AMD FIRESTREAM 9170 GPGPU IMPLEMENTATION OF A BROOK+
PARALLEL ALGORITHM TO SEARCH FOR COUNTEREXAMPLES TO BEAL’S CONJECTURE

A Thesis
Presented to the
Faculty of
San Diego State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Nirav Shah
Summer 2010
SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Project of Nirav Shah:

AMD Firestream 9170 GPGPU Implementation of a Brook+ Parallel Algorithm
to Search for Counterexamples to Beal’s Conjecture

William Root, Chair
Department of Computer Science

Carl Eckberg
Department of Computer Science

Robert Grone
Department of Mathematics and Statistics

7/28/2010
Approval Date
DEDICATION

I dedicate my work to my parents for their support and encouragement.
ABSTRACT OF THE THESIS

AMD Firestream 9170 GPGPU Implementation of a Brook+ Parallel Algorithm to Search for Counterexamples to Beal’s Conjecture
by
Nirav Shah
Master of Science in Computer Science
San Diego State University, 2010

Beal’s Conjecture states that if $A^x + B^y = C^z$ for integers $A,B,C > 0$ and integers $x,y,z > 2$, then $A$, $B$, and $C$ must share a common factor. Norvig and others have shown that the conjecture holds for $A,B,C,x,y,z < 1000$, but the truth of the general conjecture remains unresolved. Extending the search for a counterexample to significantly greater values of the conjecture’s six integer parameters is a task ideally suited to the use of an SIMD parallel algorithm implemented on a GPGPU platform. This thesis project encompassed the design, coding, and testing of such an algorithm, implemented in the Brook+ language for execution on the AMD Firestream 9170 GPGPU.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF TERMS</td>
<td>ix</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Purpose</td>
<td>1</td>
</tr>
<tr>
<td>1.2 History</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Why Use ATI Stream Computing for Computing Beal’s Conjecture Counter Examples</td>
<td>2</td>
</tr>
<tr>
<td>1.4 Stream Computing</td>
<td>2</td>
</tr>
<tr>
<td>2 THE STRUCTURE OF THE DEVICE</td>
<td>3</td>
</tr>
<tr>
<td>2.1 The Stream Processor</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Thread Processing</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Flow Control</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Memory Architecture and Access</td>
<td>7</td>
</tr>
<tr>
<td>2.5 Memory Access</td>
<td>7</td>
</tr>
<tr>
<td>3 BROOK+</td>
<td>9</td>
</tr>
<tr>
<td>3.1 Evolution of Brook+</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Brook+ Development</td>
<td>10</td>
</tr>
<tr>
<td>3.3 Brook+ Language Elements</td>
<td>11</td>
</tr>
<tr>
<td>3.3.1 Brcc</td>
<td>11</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

PAGE

Figure 2.1. Generic diagram of stream processor ................................................................. 3

Figure 2.2. Interleaving threads to hide latency due to memory access ......................... 5

Figure 2.3. Simplified ATI Stream Computing Programming Model ............................... 6

Figure 3.1. ATI Stream SDK software stack .................................................................... 10

Figure 3.2. Brook+ language elements .............................................................................. 11

Figure 4.1. Set-up of the 3D stream .................................................................................. 15

Figure 4.2. Diagram that shows how each possible result would be written to the same location in the output stream ........................................................................ 16

Figure 5.1. Stream compaction steps using prefix sum ..................................................... 25

Figure A.1. Select “Empty Project” and give the project an appropriate name ............... 33

Figure A.2. Rename the Resource files in the Solution Explorer to Brook Files ............... 34

Figure A.3. Inside the project property window select “Configuration Properties” and then select “General” to set the output and intermediate directories. ....................... 34

Figure A.4. Expand the “Linker” configuration, select “General,” and set the “Additional Library Directories” .................................................................................. 35

Figure A.5. Under “Linker,” select “Input” and set the “Additional Dependencies” .......... 35

Figure A.6. Expand “C/C++” then select “General” and set the “Additional Include Directories” ........................................................................................................... 36

Figure A.7. Select “Custom Build Step” and set the command line ................................. 37
# LIST OF TERMS

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ATI Stream SDK:</strong></td>
<td>A complete software development suite from ATI for developing applications for ATI stream processors. (is also called ATI Stream Computing SDK).</td>
</tr>
<tr>
<td><strong>CAL:</strong></td>
<td>Compute Abstraction Layer. A device driver library that provides a forward-compatible interface to ATI Stream processor devices.</td>
</tr>
<tr>
<td><strong>Host Code:</strong></td>
<td>It is the code that is executed on the CPU.</td>
</tr>
<tr>
<td><strong>Kernel:</strong></td>
<td>It is the code that is executed on the device.</td>
</tr>
<tr>
<td><strong>SIMD:</strong></td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td><strong>SIMD Engine:</strong></td>
<td>A collection of thread processors, each of which executes the same instruction per cycle</td>
</tr>
<tr>
<td><strong>Stream:</strong></td>
<td>A stream is a collection of data elements of same type that can be operated on in parallel.</td>
</tr>
<tr>
<td><strong>Stream computing:</strong></td>
<td>Stream computing (or stream processing) refers to a class of compute problems, applications or tasks that can be broken down into parallel, identical operations and run simultaneously on a single processor device</td>
</tr>
<tr>
<td><strong>Stream processor:</strong></td>
<td>A parallel processor capable of executing multiple threads of a kernel in order to process stream of data.</td>
</tr>
<tr>
<td><strong>Thread processor:</strong></td>
<td>The hardware units in a SIMD engine responsible to execute threads of a kernel.</td>
</tr>
<tr>
<td><strong>Wavefront:</strong></td>
<td>Groups of threads executed in a single SIMD engine. A full wavefront contains 64 threads. A wavefront with less than 64 threads is called a partial wavefront.</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Beal’s Conjecture is an open question in number theory which has attracted interest from many researchers who have attempted to find a counterexample to it:

