University of Louisville
ThinkIR: The University of Louisville's Institutional Repository

Electronic Theses and Dissertations

12-2009

Increasing performance of blowfish encryption using CUDA.

Joseph Anthony Feist
University of Louisville

Follow this and additional works at: https://ir.library.louisville.edu/etd

Recommended Citation

Feist, Joseph Anthony, "Increasing performance of blowfish encryption using CUDA." (2009). *Electronic Theses and Dissertations.* Paper 431.
https://doi.org/10.18297/etd/431

This Master's Thesis is brought to you for free and open access by ThinkIR: The University of Louisville's Institutional Repository. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of ThinkIR: The University of Louisville's Institutional Repository. This title appears here courtesy of the author, who has retained all other copyrights. For more information, please contact thinkir@louisville.edu.

INCREASING PERFORMANCE OF BLOWFISH ENCRYPTION USING CUDA

By

Joseph Anthony Feist

A Thesis
Submitted to the Faculty of the
University of Louisville
Speed School of Engineering
As Partial Fulfillment of the Requirements
For the Professional Degree of

MASTER OF ENGINEERING

Department of Computer Engineering and Computer Science

December 2009

TABLE OF CONTENTS

Page

APPROVAL PAGE.. ii

ACKNOWLEDGMENTS...................................... iii

ABSTRACT... iv

LIST OF TABLES....................................... v

LIST OF FIGURES...................................... vi

I. INTRODUCTION...................................... 1

II. GPGPU AND CUDA................................... 3

A. GPGPU... 3

B. CUDA.. 4

III. IMPLEMENTATION OF BLOWFISH ENCRYPTION........... 8

A. BLOWFISH.. 8

B. CPU IMPLEMENTATION................................ 11

C. CUDA IMPLEMENTATION............................... 20

IV. RESULTS AND DISCUSSION OF RESULTS................ 30

V. CONCLUSIONS AND FUTURE WORKS...................... 40

A. CONCLUSIONS....................................... 40

B. SHARED MEMORY..................................... 40

C. THREAD-BLOCK SIZE CONSIDERATIONS................... 41

D. CONSTANT MEMORY................................... 41

E. DEVICE MEMORY TRANSFER AND ALLOCATION.............. 41

REFERENCES... 43

VITA... 44

INCREASING PERFORMANCE OF BLOWFISH ENCRYPTION USING CUDA

Submitted by: ______________________________
Joseph A. Feist

A Thesis Approved On

____
(Date)

by the Following Reading and Examination Committee

____
Dr. Dar-Jen Chang, Thesis Co-director

____
Dr. Ahmed Desoky, Thesis Co-director

____
Dr. John F. Naber

ACKNOWLEDGEMENTS

Thanks to Dr. Chang and Dr. Desoky for serving as co-directors and to Dr. Naber for serving in the committee.

ABSTRACT

This thesis describes a method of gaining better performance from the Blowfish cryptographic algorithm using GPU acceleration via CUDA (Compute Unified Device Architecture). It is an attempt to resolve the issue of the cryptographic process taking up time and resources. Reducing or eliminating one or both of these issues would perhaps lead to more widespread use of algorithms such as Blowfish to secure individuals' data by making cryptographic security less costly in terms of time and resources. It is further noted that this method or similar could be used in the case of other algorithms especially those closely related to Blowfish, such as the family of block ciphers.

LIST OF TABLES

[TABLE I] - CPU AND GPU RUNTIME FOR VARIOUS DATA SIZES

[TABLE II] - PERFORMANCE MEASURES FOR 65535 THREAD-BLOCKS

[TABLE III] - PERFORMANCE MEASURES FOR 32000 THREAD-BLOCKS

[TABLE IV] - PERFORMANCE MEASURES FOR 16000 THREAD-BLOCKS

[TABLE V] - PERFORMANCE MEASURES FOR 8000 THREAD-BLOCKS

[TABLE VI] - PERFORMANCE MEASURES FOR 4000 THREAD-BLOCKS

[TABLE VII] - PERFORMANCE MEASURES FOR 2000 THREAD-BLOCKS

[TABLE VIII] - PERFORMANCE MEASURES FOR 1000 THREAD-BLOCKS

[TABLE IX] - PERFORMANCE MEASURES FOR 500 THREAD-BLOCKS

[TABLE X] - PERFORMANCE MEASURES FOR 100 THREAD-BLOCKS

[TABLE XI] - PERFORMANCE MEASURES FOR 50 THREAD-BLOCKS

[TABLE XII] - PERFORMANCE MEASURES FOR 25 THREAD-BLOCKS

[TABLE XIII] - PERFORMANCE MEASURES FOR ONE THREAD-BLOCK

LIST OF FIGURES

[FIGURE 1] – Illustrated Differences Between CPU And GPU

[FIGURE 2] – Physical Layout Of Nvidia G80 Series GPU

[FIGURE 3] – Streaming Multi-Processor(SM) Made Up Of Streaming Processor(SP) “Cluster”

[FIGURE 4] – Illustration Of Blowfish Feistel Network

[FIGURE 5] – Illustration Of Feistel Function In Blowfish

[FIGURE 6] - Blowfish_CUDA References

[FIGURE 7] - Blowfish_CUDA Project Directory Structure

[FIGURE 8] - Post-Build Event Visual

[FIGURE 9] – Relationship Between Speed-Up Factor And Number Of Thread-Blocks

[FIGURE 10] - CPU And GPU Runtimes Vs. Thread-Block Numbers

CHAPTER I

INTRODUCTION

The purpose of this thesis is to investigate the possibility and effectiveness of efficiently using the large number of computing cores in a GPU for enhancing the speed and performance of the Blowfish encryption algorithm, specifically using Nvidia's CUDA. In many instances encryption and decryption of data are time consuming and computationally intensive procedures. CUDA is specifically designed to use the power of the many small cores in a GPU to improve performance and speed of computationally intensive operations via threading and using true concurrence of these threads. Many have made efforts to accelerate cryptographic algorithms by using CUDA enabled GPUs. These attempts do not, however, use the architecture to its fullest potential. Rather, they simply run CPU code on the CUDA device. This approach only attempts to use the GPU to out-number the CPU in terms of number of cores. It does not take into account efficient use of available memory and device architecture to improve the performance. This thesis attempts to demonstrate a method to take steps to more efficiently code the Blowfish cryptographic algorithm for CUDA and achieve a much less time consuming process than currently exists.

The remainder of this paper describes the process used to improve the coding of the algorithm. First, Chapter II GPGPU AND CUDA contains a description of GPGPU, CUDA, and equipment used in setting up this experiment. This includes hardware and software utilities, as well as a general description of CUDA as a processor architecture. Chapter III IMPLEMENTAION OF BLOWFISH ENCRYPTION outlines the Blowfish encryption algorithm and identifies areas of parallelism within the algorithm that can be exploited to get good use of the architecture of the GPU. This portion involves the main work and research involved in the experiment. Next Chapter IV RESULTS AND DISCUSSION OF RESULTS presents the results obtained from this experiment, and briefly discusses the meanings of the measurements from each of the experiment’s iterations for different sizes of data. Finally, Chapter V CONCLUSIONS AND FUTURE WORKS gives conclusions based on the results obtained, and offers recommendations for possible improvements and paths for further research and experimentation.

CHAPTER II

GPGPU AND CUDA

A. GPGPU

This research is an extension of the idea of GPGPU. The experiment itself pursues the main focus presented by GPGPU.org[1]:

> "GPGPU stands for General-Purpose computation on Graphics Processing Units, also known as GPU Computing. Graphics Processing Units (GPUs) are high-performance many-core processors capable of very high computation and data throughput. Once specially designed for computer graphics and difficult to program, today's GPUs are general-purpose parallel processors with support for accessible programming interfaces and industry-standard languages such as C. Developers who port their applications to GPUs often achieve speedups of orders of magnitude vs. optimized CPU implementations."

The difference between a GPU and CPU is shown in FIGURE 1, below. Notice that in the same area the GPU has more transistors that are devoted to computation [2]. This means that the GPU has greater aptitude for computation than the CPU. It can also be inferred from the figure that because of the larger ALU size in the CPU that each ALU is more powerful. This is the largest difference between the CPU and GPU. The CPU is made up of a small number of large, powerful cores while the GPU is constructed from a large number of small, less powerful cores. This means that the CPU can do a few more involved computational

tasks, whereas the GPU can do many more, smaller operation tasks at the same time.

