table 1 shows the results of 4 runs on both the CPU and GPU. On average, the array search for a 50MB file was 6x faster than the sequential search. During this search, the power draw on the device was more efficient for the serial application. The CPU code consumed 4.96W and the CUDA (GPU) code consumed 5.28W (Figure 1). The shaded regions are rough estimates of the execution phase where a comparison on the average power can be made.

Figure 1 (Array Search 50MB Power Analysis):

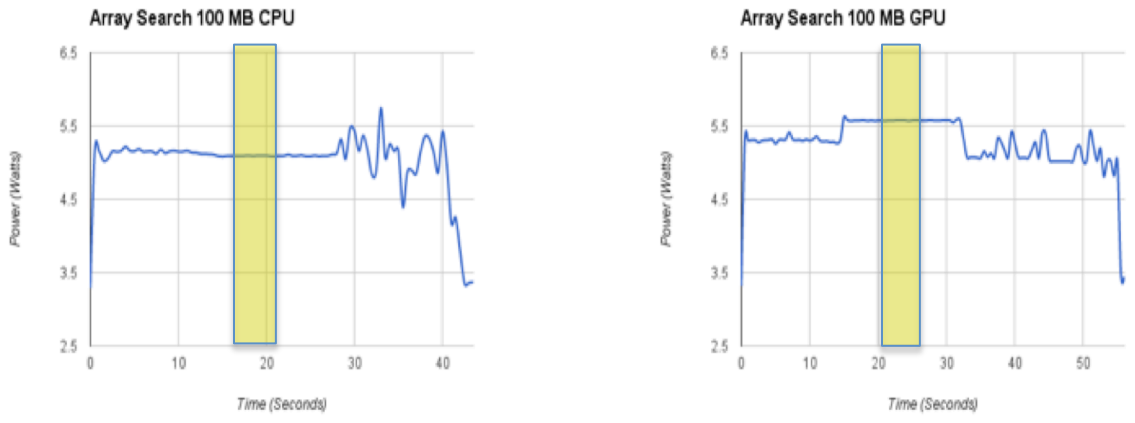


During the task execution phase the CPU consumes ~0.5W less power than the GPU

On the 100MB file, we see similar results. On average, the GPU code executes 10x faster than the CPU code. There is less power consumption on the CPU (5.00W) processing than there is on the GPU (5.24W). Table 2 shows the runtime results for the task execution with the corresponding graph (Figure 2) for the power analysis below.

Table 2: Array Search (100MG)		
RUN	Sequential Search (s)	CUDA Array Search (s)
1	0.956178	0.092537
2	0.960265	0.102327
3	0.96046	0.09946
4	0.922259	0.095043
AVG:	0.9497905	0.09734175
SPEEDUP (sequential/cuda)		9.757277838

Figure 2 (Array Search 100MB Power Analysis):



The highlighted region shows the time interval for the execution phase of the task for both CPU and GPU.

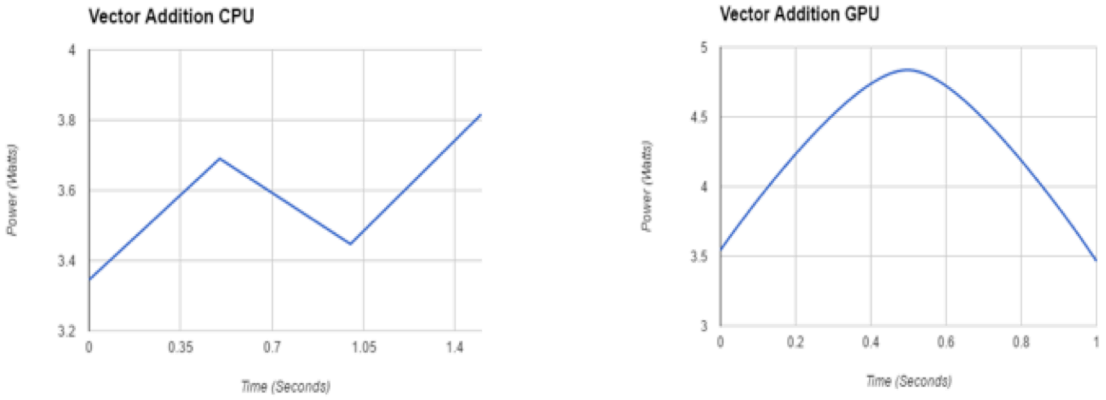
Vector Addition

The next test considers a vector addition of very large arrays. In this test, the vector length is relatively smaller than expected with a size of 413124 bytes. However, we are still able to observe some interesting data on the potential speedup and power dissipation for a large set of data. Vector addition of this size on the GPU ended up 9x faster than the CPU code on average (Table 3). The power dissipation of the Jetson TK1 was a fairly similar 3.95W compared to the CPU implementation drawing 3.45W (Figure 3). Again at the cost of a bit more power, speedup is possible.

Table 3: Vector Addition (size = 413124 ints)

RUN	Vector Add Sync	Vector Add Async
1	0.026034	0.002252
2	0.014946	0.002247
3	0.018861	0.002452
4	0.021143	0.002255
Avg:	0.020246	0.0023015
SPEEDUP		8.796871605

Figure 3 (Vector Addition on 413124 integer array*):*



In the hundred thousands range, the GPU barely saves as much energy as the CPU.

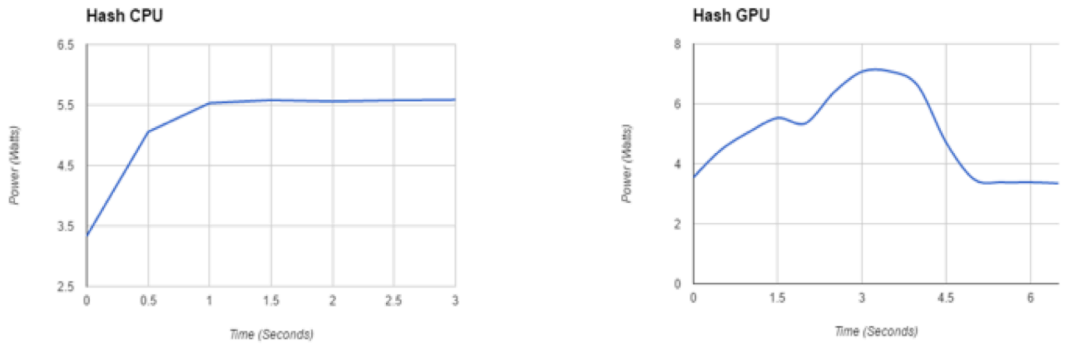
Hash

Making use of a hash algorithm tests a more real-world example where hashing may be necessary to filter data into a group of buckets. This algorithm takes an even length vector of size 2n and maps values at even indices to a smaller vector of size n. This particular test uses a vector of 123374234 integers to accomplish this task. Although the total CUDA runtime was not as efficient as the CPU, there was a notable speedup in executing the task itself. The speedup obtained was a significant 23x faster than the CPU task execution (Table 4). The power consumed by the serial code peaked at 5.5W and the CUDA code maxed at about 6.5W. Overall the average power consumption for the CUDA code seemed to have been better than the CPU code (Figure 4).

Table 4: Hash (size = 123374234 ints)

RUN	HASH SYNC	HASH ASYNC (the real deal)
1	1.020174	0.045201
2	1.016587	0.045134
3	1.016152	0.045022
4	1.015897	0.045118
Avg:	1.0172025	0.04511875
SPEEDUP		22.54500623

Figure 4 (Simple Hash on 123374234 integer array*):*

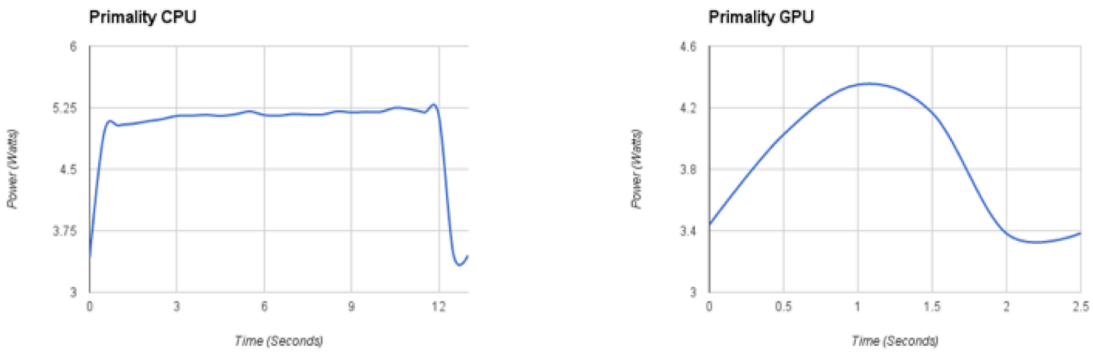


The CPU code seems to saturate during execution while the GPU code hits some peak level and drops.