Equation 1.1 states Beal’s Conjecture:

\[ A^x + B^y = C^z \quad \text{where } A, B, C \text{ and } X, Y, Z \text{ are integers such that } A, B, C > 0 \text{ and } X, Y, Z > 2, \text{ then } A, B \text{ and } C \text{ must have a common factor.} \] (1.1)

The search for a counterexample to Beal’s Conjecture is an ideal problem for SIMD parallel processing using GPGPUs (General Purpose Graphics Processing Units).

1.1 PURPOSE

The purpose of this project is to write a parallel program in the Brook+ programming language to search for counterexamples to Beal’s Conjecture using ATI Stream Computing on the AMD FireStream 9170 GPGPU.

1.2 HISTORY

Many researchers have made attempts to find counter examples to Beal’s Conjecture [1].

Peter Norvig [2] wrote a Python program to search for counterexamples using a COS desktop PC; the source code is available at [3].

Based on the results of various other searches for counterexamples, it is now widely accepted that there no counterexamples exist in which the values of all of the conjecture’s parameters (A, B, C, X, Y, Z) are less than 1000. [4]

For the more mathematically inclined reader, [5] and [6] present interesting discussions of analytical mathematical attempts to disprove the conjecture.
As of today, no one has claimed the $100,000 cash prize offered by Andrew Beal to whomever finds a proof or disproof to his conjecture [7].

1.3 Why Use ATI Stream Computing for Computing Beal’s Conjecture Counter Examples

The search for counter examples to Beal’s Conjecture involves evaluating the truth of the underlying equation for all valid choices of values for the problem parameters (A,B,C,X,Y,Z). Since each of these evaluations is independent of all the others, this task is “data parallel,” and so is an ideal application for stream computing, which can use an SIMD (Single Instruction Multiple Data) processing model and the parallel architecture of the GPU to perform tens to hundreds of independent evaluations in parallel during each clock cycle.

1.4 Stream Computing

Stream computing (or stream processing) refers to a class of compute problems, applications or tasks that can be broken down into parallel, identical operations and run simultaneously on a single device.

Today, stream computing is primarily the realm of the graphics processor unit (GPU) where the parallel hardware and software originally designed to produce graphics and imagery are used instead to perform arithmetic calculations. Applications best suited to stream computing possess two fundamental characteristics:

1. A relatively large amount of arithmetic processing is performed per system memory fetch.

2. Little or no interdependence exists between the arithmetic processing performed on different processing units.
CHAPTER 2
THE STRUCTURE OF THE DEVICE

This chapter explains the hardware functionality of a stream processor.

2.1 The Stream Processor

Figure 2.1 shows a generic diagram of ATI Stream Processor [8]. An ATI stream processor consists of a number of Single Instruction Multiple Data (SIMD) engines.

Different stream processors in general may have different number of SIMD engines. Each SIMD engine consists of a large number of thread processors, each of which in turn contains multiple stream cores. These stream cores are the fundamental programmable computational unit responsible for executing integer, single precision floating point, and double precision floating point arithmetic, including transcendental function evaluation. All of the threads running within the same SIMD engine execute the same instruction sequence, but the common instruction sequence for different SIMD engines in general may be different.

As shown in the Figure 2.1 a generic thread processor is organized as a five-way Very Long Instruction Word (VLIW) processor. A VLIW is a collection of at most 5 scalar instructions which can be issued to a thread processor such that each instruction is executed on one of five stream cores. As shown in Figure 2.1, one of the five stream cores is a T-stream core, i.e. a stream core for transcendental operations (e.g., sine, cosine and logarithm) A thread processor also has one branch execution unit to handle branch instructions. The thread processor combines the multiple stream cores (excluding the transcendental core) to perform double precision floating point scalar operations.

### 2.2 Thread Processing

Within an SIMD engine all thread processors execute the same instruction in a given cycle. Multiple threads are interleaved to hide the latencies due memory accesses and stream core operations. What latency is being referred to here? Latency is the time delay incurred every time data is read from or written to memory. So, how is this latency hidden? A thread is suspended when it is accessing memory and during this time other active threads are executing instructions. Figure 2.2 shows this behavior. There are four threads T0 - T3 that are ready to be executed before cycle 0.

Thread T0 is executed until cycle 20 and then is suspended due to a memory fetch operation. During this time, the scheduler begins thread T1 for execution and T1 executes until cycle 40 and then is suspended. Thread T2 then begins execution and this process continues until all available threads execute. Then the scheduler returns to thread T0. If
the data for which thread T0 was waiting is available, T0 continues execution. Thus, the processor is never kept idle due to memory operations and this helps stream processor in achieving maximum performance.