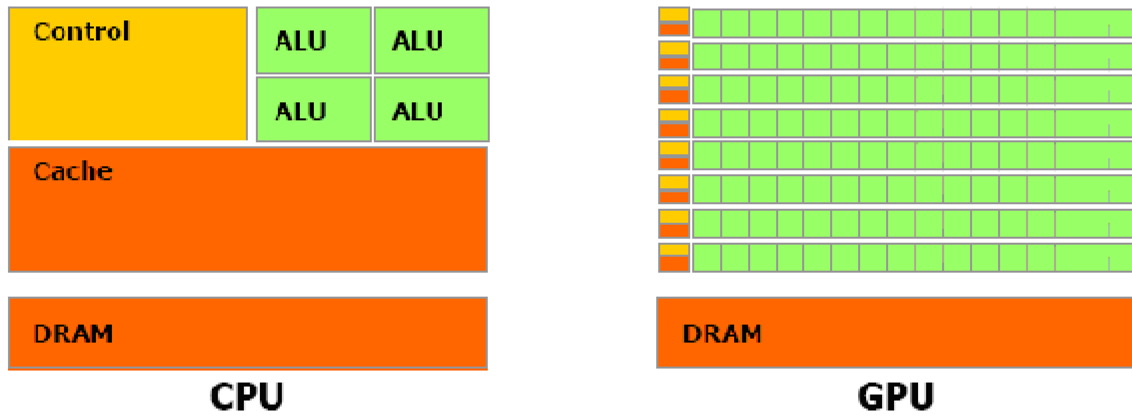

FIGURE 1 – Illustrated Differences Between CPU and GPU [3]

B. CUDA

This experiment employs GPGPU through CUDA. CUDA stands for Nvidia's Compute Unified Device Architecture. CUDA refers to the architecture of the GPU, drivers and interfaces, as well as the language used for programming. Nvidia GPUs have two modes. One is the usual display mode that interacts with a computer monitor. The second is a compute or CUDA mode used for general purpose processing. Below is a block diagram of an Nvidia G80 series GPU in CUDA mode:

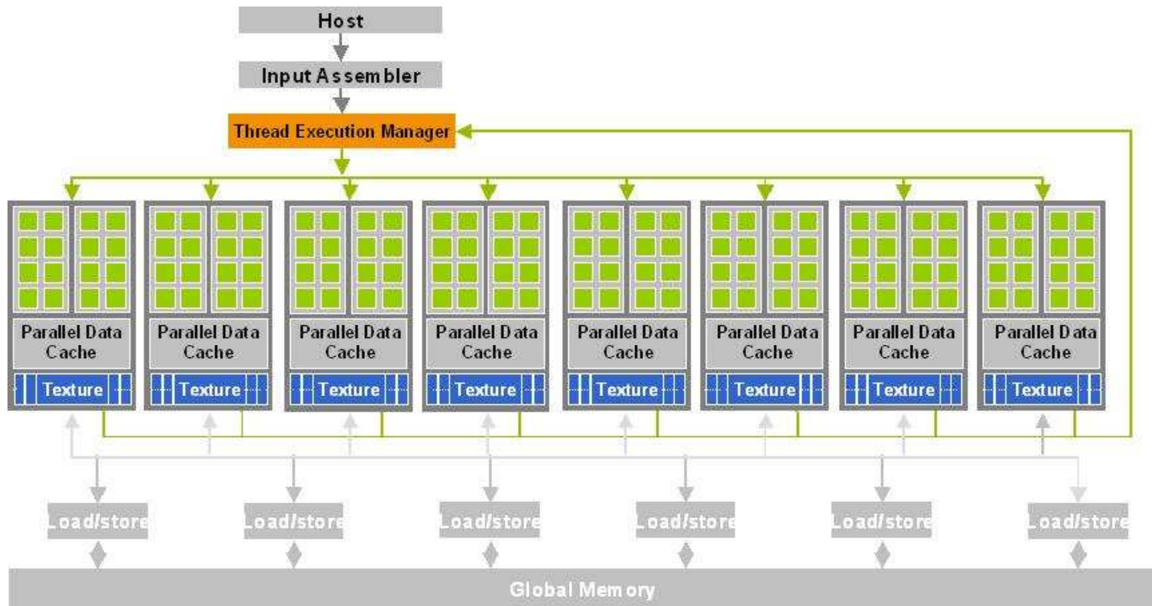

FIGURE 2 – Physical Layout of Nvidia G80 Series GPU [4]

It should be noted that a GPU has its own dedicated global device memory (see FIGURE 2). For each GPU application or kernel that is launched this memory must be allocated and the data to be processed must be transferred from the host memory. This is normally the step that takes the time not required for CPU computing (thus one of the GPGPU overheads vs. CPU computing). Which means transfer must be factored into the speedup calculation. In other words the time consuming procedure of transferring data between the host and the device must be justified before an actual speedup is produced.

FIGURE 3 below gives a little bit closer look at a portion of FIGURE 2. It indicates two of the most important parts of the GPU in CUDA mode to consider for

proper program design. The Streaming Multi-Processor(SM) processes at the thread-block level and is made up of Streaming Processors (SPs). SPs process data at the thread level. The basic processing unit is a thread. A thread-block is made up of many different threads, and multiple blocks are arranged in a grid. A grid could also be thought of as a kernel, because when a kernel function made of device code (code for the GPU) is launched the entire context of the function on the GPU is a grid.

FIGURE 3 – Streaming Multi-Processor(SM) Made Up of Streaming Processor(SP) "Cluster"[5]

To effectively program in CUDA it is necessary to know GPU limitations as far as memory, grid size, thread-block size, and the number of concurrent threads that can be handled. For instance this experiment is programmed for an Nvidia GeForce 8800 GTX GPU. This GPU has 768 MB of device memory, a maximum of 512 threads allowed per thread-block,

and 768 concurrent threads on one SM. These numbers are totally dependent upon the specific model of GPU used.

Another important idea behind CUDA not explored fully in this experiment is shared memory. This memory resides within each SM. It is shared between all threads in a thread-block and allows much faster access than global device memory. Shared memory can greatly decrease computation time by limiting the number of costly accesses to global memory. Shared memory is discussed somewhat further in Chapter V CONCLUSIONS AND FUTURE WORKS.

Now that the reader has some understanding of CUDA basics, here is a brief discussion of the setup of the environment. It is necessary to have a CUDA 2.2 enabled GPU from Nvidia. After this has been confirmed, download and install CUDA 2.2 from Nvidia's website following the instructions provided on the web site in the form of the Quick Start Guide.

CHAPTER III

IMPLEMENTAION OF BLOWFISH ENCRYPTION

A. BLOWFISH ALGORITHM

Blowfish is well documented and is described by Bruce Schneier [6], its creator in the steps below:

> "Blowfish is a variable-length key, 64-bit block cipher. The algorithm consists of two parts: a key-expansion part and a data- encryption part. Key expansion converts a key of at most 448 bits into several subkey arrays totaling 4168 bytes. Data encryption occurs via a 16-round Feistel network. Each round consists of a key-dependent permutation, and a key- and data-dependent substitution. All operations are XORs and additions on 32-bit words. The only additional operations are four indexed array data lookups per round.
>
> Subkeys:
>
> Blowfish uses a large number of subkeys. These keys must be precomputed before any data encryption or decryption.
>
> 1. The P-array consists of 18 32-bit subkeys: P1, P2,..., P18.
>
> 2. There are four 32-bit S-boxes with 256 entries each:
>
> S1,0, S1,1,..., S1,255;
> S2,0, S2,1,..,, S2,255;
> S3,0, S3,1,..., S3,255;
> S4,0, S4,1,..,, S4,255.
>
> The exact method used to calculate these subkeys will be described later.
>
> Encryption:
>
> Blowfish is a Feistel network consisting of 16 rounds (see Figure 4). The input is a 64-bit data element, x.
>
> Divide x into two 32-bit halves: xL, xR
> For i = 1 to 16:
> xL = xL XOR Pi
> xR = F(xL) XOR xR

Swap xL and xR
Next i
Swap xL and xR (Undo the last swap.)
xR = xR XOR P17
xL = xL XOR P18
Recombine xL and xR
Function F (see Figure 5):
Divide xL into four eight-bit quarters: a, b, c, and d
F(xL) = ((S1,a + S2,b mod 2^{32}) XOR S3,c) + S4,d mod 2^{32}

Decryption is exactly the same as encryption, except that P1, P2,..., P18 are used in the reverse order.
Implementations of Blowfish that require the fastest speeds should unroll the loop and ensure that all subkeys are stored in cache.

Generating the Subkeys:
The subkeys are calculated using the Blowfish algorithm. The exact method is as follows:

1. Initialize first the P-array and then the four S-boxes, in order, with a fixed string. This string consists of the hexadecimal digits of pi (less the initial 3). For example:
P1 = 0x243f6a88
P2 = 0x85a308d3
P3 = 0x13198a2e
P4 = 0x03707344

2. XOR P1 with the first 32 bits of the key, XOR P2 with the second 32-bits of the key, and so on for all bits of the key (possibly up to P14). Repeatedly cycle through the key bits until the entire P-array has been XORed with key bits. (For every short key, there is at least one equivalent longer key; for example, if A is a 64-bit key, then AA, AAA, etc., are equivalent keys.)

3. Encrypt the all-zero string with the Blowfish algorithm, using the subkeys described in steps (1) and (2).

4. Replace P1 and P2 with the output of step (3).

5. Encrypt the output of step (3) using the Blowfish algorithm with the modified subkeys.

6. Replace P3 and P4 with the output of step (5).

7. Continue the process, replacing all entries of the P- array, and then all four S-boxes in order, with the output of the continuously-changing Blowfish algorithm."

Below are two figures that demonstrate the Feistel network that Blowfish uses as well as the operations performed with in the Feistel or F-Function.