Testing Large Numbers for Prime

Our most interesting test takes a value, rather than a vector of values to operate over on the GPU. The CUDA runtime was significantly faster than the CPU version with an average 12x speedup, and the power dissipation of the CUDA code is 3.79W in comparison to the CPU consuming 4.97W (Figure 5).

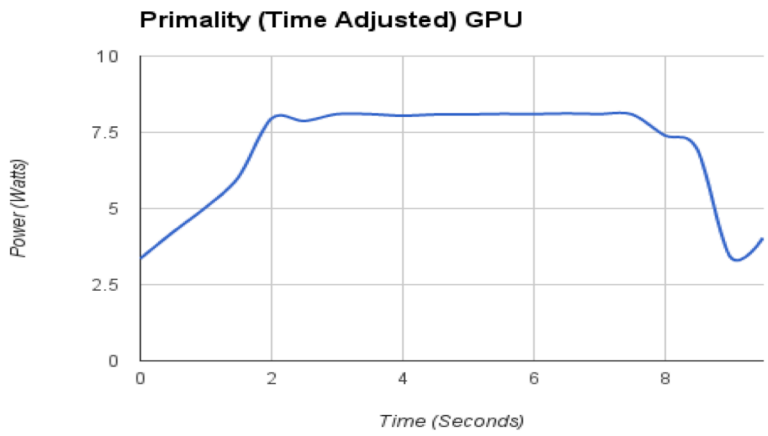
Figure 5 (Prime Test of Large Power Analysis):



The CUDA run ran significantly faster than the CPU code, consuming less power over its duration

To get a better idea of the runtime relationship to power usage, it was necessary to run the algorithm as a CUDA kernel for about 20 CPU cycles. The speedup over 20 repetitions (r = 20) is still faster, running at 1.5x the speed of the serial version. We can verify this in Table 5 below. Figure 5 of the previous page shows that a "race to finish" approach may be the best to consider for significantly parallelizable code. Figure 6 shows the data obtained for r = 20 repetitions.

Figure 6 (Prime Test of Large Power Analysis – 20 cycles):



The GPU consumed 6.86W on average. This amounts to 17% of the energy consumed by the CPU in Figure 5.

Table 5: Prime Testing (n = 900000006)			
RUN	CPU Algorithm (r = 1)	CUDA Algorithm (r = 1)	CUDA Algorithm (r = 20)
1	12.122737	1.17732	8.277523
2	12.092529	1.191744	8.192113
3	12.100882	0.664808	7.727055
4	12.089686	1.11968	8.244487
Avg:	12.1014585	1.038388	8.1102945
SPEEDUP when r = 1		11.65408161	
SPEEDUP when r = 20		1.49211086	

We can see a 12x and 1.5x speedup in both CUDA algorithms compared to the serial CPU code

Conclusion:

Possibilities for More Speedup

Based on the results of each experiment we can see that speedup of execution time, for the task itself, is very possible for general-purpose computations on the CPU. In fact, there seems to be a "sweet spot" for every parallelizable algorithm that we have encountered so far in this lab. This is mostly related to the size of the data set. It becomes exceedingly more difficult to parallelize C code into CUDA code because we usually need to find a set number of streams and an acceptable division of the threads per block to make the most efficient use of the GPU during runtime. The array search required a unique division for the streams required to properly operate over 50MB-100MB arrays, allowing for a successful speedup. However, it was a bit difficult to draw concrete conclusions on the power analysis for that data.

There is also the cost incurred when sending the data over the bus to the GPU memory region with *cudaMemcpy(...)*. We can see that the time it took to send and receive data on the GPU slowed down overall performance of the program when compared to the CPU. Finding a better strategy to send and receive data from the GPU may be a good more beneficial in the long run. For example, the Tegra K1 processor was very useful for performing tasks such as array search when the array pointer is defined as a pointer in *page-locked memory*. The CUDA pipeline allowed us to perform concurrent accesses to this vectored data, and dynamically allocate smaller junks of data during the processing of other kernels. This approach sped up runtime for larger data sets but failed to perform the search for data sets exceeding the range of 100MB. There were mysterious null values being copied back from the GPU into our result, so we settled for 50M-100MB for testing purposes.

Minimizing Average Power Consumption (Race to Finish Strategy)

The trend in power dissipation when the GPU is active suggests that heavier tasks are expected to consume more power on the device in the general case. The Jetson TK1 consistently showed a larger power draw when executing tasks such as array searching, vector addition and hashing elements of a large array. The only exception was the test for prime numbers where we see a much smaller average power draw for determining that the largest factor of the number 900000006. Although we would like to minimize the cost of power for general-purpose tasks executed on the GPU, it is also possible that some tasks cannot fully utilize the capabilities of the GPU on the Jetson TK1.

The prime test shows us that a strategy ('race-to-finish') where the most efficient number of streams is chosen sets up the processing for an even distribution amongst the threads and blocks of the multiprocessors on the GPU. Finding a way for the algorithm to run in as little time as possible allows for a smaller average in power consumption over the time interval if the power draw is large. For other parallelizable cases, this may be the best approach as the data set becomes exponentially larger. Accounting for the cost incurred on the power draw while considering the execution time may be a useful approach to these problems.

Room for Future Improvement

Due to limited access to labs, and schedule conflicts, our approach to this data collection could use improvements on accuracy. Rather than use a stopwatch for time increments, there are other methods including software that could be used to dump the current data read on the power supply into a file with the corresponding time increments. The Ny-Daq may prove useful for this task as it comes with software for plotting current readings over a time interval.

References:

1. "A view of the parallel computing landscape". *Commun. ACM*. 52 (10): 56–67

2. <http://elinux.org/Jetson/Graphics_Performance>
"Jetson/Graphics Performance." *ELinux.org*. Web. 06 Sept. 2016.

4. <http://www.umiacs.umd.edu/research/GPU/publications.html>
J. H. Jung and D. P. O'Leary. Cholesky decomposition and linear programming on a gpu. In *Workshop on Edge Computing Using New Commodity Architectures (EDGE)*, Chapel Hill, North Carolina, May 2006.

4. <http://www.nvidia.com/object/cuda_home_new.html>
"Parallel Programming and Computing Platform | CUDA | NVIDIA | NVIDIA." *Parallel Programming and Computing Platform | CUDA | NVIDIA | NVIDIA*. Web. 18 Aug. 2016.

5. <http://www.umiacs.umd.edu/research/GPU/research.html>
"Research Areas." *Maryland CPU-GPU Cluster Infrastructure*. Web. 18 Aug. 2016.

6. <https://www.nersc.gov/assets/Uploads/CUDAIntrosouthard.pdf>
"National Energy Research Scientific Computing Center." *National Energy Research Scientific Computing Center CUDA Intro*. N.p., n.d. Web. 19 Dec. 2016.