The collection of threads that are executed concurrently on a single SIMD engine is called a wavefront. The wavefront size of different stream processors in general may be different. The wavefront size of the AMD FireStream 9170 GPGPU is 64 threads; i.e. at most 64 threads can be executed on a single SIMD engine concurrently. [9]

The number of threads created for a kernel call is called the domain of execution. The number of threads is determined by the size of the output for a kernel call. Figure 2.3 shows a domain of execution where the output buffer is two dimensional. There is a limit on the domain of execution due to the limit on the size of output buffer. Hence multiple kernel calls may be required to completely execute an application. As shown in Figure 2.3 the “thread scheduler” schedules the threads among the various thread processors until all the threads for a kernel call are processed.
2.3 Flow Control

Flow control, such as branching, is done as follows: If a thread within a wavefront diverges all paths are executed serially such that the total time to execute a branch is the sum of each path time. Example if two branches A and B take same amount of time \( t \) to execute over the wavefront then the total time taken is \( 2t \) (if any thread diverges). If only one thread in the wavefront diverges, the rest of the threads in the wavefront execute the path.

For example there is a branch like this:

```c
If(x)
{
    Path A
}
Else
{
    Path B
}
```

Say there 63 threads within the wavefront that take path A and only 1 thread takes path B, then the entire wavefront has to first execute path A including the thread that was suppose to execute path B. Then all the 64 threads execute path B. When the threads are
executing path A the thread that was not suppose to execute path A would not write the result to memory while executing path A. Similarly the 63 threads that should take path A would execute path B but would not write results to memory while executing path B. Loops execute in a similar fashion where the wavefront occupies an SIMD engine as long as there is at least one thread from the wavefront still being processed. Thus the total execution time taken by the wavefront is determined by the thread with the longest execution time. Example: If \( t \) is the time it takes to execute a single iteration of a loop, and if all the threads in the wavefront execute the loop once except a single thread which executes the loop 100 times then the time taken to execute the entire wavefront is \( 100t \).

2.4 MEMORY ARCHITECTURE AND ACCESS

There are three memory domains for the execution of stream processor applications:

- **Host Memory**: It is the memory on the host computer and is not accessible by the GPU. It is the memory for host code and data.

- **PCIe Memory**: This is a section memory on the host computer set aside for PCIe use. This memory can be accessed by both the host as well as the stream processor but accessing this memory requires synchronization between CPU and the stream processor. Brook+ makes this transparent.

- **Local Memory**: This memory is on the device and can be accessed by the stream processor. The CPU cannot access this memory.

Memory transfers can be explicitly handled by using a low level API. The programs that do not handle memory transfers explicitly will have two copy processes - one from host memory to PCIe and the other from PCIe to stream processor. This double copying lowers the overall system memory bandwidth. Copies between host and PCIe are in the range of hundreds of MBps and that between PCIe and stream processor memory are in the range of GBps. On chip memory bandwidth is about tens to hundred GBps.

2.5 MEMORY ACCESS

Accessing stream processor memory is typically faster than accessing remote (system or host) memory. In a stream processor, stream cores cannot access local device memory directly. In fact there is specific hardware for accessing memory i.e. the memory controller
(see Figure 2.2, p. 5). When a thread needs to access memory it is suspended and a request to the memory controller is issued. The thread is not active while the memory controller gets the access to the memory. But other threads in the SIMD engine can still be executing and thus contributing to better performance.
CHAPTER 3

BROOK+

Brook+ is the programming language used in this project.

3.1 EVOLUTION OF BROOK+

Brook is an extension of ANSI C designed to incorporate data parallelism and high density arithmetic processing into the familiar C language. Brook was developed by Stanford University specifically for implementing stream processing.

BrookGPU is an implementation of the Brook language aimed specifically at enabling stream processing on modern graphics processors [10]. Like Brook, BrookGPU was also developed at Stanford University.

Brook+ is an implementation by Advanced Micro Devices Corporation (AMD) of BrookGPU’s spec on AMD’s “Compute Abstraction Layer” (CAL) plus some additional enhancements. Brook+ has retained those features of BrookGPU that are relevant to modern graphics hardware. The kernels are compiled to AMD’s Intermediate Language (IL). CAL is used for the GPU backend and a CPU backend is also included.

The description of Brook+ depends on two fundamental concepts: streams and kernels. A stream is a collection of similar data elements that can be operated in parallel. A kernel is a function that operates concurrently on every element of a stream[11].

The ATI stream computing programming model provides a complete suite of tools for developers. ATI’s stream computing SDK includes the following components (Figure 3.1):

2. Device driver for stream processors: ATI Compute Abstract Library (CAL)
3. Performance Profiling Tools: Stream Kernel Analyzer
4. Brook+: High level Language, C with extension for GPU.
5. Performance Libraries: AMD Core Math Library (ACML) for optimized domain-specific algorithms.

### 3.2 Brook+ Development

Brook+ SDK can be downloaded from the AMD website; the installation procedure is described in the appendix. The SDK comes with the Brook+ compiler and Brook+ runtime libraries. The SDK also comes with sample projects which are very useful for novice developers. The stream programming user guide [8] that comes along with the SDK provides all the necessary information to get started with Brook+ development. The guide also provides the information on using Visual Studio for Brook+ development. Building an application consists of:

1. Using the Brook+ compiler to compile the Brook+ source code into a C++ file. This file contains code for the CPU and stream processors.