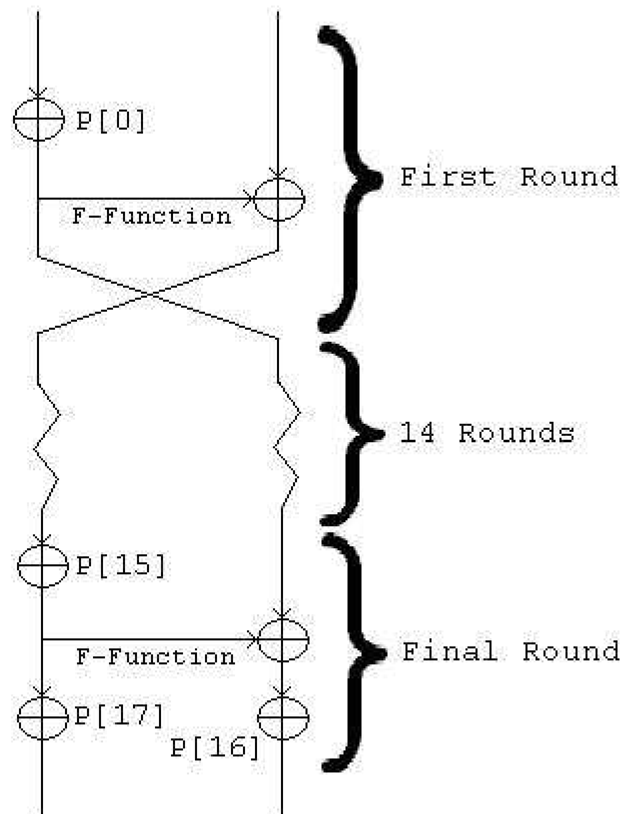

FIGURE 4 – Illustration Of Blowfish Feistel Network

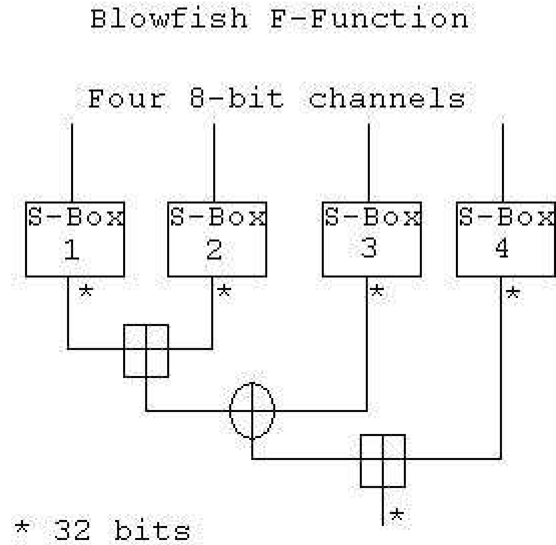

FIGURE 5 – Illustration Of Feistel Function In Blowfish

B. CPU IMPLEMENTATION

Below is C# code for the implementation of Blowfish used for this experiment. It deviates from the algorithm described in the above section in a few ways that may compromise security, but do not impact the runtime of the encryption process. First, the S-Box and P-Array values are randomly generated rather than derived from the hexadecimal digits of pi and computed with respect to the key. This decision was made simply to create neater code. Second, the plaintext data is pre-processed into an array of 32-bit integer values. Rather than taking 64 bits of data and splitting it into two 32-bit halves, this implementation takes two 32-bit entries from an array.

This is to allow the use of a simple data structure.

Third, only Blowfish encryption has been implemented.

Decryption was not implemented because it uses the same

algorithm with reversed application of the P-Array.

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using GASS.CUDA;
using GASS.CUDA.Types;
using System.IO;

namespace Blowfish_CUDA
{
    class Program
    {
        const int BLOCK_SIZE = 512;
        static Random rand = new Random(DateTime.Today.Millisecond);

        static void Main(string[] args)
        {
            string tStr,rStr;
            Console.WriteLine("This Program will compare the runtime of
CPU implementation of Blowfish with CUDA implementation of the same
algorithm");

            //Test to make sure CPU and GPU obtain the same result
            Console.WriteLine("Test to make sure CPU and GPU obtain
same result:  ");
            TestBlfsh();

            //Random data or Real data
            for(int l = 0;l<Int32.MaxValue;l++){
                      Console.WriteLine("Enter Number of blocks to use
between 0 and 65535:  ");
                  tStr = Console.ReadLine();
                  uint t = UInt32.Parse(tStr);
                      Console.WriteLine("Enter the number of times to
use this size data:  ");
                      rStr = Console.ReadLine();
                      uint r = UInt32.Parse(rStr);
                      for(int x = 0;x<r;x++){
                    string type;
                    Console.WriteLine("For real data enter 1.  Enter
anything else for random data.");
                    type = Console.ReadLine();
                    if (type == "1")
```

```
            {
                RealBlowfishCompare(t);
            }
            else
            {
                BlowfishCompare(t);
            }
              }
        }

}

//Populate random data of the specified size
static uint[] GetData(uint i)
{
    i = 2 * i;
    uint[] retD = new uint[i];
    for (int k = 0; k < i; k++)
    {
        retD[k] = (uint)rand.Next();
    }
    return (retD);
}

static uint[] GetEmptyData(uint i)
{
    i = 2 * i;
    uint fourBytes = 0;
    uint[] retD = new uint[i];
    for (int k = 0; k < i; k++)
    {
        retD[k] = fourBytes;
    }
    return (retD);
}

// Random SBoxes
static uint[,] fillSBoxes()
{
    uint[,] s = new uint[4,256];
    for (uint x = 0; x < 4; x++)
    {
        for (uint y = 0; y < 256; y++)
        {
            s[x, y] = (uint)rand.Next();
        }
    }
    return (s);
}

//2D Array for CUDA
static uint[] fillSBoxes(uint r)
{
    uint[] s = new uint[1040];
    for (int i = 0; i < 1040; i++)
    {
        s[i] = (uint)rand.Next();
```

```
            }
            return (s);
        }

        static uint[] fillPArray()
        {
            uint[] p = new uint[18];
            for (int i = 0; i < 18; i++)
            {
                p[i] = (uint)rand.Next();
            }
            return (p);
        }

        static void RealBlowfishCompare(uint t)
        {
            FileStream file = new
FileStream(@"C:\CUDA2_2\sdk\CUDA.NET\JAFEIS01\Blowfish_CUDA\eightbytes.
txt", FileMode.Open);
            uint s = 1024 * t; //65535 is the max number accepted for
number of blocks by cuda.Launch() and so the max value for t
            uint[] PlainText = new uint[s];
            try
            {
                byte[] eight = new byte[8];

                //Get the selected amount of data
                for (int y = 0; y < s; y += 2)
                {
                    for (int v = 0; v < 8; v++)
                    {
                        eight[v] = (byte)file.ReadByte();
                    }

                    file.Position = 0;

                    uint Left = (uint)(eight[0] << 24 | eight[1] << 16
| eight[2] << 8 | eight[3]);
                    uint Right = (uint)(eight[4] << 24 | eight[5] << 16
| eight[6] << 8 | eight[7]);
                    PlainText[y] = Left;
                    PlainText[y + 1] = Right;
                }
            }
            finally
            {
                file.Close();
            }

            uint[] sArr = new uint[2];
            sArr[0] = s; sArr[1] = 0;
            //uint[] hP = PlainText;
            uint[] hC = GetEmptyData(s/2);
            uint[] returnedC = GetEmptyData(s/2);
            uint[,] SBoxes = new uint[4, 256];
            uint[] SBoxesC = new uint[1040];
            uint[] PArray = new uint[18];
```

```
            SBoxes = fillSBoxes();
            SBoxesC = fillSBoxes(s);
            PArray = fillPArray();

            //Initialize CUDA using first device
            CUDA cuda = new CUDA(0, true);

            //Get kernel functions
            cuda.LoadModule(Path.Combine(Environment.CurrentDirectory,
"blfsh_kernel.cubin"));
            CUfunction Blowfish =
cuda.GetModuleFunction("Blowfish_Encrypt");

            CUdeviceptr dP, dC, pArr, sBox, ds;

            // Create events for performance timing using GPU clock
            CUevent start = cuda.CreateEvent();
            CUevent end = cuda.CreateEvent();
            CUevent cstart = cuda.CreateEvent();
            CUevent cend = cuda.CreateEvent();
            cuda.RecordEvent(start);

            // Allocate device memory and copy host memory to device
            dP = cuda.CopyHostToDevice<uint>(PlainText);
            //dC = cuda.CopyHostToDevice<uint>(hC);
            pArr = cuda.CopyHostToDevice<uint>(PArray);
            sBox = cuda.CopyHostToDevice<uint>(SBoxesC);
            ds = cuda.CopyHostToDevice<uint>(sArr);

            // Set kernel function execution parameters
            int offset = 0;
            cuda.SetParameter(Blowfish, offset, (uint)dP.Pointer);
offset += IntPtr.Size;
            //cuda.SetParameter(Blowfish, offset, (uint)dC.Pointer);
offset += IntPtr.Size;
            cuda.SetParameter(Blowfish, offset, (uint)pArr.Pointer);
offset += IntPtr.Size;
            cuda.SetParameter(Blowfish, offset, (uint)sBox.Pointer);
offset += IntPtr.Size;
            cuda.SetParameter(Blowfish, offset, (uint)ds.Pointer);