Appendix A:

```
/**
 * Array Search Serial (Fall 2016):
 *
 * Members:
 * Emanuelle Crespi, Tolga Keskinoglu
 *
 * This test implements an array search discussed in the methodology section
 * of Optimizing CPU-GPU Interactions.
 *
 * The following code makes use of the fucntion call search(int n, char *data, char *out, char c)
 * to perform a sequential search for char c amongst a large vector of characters.
 *
 * The result is a one-one mapping of the data array with 1s and 0s in the out array at
 * corresponding indices where the character has been found.  The output is written to
 * the file 'array_search_result.txt' for validation with 'parrallel_array_search_result.txt'
 * when running the executable for array_search.cu
 *
 * The output of the performance is displayed in seconds.
 * The performance results are to be compared with the performance of array_search.cu
 *
 */

//System includes
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>

/*
 * Performs a sequential search for char c in array data both of size n
 * The output 1 or 0 is written in out to indicate 'found' and 'not-found' respectively
 * search( n, ['a','b','d','c','d','e',...], result, 'd') ==> result =
['0','0','1','0','1','0',...]
 */
void search(int n, char *data, char *out, char c){
        int i;
        for (i = 0; i < n; i++){
                if (data[i] == c){
                        out[i] = '1';
                }else{
                        out[i] = '0';
                }
        }
}

int main(){
        //Initializations
        FILE *fp_data, *fp_out;
        char *data, *out, c;
        struct timeval tv_start, tv_stop, tv_diff;
        int s_data = 0, j, i = 0;

        //sys/time.h types
        clock_t start, diff;

        //printf("Computing file size...\n");
        if (!(fp_data = fopen("../../file.txt", "r")))
                perror("failed to read file\n");

        while( fscanf(fp_data,"%c",&c) != EOF ){
                s_data++;
        }

        /* Allocate necessary space for buffer */
        //printf("Mallocing %d bytes of data on CPU...\n", s_data);
        data = (char *)malloc(sizeof(char)*s_data);
        out = (char *)malloc(sizeof(char)*s_data);
```

```
        fseek(fp_data, 0, 0);

        /* Read file into buffer */
        //printf("Reading data into buffer...\n");
        for( j = 0; fscanf(fp_data,"%c",&data[j]) != EOF; j++ ){}

        /* Time the search algorithm */
        //printf("Executing search on CPU...\n");
        gettimeofday(&tv_start, NULL);
        printf("Running...\n");
        search(s_data, data, out, 'D');

        gettimeofday(&tv_stop, NULL);
        timersub(&tv_stop, &tv_start, &tv_diff);

        printf("Performance= %ld.%06ld sec\n", (long int) tv_diff.tv_sec, (long int)
tv_diff.tv_usec);

        //printf("Writing result to array_search_result.txt...\n");
        fp_out = fopen("array_search_result.txt", "w");

        for (j = 0; j < s_data; j++){
                if( i == 32 ){
                        fprintf(fp_out, "%c\n", out[j]);
                        i = 0;
                }else{
                        fprintf(fp_out, "%c", out[j]);
                        i++;
                }
        }

        /* Cleanup */
        free(data); free(out);
        fclose(fp_data); fclose(fp_out);
        return 0;
}
```

Appendix B:

```
/**
 * Array Search with CUDA (Fall 2016):
 *
 * Members:
 * Emanuelle Crespi, Tolga Keskinoglu
 *
 * This test implements an array search discussed in the methodology section
 * of Optimizing CPU-GPU Interactions.
 *
 * The following code makes use of the kernel call search(int n, char *data, char *out, char c)
 * to perform a parallel search for char c amongst a large vector of characters.
 *
 * The result is a one-one mapping of the data array with 1s and 0s in the out array at
 * corresponding indices where the character has been found.  The output is written to
 * the file 'parallel_array_search_result.txt' for validation with 'array_search_result.txt'
 * when running the executable for array_search.c
 *
 * While the overhead of executing 'cudaMemCpy(...)' slows down execution time,
 * the performance of the parallel search itself is significantly faster than it's
 * serial counterpart.
 *
 * The output of the performance is displayed in seconds for verification.
 *
 * References:
 * NVIDIA CUDA C Programming Guide Version 3.2
 */

// System includes
#include <stdio.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <time.h>

// Jetson TK1 has device capability 1.x allowing 1024 threads/block
// We also indicate a threshold of 67108864 for vectored data
#define THREADS_PER_BLOCK 1024
#define THRESHOLD 67108864

//Identify failures
#define FILE_OPEN_FAIL -1
#define MALLOC_FAIL -2


/*
 * Indicates the task search to be performed on the GPU
 * for char c in array data both of size n
 *
 * The output 1 or 0 is written in out to indicate 'found' and 'not-found' respectively
 * Results are written to device memory and must be fetched back from out for verification.
 * search( n, ['a','b','d','c','d','e',...], result, 'd') ==> result =
['0','0','1','0','1','0',...]
 */
__global__ void search(int n, char *data, char *out, char c){
        int i =  blockIdx.x*blockDim.x + threadIdx.x;
        if(i < n){
                if (data[i] == c){
                        out[i] = '1';
                }else{
                        out[i] = '0';
                }
        }
}

int main(){
        FILE *fp_data, *fp_out;
        char *data, c;
        char *d_data, *d_out;
        int s_data = 0, j = 0, i = 0;
```

```
        int flag = 0;
        cudaError_t error;
        //printf("Computing file size...\n");

        if (!(fp_data = fopen("../../file.txt", "r"))){
                perror("failed to open file.txt\n");
                return FILE_OPEN_FAIL;
        }

        while( fscanf(fp_data,"%c",&c) != EOF ){
                s_data++;
        }

        int rem = s_data % THRESHOLD;
        int sections = (THRESHOLD+s_data-1)/THRESHOLD;

        //printf("Mallocing %d bytes of data on CPU...\n", s_data);

        /* Allocate necessary space for host buffer */
        cudaMallocHost(&data, sizeof(char)*s_data);

        /* Allocate necessary space for device buffer */
        //printf("Mallocing %d bytes of data on GPU...\n", s_data);
        cudaMalloc( (void **) &d_data, sizeof(char)*s_data);
        cudaMalloc( (void **) &d_out, sizeof(char)*s_data);

        fseek(fp_data, 0, 0);

        /* Read file into buffer */
        //printf("Reading data into buffer...\n");
        for( j= 0; fscanf(fp_data,"%c",&data[j]) != EOF; j++ ){     }

        /* Identify our streams */
        cudaStream_t stream[sections];
        for (int j = 0; j < sections; j++){
                cudaStreamCreate(&stream[j]);
        }

        /* Time the search algorithm */
        //printf("Executing search on GPU...\n");


        if( rem == 0 ){
                flag = 0;
        }else{
                flag = 1;
        }

        /*******************************for testing
purposes****************************************

*********************************************************************************************/
        cudaDeviceSynchronize();

        // Allocate CUDA events that we'll use for timing
        cudaEvent_t start;
        error = cudaEventCreate(&start);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to create start event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        cudaEvent_t stop;
        error = cudaEventCreate(&stop);

        if (error != cudaSuccess)
        {
```

```
                fprintf(stderr, "Failed to create stop event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        // Record the start event
        error = cudaEventRecord(start, NULL);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to record start event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }
        printf("Running...\n");
        // Execute the kernel
        for(j = 0; j < sections-flag; j++){
                cudaMemcpyAsync(d_data + j * THRESHOLD, data + j * THRESHOLD,
                     THRESHOLD, cudaMemcpyHostToDevice, stream[j]);
                search<<<(THRESHOLD+THREADS_PER_BLOCK-
1)/THREADS_PER_BLOCK,THREADS_PER_BLOCK,0,stream[j]>>>(THRESHOLD,
                                                                data+(j*THRESHOLD),
d_out+(j*THRESHOLD), 'D');
                cudaStreamSynchronize(stream[j]);
                cudaMemcpyAsync( data + j * THRESHOLD, d_out + j * THRESHOLD,
                     THRESHOLD, cudaMemcpyDeviceToHost, stream[j]);
        }


        /* Define and run stream for remainder */
        cudaMemcpyAsync(d_data + j * THRESHOLD, data + j * THRESHOLD,
                     rem, cudaMemcpyHostToDevice, stream[j]);

        search<<<(rem+THREADS_PER_BLOCK-
1)/THREADS_PER_BLOCK,THREADS_PER_BLOCK,0,stream[j]>>>(rem,
                                                        data + j * THRESHOLD, d_out + j * THRESHOLD,
'D');
        cudaStreamSynchronize(stream[j]);

        cudaMemcpyAsync( data + j * THRESHOLD, d_out + j * THRESHOLD,
                 rem, cudaMemcpyDeviceToHost, stream[j]);

        // Record the stop event
        error = cudaEventRecord(stop, NULL);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to record stop event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        // Wait for the stop event to complete
        error = cudaEventSynchronize(stop);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to synchronize on the stop event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        float msecTotal = 0.0f;
        error = cudaEventElapsedTime(&msecTotal, start, stop);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to get time elapsed between events (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }
```

```
        // Compute and print the performance
        float msecPerisPrime = msecTotal / 1;

        printf( "Performance= %.06f sec\n", msecPerisPrime/1000.0 );
        /*****************************************************************************************
***
         **************************** for testing purposes
**************************************/

        /* Destroy streams */
        for (int j = 0; j < sections; j++){
                cudaStreamDestroy(stream[j]);
        }

        /* Copy result back to host */
        //printf("Writing result to 'parallel_array_search_result.txt'...\n");

        if ( !(fp_out = fopen("parallel_array_search_result.txt", "w")) ){
                perror("failed to open results file\n");
                return FILE_OPEN_FAIL;
        }

        //output data to file
        for (j = 0; j < s_data; j++){
                if( i == 32 ){
                        fprintf(fp_out, "%c\n", data[j]);
                        i = 0;
                }else{
                        fprintf(fp_out, "%c", data[j]);
                        i++;
                }


        }

        /* Cleanup */
        cudaFreeHost(data);
        cudaFree(d_data); cudaFree(d_out);
        fclose(fp_data); fclose(fp_out);

        return 0;
}
```