2. Compiling the C++ file with the rest of the application and link it with Brook+ runtime libraries.

A Brook+ source file is normally given a .br extension. Brook+ code is very similar to C/C++. A Brook+ program consist of host code and kernel code. Host code is executed on
the CPU and a kernel is the functions that run on stream processors. Kernels are executed by calling them, just as in C with actual parameters. For more complex applications host code can be written in C and kernel function written in Brook+ can be called from C/C++ source files.

### 3.3 Brook+ Language Elements

Figure 3.2 illustrates the Brook+ language elements.

**Figure 3.2. Brook+ language elements.**

#### 3.3.1 Brcc

Brcc is a source to source meta compiler that translates brook+ programs (.br files) into device dependent kernels embedded in valid C++ source code. The generated C++ code includes CPU and stream processor codes which are later linked to executables.
3.3.2 Brt

Brt is a runtime library that executes the kernel invoked from CPU code in the application. Brook+ includes various runtimes for CPUs and stream processors; you can select the execution model at application runtime. The CPU runtime serves as a good debugging tool when developing stream kernels.

A Simple brook+ program is shown below

```c
kernel sum (float a<>, float b<>, out float c<>)
{
    c = a + b;
}

int main()
{
    ....
    float a<10,10>;                     // Creating a stream
    ....
    streamRead(a,input_a);          //copies data from memory to stream
    sum(a,b,c);                                // kernel call
    streamWrite(c,output_c);     // copies data from stream to memory.
}
```

In Brook+ kernel are syntaxtically functions() which has one or more formal parameter which are executed by calling them just as in C with actual parameters.

In the example above a and b are input streams and c is an output stream. In the statement shown above, ‘c = a + b’ in the code is interpreted as \( c[i] = a[i] + b[i] \) (considering a ,b ,c are single dimensional streams). This addressing of the streams is done automatically and is handled by the API.

Streams are denoted using angle brackets. Streams cannot be accessed directly by the application. Data must be copied between streams and memory using streamRead() and streamWrite().
CHAPTER 4

IMPLEMENTATION

4.1 BEAL’S CONJECTURE

Recall that Beal’s Conjecture states: If $A^x + B^y = C^z$, where $A$, $B$, $C$, $x$, $y$ and $z$ are positive integers and $x$, $y$ and $z$ are all greater than 2, then $A$, $B$ and $C$ must have a common factor.

Therefore, Equation 4.1 shows a counter example to Beal conjecture would be a six-tuple $(A,B,C,x,y,z)$ such that:

\[ A^x + B^y = C^z, \text{ where } A, B, C, x, y \text{ and } z \text{ are positive integers and } x, y \text{ and } z \text{ are all greater than 2, and } A, B \text{ and } C \text{ are pairwise relatively prime.} \] (4.1)

This chapter deals with the methods that were used to implement the algorithm used to search for counterexamples, and explains the reasons for various design choices in the implementation.

4.2 METHODS

Beal’s Conjecture has six parameters/variables, each of which may be an extremely large integer; therefore, when searching for counterexamples, we must be prepared to deal with both a large number of candidate counterexample six-tuples, and also with the fact that the integers comprising those six-tuples may be extremely large. Any suitable algorithm would be one that would efficiently address both of the above considerations. The first implementation attempted was a sequential Java program with six nested loops for each of the variables. This program was written in order to understand the process and would be the base program to be parallelized. The problem of large integers can be avoided in Java since we can exploit the Java “BigInteger” class that supports arbitrary precision integer representation and arithmetic.
When implementing on the GPGPU device using Brook+, there is no direct support for large integers comparable to that available in the Java BigInteger class. The first approach (Equation 4.2) is to use a logarithm and solve for “z.”

\[ A^x + B^y = C^z \]

Now, applying logarithm on both sides, we get

\[ \log(A^x + B^y) = z \cdot \log(C), \]
\[ z = \frac{\log(A^x + B^y)}{\log(C)}. \]  \hspace{1cm} (4.2)

If z very nearly an integer (say, differing from an integer by less than \(10^{-6}\)) then the six-tuple \((A, B, C, x, y, z)\) would be considered a possible candidate for a counterexample.

The second method was to use modular arithmetic. Any six-tuple that satisfies the original equation also satisfies Equation 4.3:

\[ (A^x \mod N + B^y \mod N) \mod N = C^z \mod N \]  \hspace{1cm} (4.3)

for any integer N. Therefore, six-tuples that fail to satisfy the above equation for any value of N are definitely NOT counterexamples to Beal’s Conjecture, and six-tuples that DO satisfy the above equation for one (or, preferably, many) different (preferably, relatively prime) values of N are good candidates for further testing using exact arbitrary precision integer arithmetic. Note that to do exact integer arithmetic modulo a prime \(p\), numbers as large as \(p^3\) must fit in a machine word [4].

The second method was chosen for this project.

To implement a program on the device, we use a 3D stream and launch the kernel. Since Brook+ launches threads for each data element of this 3D stream and executes the operation in parallel, we can use the index of the 3D stream to serve for the values of variables \(A, B, \) and \(C\). Figure 4.1 shows this setup. For the values of \(x, y, \) and \(z\), we run the nested loops.