            // Launch kernel function
            cuda.SetFunctionBlockShape(Blowfish, BLOCK_SIZE, 1, 1);

            cuda.RecordEvent(cstart);
            cuda.Launch(Blowfish, (int)(s / (2*BLOCK_SIZE)), 1);
            cuda.RecordEvent(cend);

            // Copy device (GPU) result to host (CPU) memory
            cuda.CopyDeviceToHost<uint>(dP, returnedC);

            cuda.RecordEvent(end);
            cuda.SynchronizeContext();


            // Calculate GPU runtime
            float GpuTime = cuda.ElapsedTime(start, end);
```

```
            Console.WriteLine("GPU time:\t\t\t{0} ms\n", GpuTime);

            float GpuCompTime = cuda.ElapsedTime(cstart, cend);
            Console.WriteLine("GPU Computation time:\t\t{0} ms\n",
GpuCompTime);

            float GpuTTime = GpuTime - GpuCompTime;
            Console.WriteLine("GPU Transfer time:\t\t{0} ms\n",
GpuTTime);

            //Time CPU runtime
            System.Diagnostics.Stopwatch CpuTime =
System.Diagnostics.Stopwatch.StartNew();

            hC = BlowfishCPU(PlainText, hC, PArray, SBoxes);

            Console.WriteLine("CPU time:\t\t\t{0} ms\n",
CpuTime.Elapsed.TotalMilliseconds);

            // Free device (GPU) resources
            cuda.UnloadModule();
            cuda.Free(dP);
            cuda.Free(pArr);
            cuda.Free(sBox);
            cuda.Free(ds);
        }

        static void BlowfishCompare(uint t)
        {

            //Generate data (eight bytes X s) s will be total number of
threads
            uint s = 512*t; //65535 is the max number accepted for
number of blocks by cuda.Launch() and so the max value for t
            uint[] sArr = new uint[2];
            sArr[0] = s; sArr[1] = 0;
            uint[] hP = GetData(s);
            uint[] hC = GetEmptyData(s);
            uint[] returnedC = GetEmptyData(s);
            uint[,] SBoxes = new uint[4, 256];
            uint[] SBoxesC = new uint[1040];
            uint[] PArray = new uint[18];
            SBoxes = fillSBoxes();
            SBoxesC = fillSBoxes(s);
            PArray = fillPArray();

            //Initialize CUDA using first device
            CUDA cuda = new CUDA(0, true);

            //Get kernel functions
            cuda.LoadModule(Path.Combine(Environment.CurrentDirectory,
"blfsh_kernel.cubin"));
            CUfunction Blowfish =
cuda.GetModuleFunction("Blowfish_Encrypt");

            CUdeviceptr dP, dC, pArr, sBox, ds;
```

```
            // Create events for performance timing using GPU clock
            CUevent start = cuda.CreateEvent();
            CUevent end = cuda.CreateEvent();
            CUevent cstart = cuda.CreateEvent();
            CUevent cend = cuda.CreateEvent();
            cuda.RecordEvent(start);

            // Allocate device memory and copy host memory to device
            dP = cuda.CopyHostToDevice<uint>(hP);
            dC = cuda.CopyHostToDevice<uint>(hC);
            pArr = cuda.CopyHostToDevice<uint>(PArray);
            sBox = cuda.CopyHostToDevice<uint>(SBoxesC);
            ds = cuda.CopyHostToDevice<uint>(sArr);

            // Set kernel function execution parameters
            int offset = 0;
            cuda.SetParameter(Blowfish, offset, (uint)dP.Pointer);
offset += IntPtr.Size;
            cuda.SetParameter(Blowfish, offset, (uint)dC.Pointer);
offset += IntPtr.Size;
            cuda.SetParameter(Blowfish, offset, (uint)pArr.Pointer);
offset += IntPtr.Size;
            cuda.SetParameter(Blowfish, offset, (uint)sBox.Pointer);
offset += IntPtr.Size;
            cuda.SetParameter(Blowfish, offset, (uint)ds.Pointer);

            // Launch kernel function
            cuda.SetFunctionBlockShape(Blowfish, BLOCK_SIZE, 1, 1);

            cuda.RecordEvent(cstart);
            cuda.Launch(Blowfish, (int)(s/BLOCK_SIZE), 1);
            cuda.RecordEvent(cend);

            // Copy device (GPU) result to host (CPU) memory
            cuda.CopyDeviceToHost<uint>(dC, returnedC);

            cuda.RecordEvent(end);
            cuda.SynchronizeContext();


            // Calculate GPU runtime
            float GpuTime = cuda.ElapsedTime(start, end);
            Console.WriteLine("GPU time:\t\t\t{0} ms\n", GpuTime);

            float GpuCompTime = cuda.ElapsedTime(cstart, cend);
            Console.WriteLine("GPU Computation time:\t\t{0} ms\n",
GpuCompTime);

            float GpuTTime = GpuTime - GpuCompTime;
            Console.WriteLine("GPU Transfer time:\t\t{0} ms\n",
GpuTTime);

            //Time CPU runtime
            System.Diagnostics.Stopwatch CpuTime =
System.Diagnostics.Stopwatch.StartNew();

            hC = BlowfishCPU(hP,hC,PArray,SBoxes);
```

```
            Console.WriteLine("CPU time:\t\t\t{0} ms\n",
CpuTime.Elapsed.TotalMilliseconds);

            // Free device (GPU) resources
            cuda.UnloadModule();
            cuda.Free(dP);
            cuda.Free(pArr);
            cuda.Free(sBox);
            cuda.Free(ds);
        }

        static uint[] BlowfishCPU(uint[] plain, uint[] crypt, uint[] p,
uint[,] s)
        {
            int plnSize = plain.Length;
            Console.WriteLine(plnSize);
            int N = 16; //N defines the number of rounds

            //Iterate through each block of plaintext
            for (int i = 0; i < plnSize; i+=2)
            {
                //64 bit block split into left and right halves
                uint Xl = plain[i];
                uint Xr = plain[i+1];
                uint temp = 0;

                //Iterate through each round of Blowfish
                for (int k = 0; k < N; k++)
                {
                    Xl = Xl ^ p[k];

                    //Function F start
                    uint a, b, c, d;
                    uint  y;

                    d = (uint)(Xl & 0xFF);
                    Xl >>= 8;
                    c = (uint)(Xl & 0xFF);
                    Xl >>= 8;
                    b = (uint)(Xl & 0xFF);
                    Xl >>= 8;
                    a = (uint)(Xl & 0xFF);
                    y = s[0,a] + s[1,b];
                    y = y ^ s[2,c];
                    y = y + s[3,d];
                    //function F end

                    // y is the result from function F
                    Xr = y ^ Xr;

                    temp = Xl;
                    Xl = Xr;
                    Xr = temp;
                }

                temp = Xl;
```

```
                Xl = Xr;
                Xr = temp;

                Xr = Xr ^ p[N];
                Xl = Xl ^ p[N + 1];

                crypt[i] = Xl;
                crypt[i + 1] = Xr;
            }
            return (crypt);
        }

        static void TestBlfsh()
        {
            //Generate data (eight bytes X s) s will be total number of
threads
            uint s = 1; //65535 is the max number accepted for number
of blocks by cuda.Launch() and so the max value for t
            uint[] sArr = new uint[2];
            sArr[0] = s; sArr[1] = 0;
            uint[] hP = GetData(s);
            uint[] hC = GetEmptyData(s);
            uint[] returnedC = GetEmptyData(s);
            uint[,] SBoxes = new uint[4, 256];
            uint[] SBoxesC = new uint[1040];
            uint[] PArray = new uint[18];
            SBoxes = fillSBoxes();
            SBoxesC = fillSBoxes(s);
            PArray = fillPArray();

            hP = BlowfishCPU(hP, hC, PArray, SBoxes);

            //Initialize CUDA using first device
            CUDA cuda = new CUDA(0, true);

            //Get kernel functions
            cuda.LoadModule(Path.Combine(Environment.CurrentDirectory,
"blfsh_kernel.cubin"));
            CUfunction Blowfish =
cuda.GetModuleFunction("Test_Blowfish_Encrypt");

            CUdeviceptr dP, dC, pArr, sBox, ds;

            // Allocate device memory and copy host memory to device
            dP = cuda.CopyHostToDevice<uint>(hP);
            dC = cuda.CopyHostToDevice<uint>(hC);
            pArr = cuda.CopyHostToDevice<uint>(PArray);
            sBox = cuda.CopyHostToDevice<uint>(SBoxesC);
            ds = cuda.CopyHostToDevice<uint>(sArr);
            //dC = cuda.Allocate<uint>(hC);

            // Set kernel function execution parameters
            int offset = 0;
            cuda.SetParameter(Blowfish, offset, (uint)dP.Pointer);
offset += IntPtr.Size;
            cuda.SetParameter(Blowfish, offset, (uint)dC.Pointer);
offset += IntPtr.Size;
```

```
            cuda.SetParameter(Blowfish, offset, (uint)pArr.Pointer);
offset += IntPtr.Size;
            cuda.SetParameter(Blowfish, offset, (uint)sBox.Pointer);
offset += IntPtr.Size;
            cuda.SetParameter(Blowfish, offset, (uint)ds.Pointer);

            // Launch kernel function
            cuda.SetFunctionBlockShape(Blowfish, 8, 1, 1);
            cuda.Launch(Blowfish, 1, 1);