Appendix C:

```
/**
 * One Way Hash Test Serial (Fall 2016):
 *
 * Members:
 * Emanuelle Crespi, Tolga Keskinoglu
 *
 * This test implements a simple hash from a space of size 2n --> n
 * discussed in the methodology section of Optimizing CPU-GPU Interactions.
 *
 * The following code makes use of the kernel call hash(char *f, char *h, int n)
 * to perform a sequential hash of elements f --> h with corresponding indices 2i --> i
 *
 * The result is a mapping of the data within f to the data within h
 * The output is verified before the program terminates to see that every
 * element at index 2i of f is indeed at index i in h
 *
 * The output of the performance is displayed in seconds.
 * The performance results are to be compared with the performance of hash.cu
 *
 */

// System includes
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>

//this hash requires even length arrays
#define EVEN_NUM 123374234


// this function performs a hash of n elements into an array h of size n/2
// h is considered a bucket holding all of the elements at even indices in f
void hash(char *f, char *h, int n) {
        int i;

        for( i = 0; i < n; i++){
                h[i] = f[2*i];
        }
}

int main(void) {
        //r can be modified to produce as much overhead as needed during testing
        int two_n = EVEN_NUM, i, r = 50;
        struct timeval tv_start, tv_stop, tv_diff;
        char *f, *h;

        if ( two_n % 2 ){
                //printf("NO NO NO!!! Even numbers only please.\n");
                exit(EXIT_FAILURE);
        }

        //set up space for arrays
        f = calloc(sizeof(char), two_n + 1);
        h = calloc(sizeof(char), two_n/2 + 1);

        //Populate data into array
        for (i = 0; i < two_n; i++) {
                f[i] = (char) ((i % 94) + 33);
        }

        //Run test
        printf("Running...\n");
        gettimeofday(&tv_start, NULL);
        for (i = 0; i < r; i++) {
                hash(f, h, two_n/2);
        }
        gettimeofday(&tv_stop, NULL);
```

```
        timersub(&tv_stop, &tv_start, &tv_diff);
        printf("Performance= %ld.%06ld sec\n", (long int) tv_diff.tv_sec, (long int)
tv_diff.tv_usec);

        //Validate for correctness (takes extra time but avoids overhead from file writing)
        for (i = 0; i < two_n/2; i++) {
                if (h[i] != f[2*i]) {
                        printf("WRONG!\n");
                        return 1;
                }
        }

        return 0;
}
```

Appendix D:

```
/**
 * One Way Hash with CUDA (Fall 2016):
 *
 * Members:
 * Emanuelle Crespi, Tolga Keskinoglu
 *
 * This test implements a simple hash from a space of size 2n --> n
 *
 * The following code makes use of the kernel call hash(char *f, char *h, int n)
 * to perform a parallel hash of elements f --> h with corresponding indices 2i --> i
 *
 * The result is a mapping of the data within f to the data within h
 * The output is verified before the program terminates to see that every
 * element at index 2i of f is indeed at index i in h
 *
 * We can see that there is a significant speedup in comparison to the time it takes
 * to perform the hash in the serial code.
 *
 * The output of the performance is displayed in seconds.
 * The performance results are to be compared with the performance of hash.c
 *
 */

// System includes
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>


// Jetson TK1 has device capability 1.x allowing 1024 threads/block
// We also indicate EVEN_NUM as the vector size since this hash requires even length arrays
#define THREADS_PER_BLOCK 1024
#define EVEN_NUM 123374234


__global__ void hash(char *f, char *h, int n) {
        int i = blockIdx.x*blockDim.x+threadIdx.x;

        if( i < n ){
                h[i] = f[2*i];
        }
}

int main(void) {
        int two_n = EVEN_NUM, i, r=50;
        char *f, *h, *d_f,*d_h;
        cudaError_t error;

        if ( two_n % 2 ){
                printf("NO NO NO!!! Even numbers only please.\n");
                exit(EXIT_FAILURE);
        }

        //printf("Malloc space on CPU (f,h)");
        f = (char *)calloc(sizeof(char), two_n);

        if( f == NULL ){
                fprintf(stderr,"Failed to allocate %d bytes for f.",two_n);
                exit(EXIT_FAILURE);
        }

        h = (char *)calloc(sizeof(char), two_n/2);

        if( h == NULL ){
                fprintf(stderr,"Failed to allocate %d bytes for h.",two_n/2);
                exit(EXIT_FAILURE);
        }
```

```
        /* Identify our streams */
        //printf("Malloc space on GPU (d_f,d_h)\n");
        error = cudaMalloc((void **)&d_f, sizeof(char) * two_n);

        if( error != cudaSuccess ){
                fprintf(stderr,"Failed to cudaMalloc %d bytes for d_f.",two_n);
                exit(EXIT_FAILURE);
        }

        error = cudaMalloc((void **)&d_h, sizeof(char) * two_n/2);

        if( error != cudaSuccess ){
                fprintf(stderr,"Failed to cudaMalloc %d bytes for d_h.",two_n/2);
                exit(EXIT_FAILURE);
        }

        //populate data into array
        //printf("Generate vectored data (Size=%d bytes)\n",two_n);
        for (i = 0; i < two_n; i++) {
                f[i] = (char) ((i % 94) + 33);
        }

        //send data over the bus
        //printf("Send data to GPU\n");
        error = cudaMemcpy( d_f, f, two_n, cudaMemcpyHostToDevice);

        if (error != cudaSuccess)
        {
                printf("cudaMemcpy (d_f,f) returned error code %d, line(%d)\n", error, __LINE__);
                exit(EXIT_FAILURE);
        }

        /************************** Setup for testing ***********************************/
        //printf("Run kernel code \n");


        cudaDeviceSynchronize();

        // Allocate CUDA events that we'll use for timing
        cudaEvent_t start;
        error = cudaEventCreate(&start);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to create start event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        cudaEvent_t stop;
        error = cudaEventCreate(&stop);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to create stop event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        // Record the start event
        error = cudaEventRecord(start, NULL);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to record start event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }
        printf("Running...\n");
        //run kernel
        for( i = 0; i < r; i++){
```

```
                hash<<<(two_n/2+THREADS_PER_BLOCK-
1)/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_f,d_h,two_n/2);
        }

        // Record the stop event
        error = cudaEventRecord(stop, NULL);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to record stop event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        // Wait for the stop event to complete
        error = cudaEventSynchronize(stop);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to synchronize on the stop event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        float msecTotal = 0.0f;
        error = cudaEventElapsedTime(&msecTotal, start, stop);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to get time elapsed between events (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }


        /*************************************************************************************
***
         **************************** for testing purposes
*************************************/

        //send data over the bus
        error = cudaMemcpy( h, d_h, sizeof(char)*two_n/2, cudaMemcpyDeviceToHost);

        if (error != cudaSuccess)
        {
                printf("cudaMemcpy (h,d_h) returned error code %d, line(%d)\n", error, __LINE__);
                exit(EXIT_FAILURE);
        }
        //printf("Done.\n");

        //validate for correctness
        for (i = 0; i < two_n/2; i++) {
                if (h[i] != f[2*i]) {
                        //printf("index %d FAILED!\n", i);
                        exit(EXIT_FAILURE);
                }
        }

        // Compute and print the performance
        float msecPerhash = msecTotal / 1;
        printf( "Performance= %.06f sec\n", msecPerhash/1000.0 );

        free(f); free(h);
        cudaFree(d_f); cudaFree(d_h);

        cudaDeviceReset();

        return 0;
}
```