The maximum stream size in Brook+ is \(8192 \times 8192\). Therefore, the search has to be performed in blocks if the range of numbers for which it is being tested is greater than
8192 × 8192. The host code performs multiple kernel calls so that the search is performed over the entire range. The current implementation chooses a block size of 8192 × 90 × 90 (due to the limitation on the stream size). This block size launches the maximum number of threads that can be launched considering the restriction on the stream size. The idea is to launch as many threads as possible so that one can take advantage of the architecture of the GPU.

A distinctive aspect of this implementation is the way the solution is returned to the kernel. The only way this can be done is to have an output stream in the kernel that can store the results. This output stream needs to be copied onto the host memory. The host then needs to filter out the results from a very large set (array). Since the kernel call is asynchronous, the device can be made to search the next set of values while the host filters the results returned from the previous stream (at the same time). The best solution would be to have the kernel return an array holding as many values as the number of results so that the host code would not need to filter the results.

Since we have loops for the values of x,y,z, next we have to decide where these loops can be placed in the source code. There are two possible places where the loops for x,y,z can be kept: (1) in the kernel, and (2) in the host code.
4.2.1 Loops for x,y,z in Kernel Code

We have threads for each combination of A, B, and C and each thread, in turn, has loops for x, y, and z. Having loops in kernel has the advantage that we have each thread executing the loops in parallel and have the kernel doing more work rather than the host. Now since each thread which is handling a single combination of A,B,C, would execute the loops for x,y,z , a thread may have more than one result to store. Each possible result would then be written to the same location in the output stream. (Figure 4.2). This way, we would miss out on some of the results. This would not give accurate results.

![Diagram showing how each possible result would be written to the same location in the output stream.](image)

Here is the kernel code when the loops for x,y,z is in the kernel:

```c
kernel void findCounterExample(int startRangeA, int startRangeB, int startRangeC, out int3 a<>) {
    ...
    if(gcdAB==(unsigned int)1 && gcdAC==(unsigned int)1 && gcdBC==(unsigned int)1) {
        for(X=3;X<10;X++)
            for(Y=X;Y<10;Y++)
                for(Z=3;Z<10;Z++)
```

Figure 4.2. Diagram that shows how each possible result would be written to the same location in the output stream.
Here the output stream as int3 so as to store values of x,y,z in the output stream and the values of A,B,C can be calculated from the index of the output stream. The kernel launches for a thread for each element in the output stream and there are nested loops for x,y,z as shown in the kernel cod above.

### 4.2.2 Loops for x,y,z in Host Code

To solve the problem of missing out on results, what is needed is a kernel with each thread producing only a single result. The only way to do this is to have a loop for x, y, and z in the host and launch the kernel with just one value of x, y, and z so that each thread can output only one result.

Launching kernel for each combination of X, Y ,Z, the host code is as follows:

```c
for( x= startX ; x < endX;  x++)
{
    for( y= startY ; y < endY; y++)
    {
        for( z= startZ ; Z < endZ;  z++)
        {
            call kernel with the 3D stream for values of A ,B ,C and the current value of x, y, z
        }
    }
}
```
Since the kernel is launched with a single combination of x,y,z each combination of A,B,C will be tested with the same value of x,y,z. Each thread can only produce a single result therefore avoiding the overwrite problem.

Although we get accurate results with this approach, there is some performance penalty for having the loops in the host code. Firstly, since the kernel is launched for each combination of x,y,z there would be a lot for kernel calls than there were by having the loops in the host code. There is always an overhead in launching the kernel and by launching the kernel more number of times this overhead increases. Secondly, by having the loops in the host code, the host is doing more processing. Also for each kernel call the results need to be filtered from the output stream on the host code. The more the number of kernel calls, the more number of the times the host need to the filter the results. This would result in performance degradation.

4.3 **Annotated Source Code**

Kernel Code
```
#include<stdio.h>

/*
Equation to solve is A^X + B^Y = C^Z
*/

//Simple non recursive function to find gcd of 2 numbers
//(Helper function)
//This function takes two number as input and returns the gcd of this numbers.
kernel unsigned int findGcd(unsigned int u,unsigned int v)
{
    unsigned int gcd = (unsigned int)1;
    unsigned int r;
    unsigned int num1=u;
    unsigned int num2 =v;
    while (1)
    {
        if (num2 == (unsigned int)0)
        {
            gcd = num1;
            break;
        }
        else
        {
```
\[ r = (num1 \% num2); \]
\[ num1 = num2; \]
\[ num2 = r; \]
\} 
\}

return gcd;
}

// Function to calculate modular exponentiation. Algorithm taken from Wikipedia
//(Helper function)
// This function takes the number and the exponent as the input and returns
// the modpower as the return

kernel unsigned int modulusPower(unsigned int number, unsigned int exponent) {
    //biggest prime number less than 2^16
    //N is taken as less than 2^16 even though unsigned int can store
    //2^32 as max integer
    //this is because if N is greater it gives wrong result for the mod.
    unsigned int N = (unsigned int)65521;
    unsigned int base = number;
    unsigned int counter = exponent;
    unsigned int result = 1;

    while (counter > 0) {
        if (counter & 1) {
            result = ((result * base) % N);
        }
        counter = counter >> 1;
        base = ((base * base) % N);
    }
    return result;
}

kernel void findCounterExample(int startRangeA, int startRangeB, int startRangeC, int X, int Y, int Z, out int a>) {
    int A, B, C;
    unsigned int gcdAB, gcdAC, gcdBC;
    unsigned int N = (unsigned int)65521;