            // Copy device (GPU) result to host (CPU) memory
            cuda.CopyDeviceToHost<uint>(dC, returnedC);
            cuda.SynchronizeContext();

            // CPU version test run
            //hC = BlowfishCPU(hP, hC, PArray, SBoxes);

            // Make sure the GPU and CPU results agree
            bool flagit = true;
            for (int g = 0; g < s; g++)
            {
                if (hC[g] != returnedC[g])
                {
                    flagit = false;
                    Console.WriteLine("FAIL");
                    break;
                }
            }

            if (flagit)
            {
                Console.WriteLine("PASS");
            }

            //Free CUDA resources
            cuda.UnloadModule();
            cuda.Free(dP);
            cuda.Free(dC);
            cuda.Free(pArr);
            cuda.Free(sBox);
            cuda.Free(ds);
        }
    }
}
```

C. CUDA IMPLEMENTATION

Knowing how CUDA allows a programmer to fully use a GPU to the fullest possible extent for computation is necessary to grasp how to program a serial algorithm as a parallel algorithm and then to improve the code. First, it

is productive to note that there is a possibility that a certain algorithm may not exhibit characteristics that allow it to be programmed in a very parallel manner. In this case it may be that there are pieces that could be considered parallel. Even if this is not the case it is possible that a translated version of the CPU code may be accelerated simply by use of the GPU. However, using knowledge of CUDA, it is normally possible to see large, particularly computationally intensive, or repetitive pieces of code. If the complexity of these portions is great enough there is a high chance that parallelism exists. Even if it is not immediately obvious, some perspective will allow a programmer to find the parallelism.

Identifying parallelism is the first step in programming properly for CUDA. Blowfish, like other block ciphers, exhibits parallelism. Certainly there are parts that cannot be made parallel. For one, the 16 rounds in the encryption and decryption itself must be done in a specific, sequential order. If a single round of the cipher is to be completed by one thread, then 16 threads would each work on a single block of data. This does not exhibit parallelism in the sense that it is a MISD (Multiple Instruction Single Data)

operation. To exhibit the type of parallelism that can be exploited by the GPU architecture it would be beneficial to find a SIMD (Single Instruction Multiple Data) operation. Blowfish does exhibit parallelism of this form by block of data. Because each block goes through the same process in a sequential order (a single sequential set of instructions for each of the many blocks of data) it is possible to write a parallel algorithm where each block of data would be encrypted by a single thread. This experiment exploits the data-level parallelism exhibited in Blowfish. Other instances of parallelism may exist within the Blowfish algorithm; however, this is the first attempt at finding an area exhibiting parallel traits to exploit in this manner. Future attempts may bring more promising results. Some ideas drawn from this experiment are noted in the Chapter V CONCLUSIONS AND FUTURE WORKS of this thesis as possible paths for future research.

The second step to properly program an algorithm for CUDA is deciding how to divide the workload, by defining what is to be done by each thread. It is best in this instance to allow each thread to do all operations on one entire block of data. There are two major factors that

influence this decision; first, the size of the data being operated upon, second, the types of operations being done to the data. It is appropriate that a single thread do all work for one block of data because the data size is small at 64 bits, and each operation done to the data is efficiently executed by the GPU(most operations are AND, OR, or XOR).

Because each thread does all of the work for one block of data the third step, coding, is simplified somewhat. Due to this fact most of the CPU Blowfish function code can be recycled. The only decision left at this point is thread-block size. Because each SM (Streaming Multiprocessor) works on one thread-block of data at one time and contains eight SP (Streaming Processors) a thread-block containing eight threads will occupy the entire SM, albeit for a short time before moving onto the next thread-block. This experiment has revealed that a small thread-block size severely limits the size of the data that can be processed. Therefore, it uses thread-blocks containing 512 threads, the maximum number allowable for the GPU used. These settings are issued in the following two lines of code taken from the CPU BlowfishCompare function. Both are determined before computation begins on the GPU and therefore are set in CPU code. Cuda.SetFunctionBlockShape

accepts the number of threads per thread-block as the second parameter and cuda.Launch accepts the total number of thread-blocks used to process all data.

```
1)  cuda.SetFunctionBlockShape(Blowfish, BLOCK_SIZE, 1, 1);
2)  cuda.Launch(Blowfish, (int)(s/BLOCK_SIZE), 1);
```

As discussed earlier, CUDA is implemented as an extension of the C programming language. Below is the kernel function implementation of Blowfish used in this experiment. It is important to note that all accesses to kernel function parameters are global memory accesses. Chapter V CONCLUSIONS AND FUTURE WORKS discusses changes to this, including the use of shared memory. In general this works in the same way as the CPU implementation except for the fact that in the CUDA code a thread takes the place of an iteration through the outer for loop in the CPU code. Another small difference is the transformation of the two-dimensional S-Box configuration in CPU code to its one-dimensional CUDA counterpart. This decision was made for a simpler transfer of the S-Boxes to GPU memory.

blfsh_kernel.cu

```
// Blowfish Kernel function

#ifndef _BLOWFISH_KERNEL_H_
#define _BLOWFISH_KERNEL_H_

#include <stdio.h>

__device__ unsigned long cipher[];
```

```
extern "C" __global__ void Blowfish_Encrypt(unsigned long* plain,
unsigned long* P, unsigned long* S, unsigned long size){
  //block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

  //thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

  //right and left halves' indeces
      int idxl = bx*16 + tx*2;
      int idxr = bx*16 + tx*2 + 1;

  //N number of rounds
      int N = 16;

    unsigned long Xl;
    unsigned long Xr;
    unsigned long temp;
    short i;

    Xl = *(plain + idxl);
    Xr = *(plain + idxr);
      //Synchronize so that data is loaded
      __syncthreads();

  for (i = 0; i < 16; ++i) {
   Xl = Xl ^ P[i];

   //Function F start
   unsigned short a, b, c, d;
   unsigned int y;

   d = (unsigned short)(Xl & 0xFF);
   Xl >>= 8;
   c = (unsigned short)(Xl & 0xFF);
   Xl >>= 8;
   b = (unsigned short)(Xl & 0xFF);
   Xl >>= 8;
   a = (unsigned short)(Xl & 0xFF);
   y = S[a] + S[256+b];
   y = y ^ S[512+c];
   y = y + S[768+d];//function F end

   // y in place of function F
   Xr = y ^ Xr;

   temp = Xl;
   Xl = Xr;
   Xr = temp;
  }

  temp = Xl;
  Xl = Xr;
  Xr = temp;
```

```
  Xr = Xr ^ P[N];
  Xl = Xl ^ P[N + 1];
  __syncthreads();
}


extern "C" __global__ void Test_Blowfish_Encrypt(unsigned long* plain,
unsigned long* cipher,unsigned long* P, unsigned long* S, unsigned long
size){
  //block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

  //thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

  //right and left halves' indeces
      int idxl = bx*16 + tx*2;
      int idxr = bx*16 + tx*2 + 1;

  //N number of rounds
      int N = 16;

    unsigned long Xl;
    unsigned long Xr;
    unsigned long temp;
    short i;

    Xl = *(plain + idxl);
    Xr = *(plain + idxr);
      //Synchronize so that data is loaded
      __syncthreads();

  for (i = 0; i < 16; ++i) {
   Xl = Xl ^ P[i];

   //Function F start
   unsigned short a, b, c, d;
   unsigned int y;

   d = (unsigned short)(Xl & 0xFF);
   Xl >>= 8;
   c = (unsigned short)(Xl & 0xFF);
   Xl >>= 8;
   b = (unsigned short)(Xl & 0xFF);
   Xl >>= 8;
   a = (unsigned short)(Xl & 0xFF);
   y = S[a] + S[256+b];
   y = y ^ S[512+c];
   y = y + S[768+d];//function F end

   // y in place of function F
   Xr = y ^ Xr;

   temp = Xl;
   Xl = Xr;
```

```
    Xr = temp;
  }

  temp = Xl;
  Xl = Xr;
  Xr = temp;