Appendix E:

```
/**
 * Primality Test Serial (Fall 2016):
 * Members:
 * Emanuelle Crespi, Tolga Keskinoglu
 *
 * This test implements an algorithm to test for primality discussed in the methodology section
 * of Optimizing CPU-GPU Interactions.
 *
 * The following code makes use of the function call is_prime(int n, char *factor, char *prime)
 * to perform a sequential search for some factor of the value n.
 *
 * n is purposely chosen as the value n=900000006 for comparison of efficiently determining
 * a factor in the is_prime.cu code
 *
 * The output of the performance is displayed in seconds for verification.
 */

// System includes
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>

// Performs a sequential search for a factor of the value n
// The algorithm is purposely inefficient to always perform
// the worst case search of length n.  This is to assist in
// performing a power analysis during runtime.
//
// When a multiple is found, prime is written to 1 and facter
// is written as the largest multiple
// Both are meant to be read & verified by the caller
void is_prime(int n, int *factor, int *prime) {
        int i;
        *prime = 1;
        for (i = 2; i < n; i++) {
                if (n % i == 0) {
                        *prime = 0;
                        *factor = i;
                }
        }
}

int main(void) {
        //r can be modified to produce as much overhead as needed during testing
        int prime, n=900000006, r=1, i, factor;
        struct timeval tv_start, tv_stop, tv_diff;

        /* Super inefficient primality testing */
        /* with extra computation for runtime */
        gettimeofday(&tv_start, NULL);
        for (i = 0; i < r; i++) {
                is_prime(n, &factor, &prime);
        }
        gettimeofday(&tv_stop, NULL);
        timersub(&tv_stop, &tv_start, &tv_diff);

        printf("Performance= %ld.%06ld sec\n", (long int) tv_diff.tv_sec, (long int)
tv_diff.tv_usec);

        //Confirm whether the value n is prime
        if (prime == 1) {
                printf("%d is prime.\n", n);
        } else {
                printf("%d is NOT prime, %d is a factor!\n", n, factor);
        }

        return 0;
}
```

Appendix F:

```
/**
 * Primality Testing with CUDA (Fall 2016):
 *
 * Members:
 * Emanuelle Crespi, Tolga Keskinoglu
 *
 * This test implements an algorithm to test for primality discussed in the methodology section
 * of Optimizing CPU-GPU Interactions.
 *
 * The following code makes use of the kernel call is_prime(int n, char *factor, char *prime)
 * to perform a parallel search for some factor of the value n.  The kernel calls are
 * seperated into r=20 streams amongst the multi-stream processors of the CUDA compatible GPU.
 * This allows us to gather data via power analysis to find a relationship between
 * execution speed and power dissipation for the Jetsion TK1.
 *
 * While the overhead of executing executing r streams slows down execution time,
 * the performance of the parallel search itself is significantly faster than it's
 * serial counterpart. We can see a significant improvement in the output displayed during
runtime
 * when r = 1.
 *
 * The output of the performance is displayed in seconds for verification.
 *
 * References:
 * NVIDIA CUDA C Programming Guide Version 3.2
 */

// System includes
#include <stdio.h>
#include <time.h>

// Jetson TK1 has device capability 1.x allowing 1024 threads/block
#define THREADS_PER_BLOCK 1024

// Performs a parallel search for a factor of the value n
// When a multiple is found, prime is written to 1 and facter
// is written as the multiple to be read & verified by the caller
//
// The values are written to device memory and must be recovered by the caller
__global__ void is_prime(int n, int *d_factor, int *d_prime) {

        int i = blockIdx.x*blockDim.x + threadIdx.x;
        if (i > 1 && i < n && n % i == 0) {
                *d_prime = 0;
                *d_factor = i;
        }
}

int main(void) {
        //r can be modified to produce as much overhead as needed during testing
        int *prime, *d_prime, n=900000006, r=20, *factor, *d_factor;
        cudaError_t error;

        /* Generate space on the device */
        prime = (int *)calloc(1, sizeof(int));
        *prime = 1;
        cudaMalloc((void **)&d_prime, sizeof(int));
        cudaMemcpy(d_prime, prime, sizeof(int), cudaMemcpyHostToDevice);
        factor = (int *)calloc(1, sizeof(int));
        cudaMalloc((void **)&d_factor, sizeof(int));

        /* Launch encrypt() kernel on GPU */
        cudaStream_t stream[r];
        for (int i = 0; i < r; i++ )
                cudaStreamCreate(&stream[i]);

        /*******************************for testing
purposes*****************************************
```

```
*********************************************************************************************/
        cudaDeviceSynchronize();

        // Allocate CUDA events that we'll use for timing
        cudaEvent_t start;
        error = cudaEventCreate(&start);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to create start event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        cudaEvent_t stop;
        error = cudaEventCreate(&stop);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to create stop event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        // Record the start event
        error = cudaEventRecord(start, NULL);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to record start event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        // Execute the kernel
        // NEED TO PUT STREAMS FOR R VALUE IN HERE
        for( int i = 0; i < r; i++){
                is_prime<<<(n + THREADS_PER_BLOCK -
1)/THREADS_PER_BLOCK,THREADS_PER_BLOCK,0,stream[i]>>>(n, d_factor, d_prime);
                cudaStreamSynchronize(stream[i]);
        }

        // Record the stop event
        error = cudaEventRecord(stop, NULL);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to record stop event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        // Wait for the stop event to complete
        error = cudaEventSynchronize(stop);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to synchronize on the stop event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        float msecTotal = 0.0f;
        error = cudaEventElapsedTime(&msecTotal, start, stop);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to get time elapsed between events (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
```

```
        }

        // Compute and print the performance
        float msecPerisPrime = msecTotal / 1;

        printf( "Performance= %.06f sec\n", msecPerisPrime/1000.0 );
        /*****************************************************************************************
***
         **************************** for testing purposes
*************************************/

        /* Destroy streams */
        for (int j = 0; j < r; j++){
                cudaStreamDestroy(stream[j]);
        }

        /* Copy results back to host */
        error = cudaMemcpy(prime, d_prime, sizeof(int), cudaMemcpyDeviceToHost);

        if (error != cudaSuccess)
        {
                printf("cudaMemcpy (prime,d_prime) returned error code %d, line(%d)\n", error,
__LINE__);
                exit(EXIT_FAILURE);
        }

        error = cudaMemcpy(factor, d_factor, sizeof(int), cudaMemcpyDeviceToHost);

        if (error != cudaSuccess)
        {
                printf("cudaMemcpy (factor,d_factor) returned error code %d, line(%d)\n", error,
__LINE__);
                exit(EXIT_FAILURE);
        }


        /*    IS IT PRIME???   */
        if (*prime == 1) {
                printf("%d is prime.\n", n);
        } else {
                printf("%d is NOT prime, %d is a factor!\n", n, *factor);
        }


        /* Cleanup */
        free(prime); free(factor);
        cudaFree(d_prime); cudaFree(d_factor);

        return 0;
}
```

Appendix G:

```
/**
 * Vector Addition Test Serial (Fall 2016):
 *
 * Members:
 * Emanuelle Crespi, Tolga Keskinoglu
 *
 * This test implements a sequential vector addition: C = A + B
 * discussed in the methodology section of Optimizing CPU-GPU Interactions.
 *
 * The following code makes use of the function call vectorAdd( int n, int *A, int *B, int *C )
 * to perform a vector addition of two arrays A and B of length n
 *
 * The output is verified before the program terminates to see that every
 * element of index i has C[i] == A[i] + B[i]
 *
 * The output of the performance is displayed in seconds.
 * The runtime results are to be compared with the performance of vector.cu
 *
 */

// System includes
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <sys/time.h>

//dimension for vectored data
#define XDIM 413124

// Performs a sequential vector addition of two arrays v1 and v2 of length n
// The result of the addition is left within the corresponding indices
// of v: v[i] = v1[i] + v2[i]
vectorAdd( int n, int *v1, int *v2, int *v ){
        int i;
        for( i = 0; i < n; i++){
                v[i] = v1[i] + v2[i];
        }
}

int main(void){
        //XDIM can be modified to perform larger additions
        int i, v1[XDIM], v2[XDIM], v[XDIM];
        struct timeval tv_start, tv_stop, tv_diff;

        //Populate data into arrays v1 and v2
        for( i = 0; i < XDIM; i++ ){
                v1[i] = (i % 5)+1;
                v2[i] = (i % 5)+1;
        }

        //run test
        gettimeofday(&tv_start, NULL);
        vectorAdd( XDIM, v1, v2, v );
        gettimeofday(&tv_stop, NULL);
        timersub(&tv_stop, &tv_start, &tv_diff);
        printf("Performance= %ld.%06ld sec\n", (long int) tv_diff.tv_sec, (long int)
tv_diff.tv_usec);

        //Validate for correctness
        for( i = 0; i < XDIM; i++ ){
                if( v[i] != v1[i] + v2[i] ){
                        printf("WRONG!!\n");
                        exit(EXIT_FAILURE);
                }
        }

        return 0;
}
```