    // using the index of the output stream as the values for A, B, C
    // There is a thread for element in the stream.
    A = instance().x + startRangeA;
    B = instance().y + startRangeB;
    C = instance().z + startRangeC;

    // initialising a to 0 so that when we filter the results we can know
    // that 0 means that
//location does not have a result.
a=0;

gcdAB = findGcd((unsigned int)A,(unsigned int)B);
gcdAC = findGcd((unsigned int)A,(unsigned int)C);
gcdBC = findGcd((unsigned int)B,(unsigned int)C);

//if A,B,C are co primes test for the equation
if(gcdAB==(unsigned int)1 && gcdAC==(unsigned int)1 &&
gcdBC==(unsigned int)1)
{
    //(A^X + B^X)Mod N .
    unsigned int sum = modulusPower((unsigned int)A,(unsigned int)X) +
     modulusPower((unsigned int)B,(unsigned int)Y);

    //C^Z Mod N
    unsigned int cpowerZ = modulusPower((unsigned int)C,
     (unsigned int)Z);

    sum = (sum%N);
    if(cpowerZ == sum)
    {
        //make the stream equal to 1 so now to indicate that
        //postion i.e that value of A,B,C is a possiblle counterexample
        //on the host code would filter the stream to check if it has
        //value one
        // the index would be the values of A,B,C
        a= 1;
    }
}

Host Code
#include “brookgenfiles\beals.h”
#include “conio.h”
#include “brook\stream.h”
#include <time.h>
using namespace brook;

//function that filters the results from the stream and write it to the file.
void writeResultsToFile(int *counterExample,unsigned int dimension[],int
startRange[],int X,int Y,int Z)
{
    int i,j,k;
    for(i=0;i<dimension[2];i++)
        for(j=0;j<dimension[1];j++)
            for(k=0;k<dimension[0];k++)
            {
//Stream is 3-Dimensional and is read to a continous chunk of data
//Mapping the 3Dimension to a 1 dimension.
                if(counterExample[i*dimension[1]*dimension[0]+j*dimension[0]+
                    k]!=0)
//write to a file
//    A^X         B^Y         C^Z
//"(k+startRange[0]^X)+(j+startRange[1]^Y) =
//(i+startRange[2]^Z)"
//this string to be written to a file
//Getting the values of A,B,C from the index exactly the
//same way as they were formed.
printf("%d^%d + %d^%d = %d^%d \n",
    (k+startRange[0]), X,
    (j+startRange[1]), Y,
    (i+startRange[2]),Z);

}
}

int main(int argc, char ** argv)
{
    int iA,jB,kC,X,Y,Z,range[3],Xp,Yp,Zp;
    int startRange =1029;
    int endRange = 1039;
    int *counterExample;
    int exponentRange =10;
    time_t start, end;
    unsigned int dimension[3] = {0};
    unsigned int dim[] = {10,10,10};

    printf("X , Y ,Z loop in the host code :\n");
    start = time(NULL);

    //Since the maximum size of stream can be 8192 x 8192.
    //Launching kernel with stream size 8192 x 90 x 90 so that max number
    //of threads can be launched.
    for(iA=0;iA<(endRange - startRange);)
    {
        //The max size for first dimension is 8192
        //If the range for A is less than 8192 then reduce dimension
        if((endRange - startRange-iA)<8192)
            dim[0] = endRange - startRange-iA;
        else
            dim[0] = 8192;
        for(jB=0;jB<(endRange - startRange);)
        {
            //the next 2 dimension can be max 90 each
            if((endRange - startRange-jB)<90)
                dim[1] = endRange - startRange-jB;
            else
                dim[1] = 90;
            for(kC=0;kC<(endRange - startRange);)
            {
                //The max size for first dimension is 8192
if((endRange - startRange-kC)<90)
    dim[2] = endRange - startRange-kC;
dim[2] = 90;
//the next loops are for X,Y,Z
//the reason for not having the loop in the kernel
//is then each thread could possible generate more
//than one result.
//There is no other way to return the result then
//through streams
for( X = 3; X < exponentRange; X++)
{
    for( Y = X; Y < exponentRange; Y++)
    {
        for( Z = 3; Z < exponentRange; Z++)
        {
            //Creating a stream with the
            //calculated dimensions
            Stream<int> aStream(3,dim);
            /*
            Constructor to create a stream
            Stream :: Stream (unsigned short rank
            ,unsigned int * dimension)
            where
            rank is the number of dimension in the
            stream.
            dimension is upper bound for each
dimension
            */
            //Calling the kernel with the stream
            findCounterExample(startRange+iA,
                startRange+jB,
                startRange+kC,
                X,Y,Z,aStream);
            //Every pass writes the result of the
            //previous
            //this check is to see if its the first
            //pass
            //Not checking for aStream.isSync()
            //since in either case this step needs
            //to be done
            //By default it will be done in
            //parallel as the control return to
            //host code after
            //kernel call. Kernel call are
            //asynchronous.
            if(iA!=0||kC!=0||Z!=3){
                writeResultsToFile(counterExample,
                    dimension,range,
                    Xp,Yp,Zp);
                free(counterExample);
            }
        }
    }
}
counterExample = (int *)
malloc(dim[0]*dim[1]*dim[2]*sizeof(int));