  Xr = Xr ^ P[N];
  Xl = Xl ^ P[N + 1];
  __syncthreads();
}

#endif
```

The following are steps to execute this experiment using Visual studio 2008 and CUDA 2.2:

1) Create a C# project in Visual Studio 2008.
2) Copy Source Code found earlier in this chapter into the project.
3) Match all project references shown in FIGURE 6 below. This must be done for the CUDA.NET “using” statements in Program.cs.
4) Locate the C/C++ compiler executable on your system. Click on Project in the Visual Studio Menubar and then click on ProjectName Properties(probably the last entry in the list).
5) Go to the Pre/Post-build events tab and in the Post-Build events text box type the following: nvcc blfsh_kernel.cu --cubin --compiler-bindir="C:\Program Files\Microsoft Visual Studio 9.0\VC\bin"
6) Match the Directory Structure of the project shown in FIGURE 7 below.

7) Now build and execute the program.

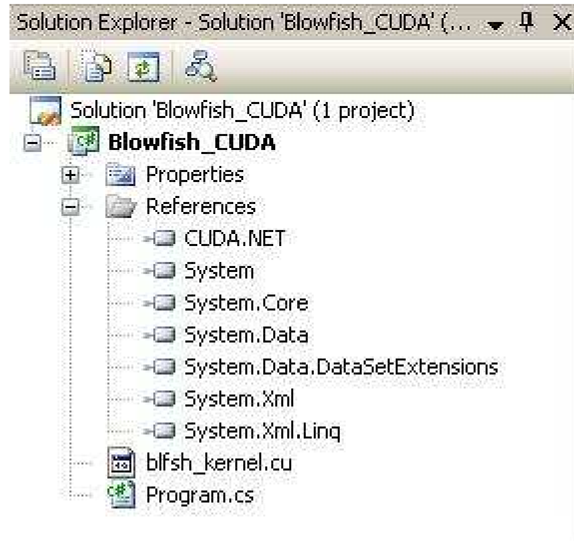

FIGURE 6 - Blowfish_CUDA References

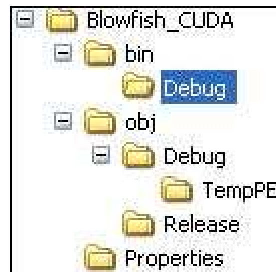

FIGURE 7 - Blowfish_CUDA Project Directory Structure

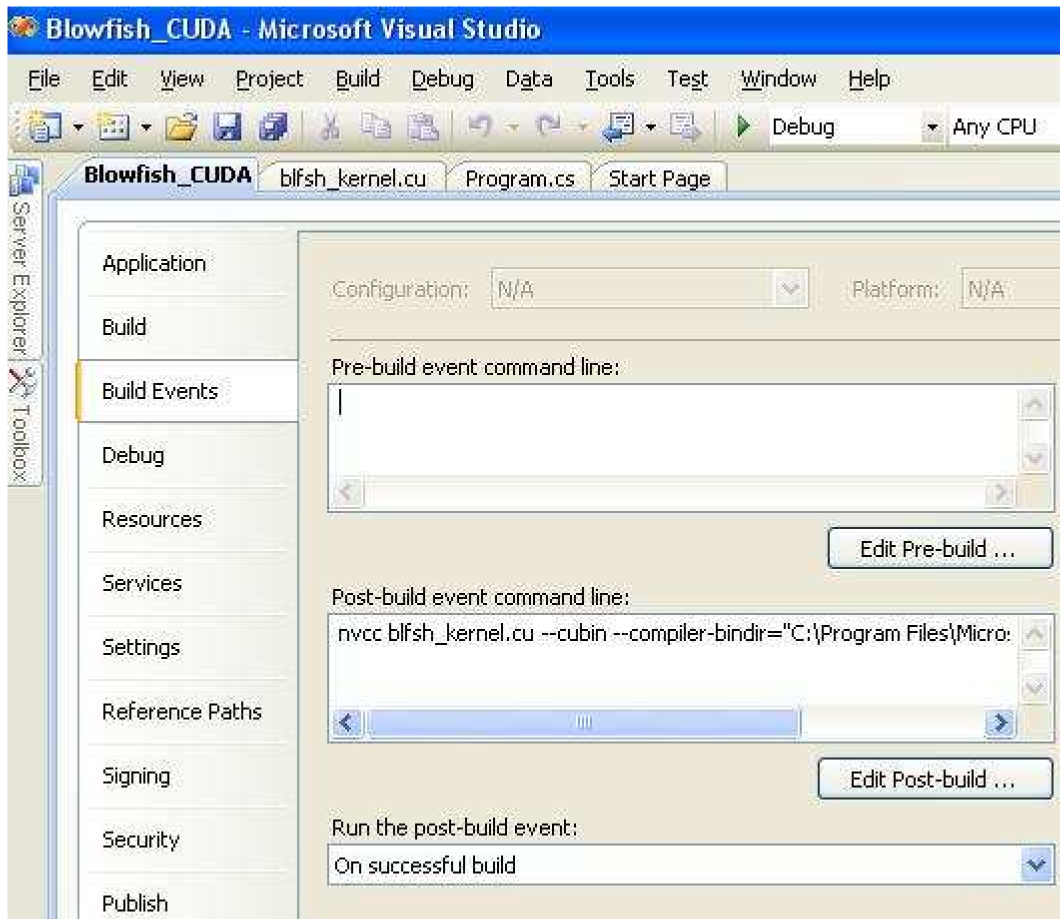

FIGURE 8 - Post-Build Event Visual

CHAPTER IV

RESULTS AND DISCUSSION OF RESULTS

TABLE I

CPU AND GPU RUNTIME FOR VARIOUS DATA SIZES

Number of Thread Blocks	Size (KB)	GPU Transfer Time (ms)	GPU Computation Time (ms)	GPU Runtime (ms)	CPU Runtime (ms)	Speed Up Factor
65535	262140	209.055	0.513056	209.5681	21218.35	101.248
32000	128000	104.1726	0.25872	104.4313	10432.79	99.90099
16000	64000	52.39123	0.131168	52.5224	5092.851	96.96531
8000	32000	41.41222	0.082688	41.49491	2435.688	58.69849
4000	16000	22.30045	0.03872	22.33917	1259.037	56.36008
2000	8000	12.43674	0.022752	12.45949	606.6368	48.68873
1000	4000	7.630848	0.015232	7.64608	305.3618	39.93704
500	2000	5.157984	0.011456	5.16944	162.2104	31.37872
100	400	2.422176	0.008064	2.43024	33.189	13.65668
50	200	2.127232	0.008384	2.135616	16.2128	7.591627
25	100	1.92288	0.007808	1.930688	7.7382	4.008001
1	4	1.688256	0.007232	1.695488	0.4038	0.238162

TABLE II

PERFORMANCE MEASURES FOR 65535 THREAD-BLOCKS

Run #	GPU Transfer Time (ms)	GPU Computation Time (ms)	GPU Runtime (ms)	CPU Runtime (ms)	Speed Up Factor
1	216.9714	0.606336	217.577736	21096.2671	96.95967744
2	209.055	0.513056	209.568056	21218.3452	101.2479936
3	209.5557	0.513024	210.068724	23566.2291	112.1834257
4	209.0928	0.513088	209.605888	20995.928	100.1685983
5	209.6069	0.513216	210.120116	20873.2834	99.33976716
6	209.4368	0.513248	209.950048	20991.7223	99.98436533
7	209.2536	0.51296	209.76656	21077.3445	100.4800026
8	209.1368	0.51408	209.65088	20987.6839	100.1077787
9	209.3771	0.513952	209.891052	20996.2555	100.0340667
10	209.1601	0.512992	209.673092	21025.8293	100.2791016

TABLE III
PERFORMANCE MEASURES FOR 32000 THREAD-BLOCKS

Run #	GPU Transfer Time (ms)	GPU Computation Time (ms)	GPU Runtime (ms)	CPU Runtime (ms)	Speed Up Factor
1	111.237	0.347104	111.584104	11369.3434	101.89035
2	103.9436	0.254336	104.197936	10888.8625	104.50171
3	103.3067	0.254144	103.560844	13232.3277	127.773463
4	103.5707	0.254368	103.825068	12050.6759	116.067113
5	103.2256	0.25456	103.48016	12047.9979	116.428095
6	104.1726	0.25872	104.43132	10432.7921	99.9009885
7	103.7636	0.254176	104.017776	10446.5671	100.430595
8	103.1958	0.25536	103.45116	12291.4898	118.814422
9	103.0238	0.25552	103.27932	12768.6121	123.631837
10	103.6873	0.25424	103.94154	13000.4769	125.074892

TABLE IV
PERFORMANCE MEASURES FOR 16000 THREAD-BLOCKS

Run #	GPU Transfer Time (ms)	GPU Computation Time (ms)	GPU Runtime (ms)	CPU Runtime (ms)	Speed Up Factor
1	60.60598	0.224576	60.830556	5680.6046	93.3840651
2	52.71184	0.130752	52.842592	6131.1593	116.026846
3	52.48378	0.130816	52.614596	6367.2855	121.017474
4	52.96282	0.13088	53.0937	6127.8728	115.416194
5	52.52346	0.131232	52.654692	6011.1833	114.162349
6	52.39123	0.131168	52.522398	5092.8507	96.9653118
7	52.98938	0.13168	53.12106	5223.4985	98.3319704
8	52.42531	0.132096	52.557406	5991.0949	113.99145
9	52.39341	0.132	52.52541	5570.661	106.056497
10	52.95222	0.131008	53.083228	6513.1562	122.697064

TABLE V
PERFORMANCE MEASURES FOR 8000 THREAD-BLOCKS

Run #	GPU Transfer Time (ms)	GPU Computation Time (ms)	GPU Runtime (ms)	CPU Runtime (ms)	Speed Up Factor
1	49.01427	0.16464	49.17891	2430.0354	49.4121444
2	40.79958	0.069024	40.868604	2506.2436	61.324424
3	40.78259	0.068928	40.851518	2618.539	64.09894
4	41.36323	0.069248	41.432478	2509.1531	60.5600539
5	40.79446	0.06928	40.86374	2507.107	61.3528522
6	40.92758	0.069216	40.996796	2637.9763	64.3459138
7	41.41222	0.082688	41.494908	2435.6884	58.698489
8	40.84682	0.07008	40.9169	2435.2913	59.5179816
9	42.37037	0.07024	42.44061	2754.5848	64.9044583
10	41.41485	0.06896	41.48381	2443.9241	58.9127204

TABLE VI
PERFORMANCE MEASURES FOR 4000 THREAD-BLOCKS

Run #	GPU Transfer Time (ms)	GPU Computation Time (ms)	GPU Runtime (ms)	CPU Runtime (ms)	Speed Up Factor
1	29.95341	0.133152	30.086562	1245.0502	41.3822689
2	21.7072	0.038272	21.745472	1248.33925	57.4068592
3	22.05629	0.0384	22.09469	1297.9654	58.745581
4	22.30045	0.03872	22.33917	1259.0373	56.3600752
5	21.84291	0.03808	21.88099	1255.9943	57.4011642
6	21.62851	0.038016	21.666526	1347.2964	62.1833145
7	22.25155	0.038016	22.289566	1209.424	54.2596478
8	21.73011	0.038016	21.768126	1211.9089	55.6735522
9	21.69107	0.038048	21.729118	1337.3598	61.5468976
10	22.50925	0.038496	22.547746	1217.0087	53.9747388

TABLE VII
PERFORMANCE MEASURES FOR 2000 THREAD-BLOCKS

Run #	GPU Transfer Time (ms)	GPU Computation Time (ms)	GPU Runtime (ms)	CPU Runtime (ms)	Speed Up Factor
1	20.0951	0.121664	20.216764	611.8345	30.2637207
2	12.13632	0.022656	12.158976	603.9169	49.6684014
3	11.89958	0.022624	11.922204	658.9251	55.2687322
4	12.46518	0.02288	12.48806	628.9473	50.3638916