Appendix H:

```
/**
 * Vector Addition with CUDA (Fall 2016):
 *
 * Members:
 * Emanuelle Crespi, Tolga Keskinoglu
 *
 * This test implements a parallel vector addition: C = A + B
 * discussed in the methodology section of Optimizing CPU-GPU Interactions.
 *
 * The following code makes use of the function call vectorAdd( int n, int *A, int *B, int *C )
 * to perform a vector addition of two arrays A and B of length n
 *
 * The output is verified before the program terminates to see that every
 * element of index i has C[i] == A[i] + B[i]
 *
 * We can see that there is a significant speedup in comparison to the time it takes
 * to perform the vectorAdd in the serial code.
 *
 * The output of the performance is displayed in seconds.
 * The runtime results are to be compared with the performance of vector.c
 *
 */

// System includes
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

// Jetson TK1 has device capability 1.x allowing 1024 threads/block
// defines dimension of vectored data as XDIM
#define XDIM 413124
#define THREADS_PER_BLOCK 1024

// Performs a vector addition of arrays v1 and v2
// in parallel on the GPU.  The index i is specified as
// a particular thread within some block on the device
// This allows for parallel computation on the device
__global__ void vectorAdd( int n, int *v1, int *v2, int *v ){
        int i =  blockIdx.x*blockDim.x + threadIdx.x;
        if( i < n ){
                v[i] = v1[i] + v2[i];
        }
}

int main(void){
        int i, v1[XDIM], v2[XDIM], v[XDIM];
         int *d_v1, *d_v2, *d_v, size = sizeof(int)*XDIM;
        cudaError_t error;

        //Populate data in v1 and v2
        for( i = 0; i < XDIM; i++ ){
                v1[i] = (i % 5)+1;
                v2[i] = (i % 5)+1;
        }

        //set up space on GPU device
        cudaMalloc( (void **) &d_v1, size );
        cudaMalloc( (void **) &d_v2, size );
        cudaMalloc( (void **) &d_v, size );

        //write data to GPU device
        cudaMemcpy( d_v1, v1, size, cudaMemcpyHostToDevice );
        cudaMemcpy( d_v2, v2, size, cudaMemcpyHostToDevice );
        /*******************************for testing
purposes*************************************

*********************************************************************************************/
        cudaDeviceSynchronize();
```

```
        // Allocate CUDA events that we'll use for timing
        cudaEvent_t start;
        error = cudaEventCreate(&start);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to create start event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        cudaEvent_t stop;
        error = cudaEventCreate(&stop);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to create stop event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        // Record the start event
        error = cudaEventRecord(start, NULL);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to record start event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        vectorAdd<<<(XDIM+THREADS_PER_BLOCK-1)/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( XDIM,
d_v1, d_v2, d_v );

        // Record the stop event
        error = cudaEventRecord(stop, NULL);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to record stop event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        // Wait for the stop event to complete
        error = cudaEventSynchronize(stop);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to synchronize on the stop event (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        float msecTotal = 0.0f;
        error = cudaEventElapsedTime(&msecTotal, start, stop);

        if (error != cudaSuccess)
        {
                fprintf(stderr, "Failed to get time elapsed between events (error code %s)!\n",
cudaGetErrorString(error));
                exit(EXIT_FAILURE);
        }

        // Compute and print the performance
        float msecPerVecAdd = msecTotal / 1;

        printf( "Performance= %.06f sec\n", msecPerVecAdd/1000.0 );
        /*****************************************************************************
***
```

```
        ***************************** for testing purposes
************************************/


        //Retrieve data from GPU
        cudaMemcpy( v, d_v, size, cudaMemcpyDeviceToHost );

        //Validate for correctness
        for( i = 0; i < XDIM; i++ ){
                if( v[i] != v1[i] + v2[i] ){
                        printf("WRONG!!\n");
                        exit(EXIT_FAILURE);
                }
        }

        return 0;
}
```

Appendix I: ***(50MB CPU Array Search - Data)***

Steady State: 12 V, 275 mA							
Tests	Array Search 50M						
	Run1	Run2	Run3	Run4			
Time	Current (mA)	Current (mA)	Current (mA)	Current (mA)	Avg. Current	Power (Watts)	Avg. Power
0	281	275	279	276	277.75	3.333	4.961733333
0.5	459	419	420	328	406.5	4.878	
1	457	420	433	420	432.5	5.19	Avg. Power Runtime
1.5	458	418	418	419	428.25	5.139	5.1342
2	458	421	419	419	429.25	5.151	
2.5	453	420	418	419	427.5	5.13	Energy (Joules)
3	457	420	418	422	429.25	5.151	223.278
3.5	458	421	415	421	428.75	5.145	
4	459	439	419	419	434	5.208	
4.5	458	420	420	420	429.5	5.154	
5	458	438	419	420	433.75	5.205	
5.5	459	421	419	419	429.5	5.154	
6	461	420	419	419	429.75	5.157	
6.5	460	421	421	420	430.5	5.166	
7	468	420	420	421	432.25	5.187	
7.5	459	418	417	420	426.5	5.142	
8	458	417	417	416	427	5.124	
8.5	458	417	416	418	427.25	5.127	
9	459	418	416	437	432.5	5.19	
9.5	455	418	416	419	427	5.124	
10	459	418	417	417	427.75	5.133	
10.5	459	418	416	417	427.5	5.13	
11	459	431	417	417	431	5.172	
11.5	456	418	417	418	427.25	5.127	
12	455	416	419	414	426	5.112	
12.5	456	415	414	415	425	5.1	
13	455	415	413	414	424.25	5.091	
13.5	456	415	413	414	424.5	5.094	
14	456	415	414	414	424.75	5.097	
14.5	430	415	414	414	418.25	5.019	
15	430	463	383	402	419.5	5.034	
15.5	429	431	448	463	442.75	5.313	
16	430	430	428	430	429.5	5.154	
16.5	429	430	428	430	429.25	5.151	
17	429	430	429	431	429.75	5.157	
17.5	429	430	462	431	438	5.256	
18	493	454	429	430	451.5	5.418	
18.5	499	430	485	430	461	5.532	
19	352	490	429	487	439.5	5.274	
19.5	283	487	429	488	421.75	5.061	
20	279	430	386	499	396.5	4.782	
20.5	280	347	298	350	318.75	3.825	
21	281	319	279	283	290.5	3.486	
21.5	280	281	279	279	279.75	3.357	
22	279	279	279	279	279	3.348	

Appendix J: *(100MB CPU Array Search - Data)*

	Array Search 100M						
	Run1	Run2	Run3	Run4			
Time	Current (mA)	Current (mA)	Current (mA)	Current (mA)	Avg. Current	Power (Watts)	Avg. Power
0	276	276	267	278	274.25	3.291	5.0065
0.5	425	419	419	456	429.75	5.157	
1	421	420	420	457	429.5	5.154	Avg. Power Runtime
1.5	385	415	419	456	418.75	5.025	5.244333333
2	389	420	419	457	421.25	5.055	
2.5	421	421	419	457	429.5	5.154	Energy (Joules)
3	420	420	419	457	429	5.148	440.572
3.5	421	422	420	457	430	5.16	
4	425	438	420	457	435	5.22	
4.5	421	423	420	457	430.25	5.163	
5	420	421	421	457	429.75	5.157	
5.5	421	420	427	461	432.25	5.187	
6	420	421	419	457	429.25	5.151	
6.5	421	420	419	458	429.5	5.154	
7	421	421	419	457	429.5	5.154	
7.5	410	420	419	459	427	5.124	
8	413	420	436	457	431.5	5.178	
8.5	420	420	412	458	427.5	5.13	
9	421	420	419	458	429.5	5.154	
9.5	421	424	419	457	430.25	5.163	
10	420	421	419	457	429.25	5.151	
10.5	420	420	419	458	429.25	5.151	
11	420	421	421	459	430.25	5.163	
11.5	421	420	415	457	428.25	5.139	
12	420	417	416	459	428	5.136	
12.5	417	418	417	455	426.75	5.121	
13	417	417	417	455	426.5	5.118	
13.5	417	418	416	454	426.25	5.115	
14	418	418	413	454	425.75	5.109	
14.5	414	414	414	455	424.25	5.091	
15	414	414	413	454	423.75	5.085	
15.5	415	414	414	454	424.25	5.091	
16	415	414	413	455	424.25	5.091	
16.5	415	414	414	454	424.25	5.091	
17	416	414	413	455	424.5	5.094	
17.5	414	415	414	455	424.5	5.094	
18	415	415	414	455	424.75	5.097	
18.5	415	414	414	454	424.25	5.091	
19	416	414	414	455	424.75	5.097	
19.5	415	414	416	454	424.75	5.097	
20	415	415	414	455	424.75	5.097	
20.5	415	414	413	454	424	5.088	
21	415	413	413	455	424	5.088	
21.5	416	414	413	455	424.5	5.094	
22	416	413	414	455	424.5	5.094	
22.5	416	414	414	459	425.75	5.109	
23	415	414	413	455	424.25	5.091	
23.5	415	415	414	454	424.5	5.094	
24	416	415	414	455	425	5.1	
24.5	415	414	413	454	424	5.088	
25	415	414	413	454	424	5.088	
25.5	416	415	414	454	424.75	5.097	
26	415	414	412	455	424	5.088	
26.5	416	414	413	455	424.5	5.094	
27	415	415	414	454	424.5	5.094	
27.5	415	414	414	461	426	5.112	
28	416	414	414	465	427.25	5.127	
28.5	415	414	414	529	443	5.316	
29	415	415	423	430	420.75	5.049	
29.5	465	464	464	429	455.5	5.466	
30	456	492	432	432	453	5.436	
30.5	431	431	429	429	430	5.16	
31	433	430	429	499	447.75	5.373	
31.5	430	430	431	429	430	5.16	
32	430	430	321	429	402.5	4.83	
32.5	430	430	348	429	409.25	4.911	
33	488	498	498	431	478.75	5.745	
33.5	491	430	430	337	422	5.064	
34	431	463	429	428	437.75	5.253	
34.5	431	430	429	429	429.75	5.157	
35	401	430	458	429	429.5	5.154	
35.5	348	340	347	430	366.25	4.395	
36	430	333	430	430	405.75	4.869	
36.5	432	340	430	430	408	4.896	
37	432	326	429	429	404	4.848	
37.5	429	430	429	428	429	5.148	
38	430	431	496	429	446.5	5.358	
38.5	430	430	487	429	444	5.328	
39	491	489	430	305	428.75	5.145	
39.5	486	427	429	279	405.25	4.863	
40	499	427	431		452.3333333	5.428	
40.5	464	427	350		413.6666667	4.964	
41	339	427	279		348.3333333	4.18	
41.5	282	426			354	4.248	
42	282	351			316.5	3.798	
42.5	279	280			279.5	3.354	
43		280			280	3.36	
43.5		281			281	3.372	