//copies data from stream to memory
aStream.write(counterExample);
/*
void Stream::write(void *ptr
 , const char
*flags = NULL)
This copies the data from memory
associated with the stream to a
host-side pointer.
It is a synchronous call and blocks any
return to the caller until all data
has been written to the host.
There is no check at runtime if the
*ptr points to sufficiently large
memory.
It is left to the user.
The flags parameter controls the
behaviour of stream write when
requesting asynchronous
operation and memory pinning.
*/

// since the result of previous kernel
call are filtered // need to store the
dimension of the stream with // which
the previous kernel was called.
dimension[0] = dim[0];
dimension[1] = dim[1];
dimension[2] = dim[2];
// Stores the start range for A,B,C for
// each kernel call
range[0] = startRange+iA;
range[1] = startRange+jB;
range[2] = startRange+kC;
// Stores the values of X,Y,Z used for
// previous kernel call
Xp=X;
Yp=Y;
Zp=Z;
}

}
kC+=90;
}
jB+=90;
}
iA+=8192;
}
// Filtering the results from the stream for the last kernel call.
writeResultsToFile(counterExample,dimension,range,Xp,Yp,Zp);
/Getting the end time
end = time(NULL);
printf("The time taken is : %.2f sec's\n", difftime(end, start));
getch();
return 0;
CHAPTER 5

CONCLUSION

5.1 FUTURE ENHANCEMENT: STREAM COMPACTION

Stream compaction is a general technique for reordering a disorganized stream of elements into compact sub streams of like elements [12]. Stream compaction creates an ordered vector from a disorganized input by computing a new index for each element’s location in a compacted output vector. It is implemented with the operations of prefix sum and scatter.

Main steps of performing compaction with a prefix sum are shown in Figure 5.1.

Prefix sum gives the location of the valid input flags in the output vector and the scatter step is used to move the valid input elements into the output vector.

How can stream compaction help optimize the implementation described in the previous chapter?

Now that we have an idea of stream compaction, let’s look at how we can use stream compaction for optimizing the implementation used in this project.
In the implementation used in this project the kernel is called with the output stream and it does not have any input stream. The kernel then calculates the values of A, B, C implicitly from the index of the output stream. If the kernel finds a candidate for a particular combination of A, B, C it stores a value of 1 in the index obtained from the thread IDs so that the host can recalculate the value of A, B, C from the index in the output stream while it is scanning the stream for results. The only way to return the results from the kernel to the host code in Brook+ is by writing the results in the output stream. The only place where the results for a particular combination of A, B, C can be written is the corresponding index location in the output stream.

Why is it that we can only write results to the corresponding location in the output stream? For example, if values of A, B, C = 1001, 1002, 1003, these values are generated for index 1, 2, 3 of the output stream and if this values of A, B, C along with the value of x, y, z is a possible counter example then we write ‘1’ in output stream at location 1, 2, 3. If this is written at some other location like 2, 3, 4 in the output stream then firstly, we might not be able to calculate the corresponding value of A, B, C and secondly, the value of A, B, C corresponding to index 2, 3, 4 might also be candidate and the thread handling that particular combination may have executed and already written the result to that location. This means the result can be overwritten.

As we know the size of the output stream used in the implementation is very large (8192 x 8192) with the results scattered in the stream, each kernel call would require the host code to filter a stream as large as 8192 x 8192 elements. On the host code filtering is done by checking every element in the stream to see if it holds the result (value 1 represents it is a result). So we will be scanning the entire stream even though we know there might be a very few results in the stream. This filtering on host code for a large stream for each kernel call can reduce the overall performance of the search. So the idea is to implement stream compaction using prefix sum in parallel on the kernel instead of the host code. The prefix sum and scatter/gather step of stream compaction can be parallelized and executed
on the kernel. The complexity of an efficient prefix sum algorithm is O(n) [13]. With stream compaction, the entire stream need not be scanned for results. The last entry in the prefix sum would let us know the number of results present in the stream and since stream compaction would ensure that the results are arranged at the start of the stream we would then get only the results on the host. As shown in Figure 5.1 the last entry of the prefix sum output tells us that there only 8 results and the scatter/gather steps makes sure all the 8 results are organized at the start of the stream.

The filtering of result from the output stream is the major bottleneck and implementing stream compaction would improve the overall performance.

5.2 Future Enhancement: Multi-GPU Support

More systems are configured with two or more GPUs. Brook+ lets the developer use a single brook+ program to take advantage of all compatible GPUs in a system. There are routines provided that let the developers query what devices are available on the system and select which devices streams are allocated on and kernels are invoked on; so we can have one GPU look for the possible counter examples for Beal’s conjecture while the other GPU can perform stream compaction in parallel to take advantage of the GPUs available and further improve the performance.

This project uses the brute force technique to find a counter example to Beal Conjecture. The performance of this project could be improved if we could generate co primes triplet A, B, C and test these values for a possible counter example. This implementation uses ATI stream computing SDK 1.4.0 and could be implemented on the new SDK available now to add more flexibility to the algorithm and take advantage of the new features included.