5	12.25766	0.0232	12.28086	625.9028	50.9657141
6	12.00963	0.023246	12.032876	655.4972	54.4755219
7	12.43674	0.022752	12.459492	606.6368	48.6887266
8	12.09114	0.023648	12.114788	605.1907	49.9547082
9	11.91776	0.024032	11.941792	657.904	55.092569
10	12.46051	0.023968	12.484478	609.8583	48.8493231

TABLE VIII
PERFORMANCE MEASURES FOR 1000 THREAD-BLOCKS

Run #	GPU Transfer Time (ms)	GPU Computation Time (ms)	GPU Runtime (ms)	CPU Runtime (ms)	Speed Up Factor
1	14.9841	0.107456	15.091556	305.5179	20.2442942
2	6.93904	0.014848	6.953888	303.6866	43.6714828
3	6.981312	0.014976	6.996288	333.2804	47.6367468
4	7.630848	0.015232	7.64608	305.3618	39.9370396
5	6.983072	0.0152	6.998272	303.3122	43.3410133
6	7.048288	0.015232	7.06352	3317594	469679.99
7	7.499008	0.015872	7.51488	304.5999	40.5329027
8	6.940768	0.016096	6.956864	304.6862	43.7964865
9	7.095968	0.016256	7.112224	330.2393	46.4326349
10	7.620416	0.015136	7.635552	301.563	39.4945906

TABLE IX
PERFORMANCE MEASURES FOR 500 THREAD-BLOCKS

Run #	GPU Transfer Time (ms)	GPU Computation Time (ms)	GPU Runtime (ms)	CPU Runtime (ms)	Speed Up Factor
1	12.46422	0.102112	12.566332	163.0478	12.9749715
2	4.541376	0.01104	4.552416	155.9118	34.2481443
3	5.041824	0.011072	5.052896	167.5094	33.1511672
4	4.521344	0.013152	4.534496	170.4574	37.591256
5	4.614592	0.011232	4.625824	153.0522	33.0864728
6	4.579616	0.011072	4.590688	154.5119	33.6576783
7	5.157984	0.011456	5.16944	162.2104	31.378718
8	4.423072	0.011264	4.434336	171.4318	38.6600835
9	4.543968	0.01104	4.555008	152.8975	33.5669004
10	4.467648	0.011296	4.478944	154.8735	34.5781282

TABLE X
PERFORMANCE MEASURES FOR 100 THREAD-BLOCKS

Run #	GPU Transfer Time (ms)	GPU Computation Time (ms)	GPU Runtime (ms)	CPU Runtime (ms)	Speed Up Factor
1	10.3657	0.099872	10.465572	32.4576	3.10136895
2	2.376192	0.00816	2.384352	33.1286	13.8941733
3	2.37952	0.008256	2.387776	34.0425	14.2569906
4	2.430464	0.008128	2.438592	33.029	13.5442911
5	2.410528	0.008032	2.41856	33.5599	13.8759841
6	2.3888	0.008448	2.397248	32.8621	13.7082605
7	2.391296	0.00816	2.399456	41.4784	17.286585
8	2.422176	0.008064	2.43024	33.189	13.6566759
9	2.387456	0.008256	2.395712	33.162	13.8422315
10	2.391488	0.008224	2.399712	33.5271	13.9713016

TABLE XI
PERFORMANCE MEASURES FOR 50 THREAD-BLOCKS

Run #	GPU Transfer Time (ms)	GPU Computation Time (ms)	GPU Runtime (ms)	CPU Runtime (ms)	Speed Up Factor
1	10.0415	0.09952	10.14102	16.644	1.64125502
2	2.087296	0.007712	2.095008	16.5233	7.88698659
3	2.084992	0.007904	2.092896	16.7076	7.98300537
4	2.064128	0.008	2.072128	16.3162	7.87412747
5	2.099616	0.007744	2.10736	19.4423	9.2259035
6	2.074048	0.00784	2.081888	16.3739	7.86492837
7	2.110336	0.008128	2.118464	20.4599	9.65789364
8	2.127232	0.008384	2.135616	16.2128	7.59162696
9	2.084128	0.007872	2.092	16.0113	7.65358509
10	2.125344	0.011456	2.1368	16.3656	7.65892924

TABLE XII
PERFORMANCE MEASURES FOR 25 THREAD-BLOCKS

Run #	GPU Transfer Time (ms)	GPU Computation Time (ms)	GPU Runtime (ms)	CPU Runtime (ms)	Speed Up Factor
1	9.872128	0.102592	9.97472	8.3416	0.8362741
2	1.885152	0.007552	1.892704	8.3234	4.39762372
3	1.895328	0.007744	1.903072	8.6326	4.53613946
4	1.92288	0.007808	1.930688	7.7382	4.00800129
5	1.8984	0.007648	1.906048	8.5171	4.4684604
6	1.917888	0.007968	1.925856	8.2932	4.30624097
7	1.88992	0.008032	1.897952	8.6742	4.57029472
8	1.940064	0.007712	1.947776	8.2256	4.22307288
9	1.90576	0.007712	1.913472	8.2335	4.30291115
10	1.94544	0.007488	1.952928	8.3534	4.27737223

TABLE XIII
PERFORMANCE MEASURES FOR ONE THREAD-BLOCK

Run #	GPU Transfer Time (ms)	GPU Computation Time (ms)	GPU Runtime (ms)	CPU Runtime (ms)	Speed Up Factor
1	9.84352	0.100512	9.944032	0.6385	0.06420937
2	1.682272	0.007232	1.689504	0.3348	0.19816467
3	1.696768	0.007296	1.704064	0.3151	0.1849109
4	1.69152	0.007232	1.698752	0.3298	0.19414252
5	1.9824	0.0072	1.9896	0.3398	0.1707881
6	1.688256	0.007232	1.695488	0.4038	0.23816152
7	1.727008	0.007264	1.734272	0.3188	0.18382353
8	1.717472	0.007232	1.724704	0.3424	0.19852682
9	1.692544	0.007232	1.699776	0.344	0.20237961
10	1.773824	0.00736	1.781184	0.3905	0.21923619

*All tables reflect values obtained using the following hardware: CPU: AMD Phenom II X3 720 2.80 GHz clock
GPU: Nvidia Tesla C1060 240 cores 1.296 GHz core clock

TABLE I shows a sample run of CPU and GPU runtimes for different data sizes. The speed up factor, determined by the quotient of CPU runtime and GPU runtime, is the most important metric to show improvement through the use of CUDA. Other metrics may show information more valuable for determining efficiency and usage data. This is not the goal of this experiment, and would be more important with larger data sets where occupancy would play a role in the runtime.

There are two very important patterns to note in TABLE I. First, notice that the speed up is directly related to the size of the data being processed. Second, notice that

most of the time the GPU spends is in data transfer or preprocessing. Actual GPU computation times are very small. Perhaps this gives a clue to the programmer where improvement is possible. However, it is possible that the programmer cannot control this metric. More analysis needs to be done to be certain.

TABLE I shows the maximum data size that can be processed by this implementation. This size limit is due to restrictions of CUDA based on hardware constraints as well as numerical representation limits in C. For the model of GPU used during programming and data collection thread-block size is limited to 512 threads. In combination with this the software function that launches a CUDA kernel accepts a thread-block size parameter of type int, which is limited to a maximum value of 65,535 in C, the main interface language for CUDA. This results in a maximum number of 33,553,920 threads (512 threads per thread-block times 65,535). Because each thread processes 64 bits (eight bytes) of data, a maximum number of 262,140 Kilobytes of data can be processed on this particular GPU. More advanced GPUs may have higher limits on thread-block size. Therefore, larger datasets could be processed, probably yielding even higher degrees of speedup.