Appendix K: ***(CPU Prime Test - Data)***

Primality						
Run1	Run2	Run3	Run4			
Current (mA)	Current (mA)	Current (mA)	Current (mA)	Avg. Current	Power (Watts)	Avg. Power
275	308	279	279	285.25	3.423	4.963
412	409	416	412	412.25	4.947	
419	420	419	420	419.5	5.034	Energy (Joules)
423	420	421	421	421.25	5.055	134.001
416	425	428	426	423.75	5.085	
425	425	425	429	426	5.112	
430	429	430	428	429.25	5.151	
429	430	429	430	429.5	5.154	
431	430	430	430	430.25	5.163	
426	431	430	430	429.25	5.151	
428	431	429	435	430.75	5.169	
429	430	430	446	433.75	5.205	
430	430	430	430	430	5.16	
429	430	430	429	429.5	5.154	
430	431	431	432	431	5.172	
431	431	430	430	430.5	5.166	
431	431	429	431	430.5	5.166	
433	434	434	434	433.75	5.205	
433	432	433	433	432.75	5.193	
433	434	433	433	433.25	5.199	
433	434	433	433	433.25	5.199	
434	449	433	434	437.5	5.25	
433	434	433	444	436	5.232	
434	434	434	429	432.75	5.193	
427	431	430	430	429.5	5.154	
282	308	281	283	288.5	3.462	
304	281	281	283	287.25	3.447	

(CPU Vector Addition - Data)

Vector Addition						
Run1	Run2	Run3	Run4			
Current (mA)	Current (mA)	Current (mA)	Current (mA)	Avg. Current	Power (Watts)	Avg. Power
276	279	279	281	278.75	3.345	3.5745
314	315	301	300	307.5	3.69	
282	302	283	282	287.25	3.447	Energy (Joules)
			318	318	3.816	14.298

(CPU Hash Test - Data)

HASH						
Run1	Run2	Run3	Run4			
Current (mA)	Current (mA)	Current (mA)	Current (mA)	Avg. Current	Power (Watts)	Avg. Power
275	280	276	280	277.75	3.333	5.174571429
460	460	482	283	421.25	5.055	
461	462	459	462	461	5.532	Energy (Joules)
469	466	461	464	465	5.58	36.222
463	465	463	462	463.25	5.559	
464	466	465	464	464.75	5.577	
	466		465	465.5	5.586	

Appendix L: *(50MB GPU Array Search - Data)*

	Array Search 50M						
	Run1	Run2	Run3	Run4			
Time	Current (mA)	Current (mA)	Current (mA)	Current (mA)	Avg. Current	Power (Watts)	Avg. Power
0	275	279	278	281	278.25	3.339	5.275644068
0.5	423	461	461	463	452	5.424	
1	424	462	462	463	452.75	5.433	Avg. Power Runtime
1.5	424	462	462	463	452.75	5.433	5.073
2	424	461	463	465	453.25	5.439	
2.5	430	462	462	464	454.5	5.454	Energy (Joules)
3	423	462	463	464	453	5.436	311.263
3.5	424	463	462	464	453.25	5.439	
4	435	463	463	464	456.25	5.475	
4.5	426	478	482	463	464.75	5.577	
5	427	500	463	464	463.5	5.562	
5.5	434	463	463	463	455.75	5.469	
6	425	463	463	463	453.5	5.442	
6.5	427	462	463	464	454	5.448	
7	426	463	471	465	456.25	5.475	
7.5	471	469	472	472	471	5.652	
8	469	469	469	473	470	5.64	
8.5	478	491	469	473	477.75	5.733	
9	471	469	469	471	470	5.64	
9.5	469	469	470	471	469.75	5.637	
10	469	468	470	470	469.25	5.631	
10.5	468	468	469	471	469	5.628	
11	468	482	470	471	472.75	5.673	
11.5	465	469	469	470	468.25	5.619	
12	465	465	466	467	465.75	5.589	
12.5	465	465	466	467	465.75	5.589	
13	465	464	466	466	465.25	5.583	
13.5	465	464	466	467	465.5	5.586	
14	465	466	467	467	466.25	5.595	
14.5	465	466	466	466	465.75	5.589	
15	465	465	465	466	465.25	5.583	
15.5	466	466	465	466	465.75	5.589	
16	465	466	466	467	466	5.592	
16.5	421	422	424	424	422.75	5.073	
17	422	422	424	423	422.75	5.073	
17.5	434	421	423	420	424.5	5.094	
18	422	421	424	420	421.75	5.061	
18.5	421	421	423	423	422	5.064	
19	421	422	423	422	422	5.064	
19.5	422	422	423	422	422.25	5.067	
20	438	422	423	425	427	5.124	
20.5	434	422	440	423	429.75	5.157	
21	422	454	491	426	448.25	5.379	
21.5	422	421	463	482	447	5.364	
22	424	422	424	481	437.75	5.253	
22.5	422	422	423	481	437	5.244	
23	422	422	423	423	422.5	5.07	
23.5	492	484	423	423	455.5	5.466	
24	487	499	423	423	458	5.496	
24.5	421	422	423	423	422.25	5.067	
25	422	422	423	422	422.25	5.067	
25.5	423	422	423	422	422.5	5.07	
26	428	422	423	422	423.75	5.085	
26.5	419	423	460	424	431.5	5.178	
27	423	424	461	423	432.75	5.193	
27.5	519	385	374	489	441.75	5.301	
28	283		340	347	323.3333333	3.88	
28.5	279		349		314	3.768	
29			301		301	3.612	