5.3 Conclusion

An algorithm for finding counter examples to Beal’s Conjecture was implemented in Brook+ using ATI SDK 1.4.0. The project was executed with CPU as the backend and was
tested for functionality. It was tested for a small range of values and is scalable for larger ranges. To verify the results of the project a serial code was written. This project can be executed on a GPU with no changes required in the source code. To run the code on the GPU, the environment variable “BRT_RUNTIME” should be set to “cal”.
REFERENCES


APPENDIX

BROOK+ SDK INSTALLATION AND BROOK+
PROJECT INSTRUCTIONS
APPENDIX

This document gives the steps to install the ATI Brook+ SDK 1.4.0 and how to make a Brook+ project in Visual Studio 8.

The Prerequisites for Installing Brook+ SDK

- Administrative privileges on the system it is being installed on.
- One of the following operating system.
  - Windows XP 32 bit /64 bit (Service Pack 1)
  - Windows Vista 32 bit /64 bit(Service Pack 1)
  - Linux (32 bit or 64 bit) : Red Hat Enterprise Linux 5.1,SUSE Linux Enterprise Server SLEP 10 with service pack 1
- A graphics card supported by SDK 1.4.0.
- Visual studio 2008 or 2005 – All the samples provided with the SDK are in VS8 format along with - makefiles.
- ATI Catalyst driver.

  The SDK for appropriate operating system and the catalyst driver can be downloaded from the website http://www.amd.com/stream.

Installing Brook+ on Windows System

- Install Brook+ using the installer provided in the SDK and follow the prompts. The installer adds an environment variable BROOKROOT and sets it to the root directory of the installation.
  - The default value of this variable is BROOKROOT = C:\Program Files\ATI\<Brook+ sdk>
- Make sure sdk/lib directory under ATI Brook+ installation is included in PATH.
- Install ATI CAL using the installer provided in the sdk and follow the prompts. The installer adds an environment variable CALROOT and sets it to the root directory of the installation.
  - The default value of this variable is BROOKROOT = C:\Program Files\ATI\<CAL+ sdk>.
Installing Brook+ on Linux System

- With root permissions:
  - Invoke atistream-brook-1.4.0_beta.i386.run for 32 bit systems,
  - Invoke atistream-brook-1.4.0_beta.x86_64.run for 64 bit systems

After Installing, Set the Environment Variables

- CALROOT to the directory where CAL is installed default being /usr/local/atical
- Also add the lib folder of the CAL installation to LD_LIBRARY_PATH also add the lib folder of the BROOK+ installation so that applications can find run time libraries of Brook+.
- Add the sdk/bin folder to the PATH so as to have access to brcc in the build scripts. The default value is : /usr/local/atibrook/sdk/bin
- Set the BROOKROOT to Brook+ installation directory. The default is : /usr/local/atibrook/

After installation if one needs to execute the code on CPU backend for debugging or to test the functionality one needs to create the environment variable BRT_RUNTIME and set it to cpu and when one wants GPU backend, the value is set to cal.

To test that the installation was correct we can open and check if the samples provided in the sdk compiles and are executed. The samples in the sdk are visual studio projects which can be easily built to verify if the installation was correctly done. For Linux systems the samples have makefiles.
Creating a Brook+ Project in Visual Studio 2008

- Open Visual studio and click on "New Project: on the “File” menu.
- On the “New Project” window select project type as “General” and “Template” as “Empty Project”.
- Give the project appropriate name (see Figure A.1).

As shown in Figure A.2, rename the Resource files in the Solution Explorer to Brook Files (for better organization of files).

Go to the project properties from the Project menu. Inside the project property window select “Configuration Properties” and then select “General” and set the following fields, as shown in Figure A.3:

- Output Directory -> $(ProjectDir)
- Intermediate Directory -> $(ConfigurationName)

Now expand the “Linker” configuration and select “General” and set the following fields, as shown in Figure A.4:

- Additional Library -> "$(BROOKROOT)/sdk/lib"

On the Linker select “Input” and set following fields, as shown in Figure A.5:

- Additional Dependencies - > brook_d.lib

Expand “C/C++” and select “General” and set the following, as shown in Figure A.6:

- Additional Include Directories - > "$(BROOKROOT)/sdk\include\";
Figure A.2. Rename the Resource files in the Solution Explorer to Brook Files.

Figure A.3. Inside the project property window select “Configuration Properties” and then select “General” to set the output and intermediate directories.
Figure A.4. Expand the “Linker” configuration, select “General,” and set the “Additional Library Directories.”

Figure A.5. Under “Linker,” select “Input” and set the “Additional Dependencies.”
Now create a file in the Brook Files folder to write your Brook+ code. This file has the extension .br.

On the properties of this .br file select Custom Build Step and set the command line as shown in Figure A.7:

- mkdir brookgenfiles | "$(BROOKROOT)\sdk\bin\brcc_d.exe" -o "$(ProjectDir)\brookgenfiles\$(InputName)" "$(InputPath)"

- Also outputs as $(ProjectDir)\brookgenfiles\$(InputName).cpp

This completes the configuration of a general project in Brook+ using visual studio. Apart from this if you have a C++ file which has the host code, place that file in the Source Files folder.

Next Build the project and once it is built, execute it and see the results.
Figure A.7. Select “Custom Build Step” and set the command line.