Each of the TABLES II through XIII shows the results from multiple runs for the same size of randomly generated data. Notice that each of the first runs yields somewhat higher GPU times than subsequent iterations. This shows that there is a certain "warm-up" time associated with the GPU. For large data sets this time is fairly negligible, but it remains constant and is therefore more consequential for transfer of smaller data.

FIGURE 9 shown below illustrates the main result of this experiment. Speed-up is heavily impacted by the size of the data processed. It is not a linear relationship. As the size of the data doubles the percentage increase in the speed-up factor is decreasing in a shallow exponential pattern.

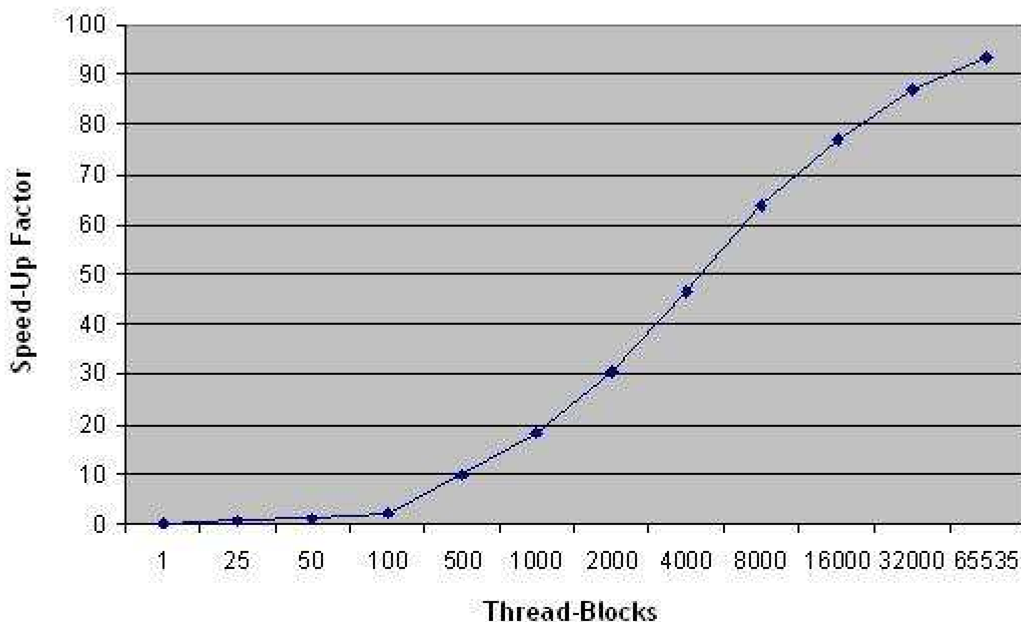

FIGURE 9 – Relationship Between Speed-Up Factor And Number Of Thread-Blocks

FIGURE 10 below is a plot of the GPU and CPU runtimes from TABLE I versus the number of thread-blocks used. This is a great visualization of the amount of time that is actually saved using the parallel processing of CUDA and the GPU.

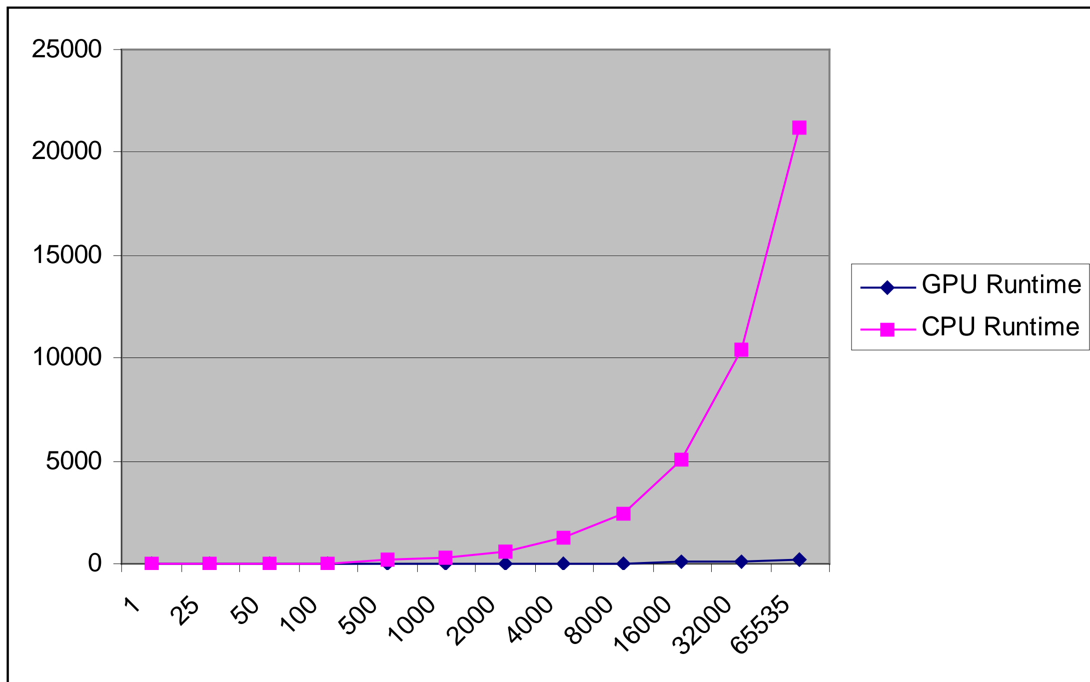

FIGURE 10 - CPU And GPU Runtimes Vs. Thread-Block Numbers

CHAPTER V

CONCLUSIONS AND FUTURE WORKS

A. CONCLUSIONS

Overall this experiment was successful. It shows that, specifically Blowfish, but potentially other related algorithms can be accelerated using a GPU and CUDA. It is apparent that even more can be done to expand and extend this experiment. Greater acceleration is possible. It is necessary to do more analysis in order to work toward the optimum setup to obtain the best results. The rest of this section focuses on possibilities for obtaining a better result. These have not been attempted for this experiment, but would be good ideas for beginning further research.

B. SHARED MEMORY

Chapter II GPGPU AND CUDA mentioned the use of shared memory. The implementation used in this analysis does not make use of shared memory. Integrating shared memory usage into this project should incur few changes. Using shared memory decreases computation time because it cuts out expensive transfers from global memory. Shared memory exhibits transfer speeds similar to registers. Global memory transfers can take many more clock cycles.

C. THREAD-BLOCK SIZE CONSIDERATIONS

Another area that shows promise for improvement is thread-block size. The explanation given for choosing a thread-block size of 512 threads was made in order to process the largest possible size data. This decision gave little consideration to efficient hardware usage other than the general truth that the more data transferred from host to device at one time saves on transfer time.

D. CONSTANT MEMORY

CUDA also allows the use of a special memory space within global device memory called constant memory. Although this is located within global memory after the first use it is cached. In this specific experiment constant memory could be used to store the P-Array and S-Boxes. Both the P-Array and S-Boxes are accessed multiple times from global memory for each block of data that is encrypted. Therefore using constant memory in this way could further accelerate computation times.

E. DEVICE MEMORY TRANSFER AND ALLOCATION

At this time there is very little to do with the transfer of data from the host to device or allocation of device memory that is under the programmer’s control. However, it is obvious that for this experiment this cannot be ignored. CUDA and GPUs are growing and changing often

so it is a possibility that at a later time even this could be coded in such a way as to accelerate transfer and allocation. It is obvious from the results of this experiment that the bulk of time spent with the GPU is derived from the transfer of data. So for this and other applications like it development of a less time consuming data transfer would be critical in creating an optimized parallel version.

REFERENCES

[1]*About GPGPU.org,* http://gpgpu.org/about/, accessed on 10 December 2009.

[2]*NVIDIA CUDA Programming Guide*, version 2.2.1, available online from http://developer.download.nvidia.com/compute/cuda/2_21/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.1.pdf (26 May 2009), accessed on 20 November 2009, p. 3.

[3]Ibid.

[4]Diagram from ECE498AL UIUC, *Lecture 2 CUDA Spring 2009,*available online from http://courses.ece.illinois.edu/ece498/al/Syllabus.html, accessed 20 November 2009, p. 11.

[5]Ibid.

[6]*Fast Software Encryption, Cambridge Security Workshop Proceedings,* available online from http://www.schneier.com/paper-blowfish-fse.html *(December 1993)*, Springer-Verlag, 1994, pp. 191-204, accessed on 07 December 2009.

VITA

Joe Feist

1800 S. 2nd St. A7 joefeist@gmail.com
Louisville, KY 40208

EDUCATION

Masters of Engineering, Computer Engineering and Computer Science, expected December 2009.
University of Louisville, Louisville, Kentucky.
Thesis: INCREASING PERFORMANCE OF BLOWFISH ENCRYPTION USING CUDA
Directors: Dr. Dar-jen Chang and Dr. Ahmed Desoky
Committee Member: Dr. John F. Naber

Bachelor of Science, Computer Engineering and Computer Science, May 2009. University of Louisville, Louisville, Kentucky.

HONORS & AWARDS

Trustees Presidential Scholarship, August 2004 – May 2009.

National S.M.A.R.T. Grant, 2006 – 2008.

INTERESTS

Web Application Development

General Purpose GPU Programming