Appendix M: *(100MB GPU Array Search - Data)*

Tests	Array Search 100M						
	Run1	Run2	Run3	Run4			
Time	Current (mA)	Current (mA)	Current (mA)	Current (mA)	Avg. Current	Power (Watts)	Avg. Power 5.263106195
0	276	275	276	279	276.5	3.318	
0.5	421	422	459	460	440.5	5.286	
1	421	422	463	460	441.5	5.298	Avg. Power Runtime 5.0745
1.5	423	424	461	461	442.25	5.307	
2	423	423	461	462	442.25	5.307	
2.5	422	422	462	453	439.75	5.277	Energy (Joules) 594.731
3	421	422	462	461	441.5	5.298	
3.5	423	424	461	461	442.25	5.307	
4	422	423	464	461	442.5	5.31	
4.5	422	425	464	462	443.25	5.319	
5	408	423	461	461	438.25	5.259	
5.5	422	424	461	463	442.5	5.31	
6	423	424	462	462	442.75	5.313	
6.5	423	424	464	461	443	5.316	
7	456	424	464	462	451.5	5.418	
7.5	424	424	463	462	443.25	5.319	
8	422	423	462	462	442.25	5.307	
8.5	423	424	462	461	442.5	5.31	
9	422	424	463	457	441.5	5.298	
9.5	424	424	462	462	443	5.316	
10	423	423	459	462	441.75	5.301	
10.5	425	424	462	462	443.25	5.319	
11	442	420	463	461	446.5	5.358	
11.5	423	423	463	457	441.5	5.298	
12	419	425	458	461	440.75	5.289	
12.5	420	424	459	461	441	5.292	
13	420	424	458	458	440	5.28	
13.5	420	426	458	457	440.25	5.283	
14	419	420	458	458	438.75	5.265	
14.5	420	421	458	471	442.5	5.31	
15	468	469	468	468	468.25	5.619	
15.5	467	464	465	464	465	5.58	
16	463	465	464	464	464	5.568	
16.5	464	466	464	464	464.5	5.574	
17	464	465	464	465	464.5	5.574	
17.5	464	466	465	464	464.75	5.577	
18	465	465	465	464	464.75	5.577	
18.5	464	465	464	464	464.25	5.571	
19	465	465	464	465	464.75	5.577	
19.5	464	465	464	464	464.25	5.571	
20	464	465	465	464	464.5	5.574	
20.5	464	466	464	464	464.5	5.574	
21	464	465	464	464	464.25	5.571	
21.5	464	465	465	464	464.5	5.574	
22	464	466	464	464	464.5	5.574	
22.5	464	465	465	465	464.75	5.577	
23	464	465	465	465	464.75	5.577	
23.5	465	465	465	465	465	5.58	
24	464	466	464	465	464.75	5.577	
24.5	464	465	464	464	464.25	5.571	
25	464	465	465	465	464.75	5.577	
25.5	464	465	465	465	464.75	5.577	
26	465	465	465	464	464.75	5.577	
26.5	464	465	465	464	464.5	5.574	
27	464	465	465	464	464.5	5.574	
27.5	464	466	466	464	465	5.58	
28	465	465	465	464	464.75	5.577	
28.5	465	465	464	464	464.5	5.574	
29	464	466	464	464	464.5	5.574	
29.5	465	466	465	464	465	5.58	
30	465	465	466	464	465	5.58	
30.5	465	466	464	464	464.75	5.577	
31	465	466	466	453	462.5	5.55	
31.5	464	465	465	469	465.75	5.589	
32	464	465	465	465	464.75	5.577	
32.5	423	465	465	422	443.75	5.325	
33	422	425	422	422	422.75	5.073	
33.5	423	423	422	424	423	5.076	
34	423	425	422	422	423	5.076	
34.5	423	423	422	421	422.25	5.067	
35	423	422	421	422	422	5.064	
35.5	423	423	452	421	429.75	5.157	
36	424	422	421	426	423.25	5.079	
36.5	425	423	421	440	427.25	5.127	
37	422	425	421	420	422	5.064	
37.5	481	423	421	452	444.25	5.331	
38	422	423	458	457	440	5.28	
38.5	458	422	421	421	430.5	5.166	
39	423	423	421	422	422.25	5.067	
39.5	422	482	482	422	452	5.424	
40	422	491	421	421	438.75	5.265	
40.5	425	424	421	422	423	5.076	
41	423	423	421	422	422.25	5.067	
41.5	423	422	423	421	422.25	5.067	
42	423	422	422	421	422	5.064	
42.5	423	423	423	457	431.5	5.178	
43	493	423	421	422	439.75	5.277	
43.5	422	422	421	422	421.75	5.061	
44	422	422	479	479	450.5	5.406	
44.5	418	476	418	487	449.75	5.397	
45	419	429	418	418	421	5.052	
45.5	419	418	417	419	418.25	5.019	
46	418	419	417	418	418	5.016	
46.5	418	419	417	418	418	5.016	
47	418	418	418	418	418	5.016	
47.5	418	419	418	418	418.25	5.019	
48	419	419	417	418	418.25	5.019	
48.5	419	418	418	418	418.25	5.019	
49	418	418	418	479	433.25	5.199	
49.5	419	418	419	486	435.5	5.226	
50	418	419	418	418	418.25	5.019	
50.5	419	422	418	418	419.25	5.031	
51	474	445	475	419	453.25	5.439	
51.5	487	418	419	419	435.75	5.229	
52	419	419	418	418	418.5	5.022	
52.5	419	478	417	418	433	5.196	
53	419	348	418	419	401	4.812	
53.5	418	419	418	418	418.25	5.019	
54	419	419	418	419	418.75	5.025	
54.5	419	419	418	350	401.5	4.818	
55	548	418	283		416.3333333	4.996	
55.5	303	285	282		290	3.48	
56	301	282	279		287.3333333	3.448	

Appendix N: *(GPU Prime Test - Data)*

Primality (Regular)						
Run1	Run2	Run3	Run4			
Current (mA)	Current (mA)	Current (mA)	Current (mA)	Avg. Current	Power (Watts)	Avg. Power
314	276	278	278	286.5	3.438	3.791
319	417	278	328	335.5	4.026	
383	298	386	383	362.5	4.35	Energy (Joules)
418	384	283	304	347.25	4.167	22.746
282	281	282	282	281.75	3.381	
	282			282	3.384	

(GPU Prime Test - 20 repetitions - Data)

Primality (Long Run)						
Run1	Run2	Run3	Run4			
Current (mA)	Current (mA)	Current (mA)	Current (mA)	Avg. Current	Power (Watts)	Avg. Power
280	281	279	278	279.5	3.354	6.86085
327	369	394	319	352.25	4.227	
384	308	654	335	420.25	5.043	Energy (Joules)
514	384	654	459	502.75	6.033	137.217
670	652	675	653	662.5	7.95	
654	671	641	657	655.75	7.869	
699	672	658	671	675	8.1	
672	674	676	678	675	8.1	
676	656	676	674	670.5	8.046	
674	672	674	676	674	8.088	
672	673	675	677	674.25	8.091	
675	675	677	676	675.75	8.109	
676	676	675	674	675.25	8.103	
678	674	680	675	676.75	8.121	
676	673	676	676	675.25	8.103	
674	674	674	676	674.5	8.094	
676	677	435	679	616.75	7.401	
676	676	284	675	577.75	6.933	
284	284		287	285	3.42	
	336			336	4.032	

(GPU Vector Addition - Data)

Vector Addition						
Run1	Run2	Run3	Run4			
Current (mA)	Current (mA)	Current (mA)	Current (mA)	Avg. Current	Power (Watts)	Avg. Power
328	278	296	279	295.25	3.543	3.948
436	301	429	446	403	4.836	
283	285	282	305	288.75	3.465	Energy (Joules)
						11.844

(GPU Hash Test - Data)

HASH						
Run1	Run2	Run3	Run4			
Current (mA)	Current (mA)	Current (mA)	Current (mA)	Avg. Current	Power (Watts)	Av.g Power
318	304	281	279	295.5	3.546	4.958357143
301	284	448	460	373.25	4.479	
305	460	461	464	422.5	5.07	Energy (Joules)
460	461	460	462	460.75	5.529	69.417
461	457	433	433	446	5.352	
535	432	580	580	531.75	6.381	
479	580	649	650	589.5	7.074	
658	650	534	519	590.25	7.083	
650	508	543	498	549.75	6.597	
507	484	296	283	392.5	4.71	
296	295	285	284	290	3.48	
282				282	3.384	
282				282	3.384	
279				279	3.348	

Acknowledgements:

We would like to express our thanks to the following faculty members of the University of Maryland Department of Electrical and Computer Engineering/Computer Science…

This research would not have been possible without the support of Dr. R.D. Gomez for his prior approval and funding of the Jetson TK1 board. We would also like to thank Dr. Manoj Franklin for his superb wisdom and guidance throughout multiple phases of this research development over the semesters. Our gratitude goes out to Brian Quinn, and Gwen Flasinski for their assistance with providing equipment and temporary lab space towards the end of the semester. We are also appreciative of Dr. Bruce Jacobs, Dr. Donald Yeung, Dr. Evan Golub, and Daniel Gerzhoy for their valuable insight throughout the remainder of this project.

Finally, a special thanks goes to Jenn Wivell for going out of her way to get the proposal approved for research credits.

Thank you all for helping us grow throughout the process of this research.