Using Multiple Languages on a GPU Computation Platform

Kelly O'Conor, Advisor: Dr.Don Spickler
Villanova University, Salisbury University

Abstract

As the use of graphic-processor units (GPU) to obtain faster performance improvements becomes more popular, the use of CUDA as a programming model for GPUs for use by C/C++ programmers has increased as well. In this investigation, I present the use of multiple languages on a GPU computation platform exploring extended precision integer arithmetic, specifically Java and C in CUDA. To do so, I used a programming interface called JCUDA that can be used by Java programmers to invoke CUDA kernels. By using this interface, programmers can write Java codes that directly call CUDA kernels and generate the Java-CUDA bridge codes and allows for the host device data transfer to the GPU. As a benchmark application for run times, I used the factoring of large semi-primes using a quadratic sieve method. While still slower than C in CUDA, the preliminary performance results show that this interface delivers a significant performance improvement to Java programmers.

Background Information

GPU Processing and CUDA

A GPU (Graphics Processing Unit) is a highly parallel computing device designed for the task of graphics rendering. Due largely in part to the demand for high definition graphics, 3D gaming, and multimedia experiences, the GPU has evolved into a very parallel, multithreaded, multi-core, more general processor allowing users to program certain aspects of the GPU to create detailed graphics and scientific application. In general, the GPU has become a powerful device for the execution of data-parallel, arithmetic intensive applications in which the same operations are carried out on many elements of data in parallel. Example applications include video processing, machine learning, and 3D medical imaging.

The NVIDIA's Compute Unified Device Architecture (CUDA) has become the popular programming model for GPUs for use by C/C++ programmers. The basic idea behind computing on the GPU is to use to speed up and accelerate specific computations in applications, which traditionally are done by the CPU (Central Processing Unit). While using CUDA, a written application contains two sections of code: functions on the CPU host and functions on the GPU device. The functions for the GPU are labeled with the keyword **global** and are called **kernels**. The kernel, which operates across an array of data, executes across a set of parallel threads in parallel. Triple angle brackets mark a call from host code to device code. CUDA enables dramatic increases in computing performance by harnessing the power of the graphics processing unit.

The Quadratic Sieve Factoring Method

The Quadratic Sieve Factoring method was the benchmark application for runtime comparisons.

The QS consists of two major steps: the *sieving step*, to collect the relations, and the *matrix step*, where the relations are combined and the factorization is derived. For numbers in the current range, the sieving step is by far the most time consuming. It is also the step that allows easy parallelization.

Methods & Results

Rootbeer

Originally, I intended to use Rootbeer in order to use Java on the GPU.

Developed by Philip C. Pratt-Szeliga, a Ph.D candidate at Syracuse University, the Rootbeer GPU Compiler makes it easy to use Graphics Processing Units from within Java. Rootbeer automatically (de)serializes complex graphs of objects into arrays of primitive data and generates the CUDA code through a static analysis of the Java Bytecode.

Unfortunately while the sample codes ran, when I used the Big Integer class needed for factoring instead of the int class in Java, Rootbeer failed.

```
import edu.syr.pcpratts.rootbeer.runtime.Kernel;
import edu.syr.pcpratts.rootbeer.runtime.Rootbeer;
import edu.syr.pcpratts.rootbeer.runtime.util.Stopwatch;

public class BigIntegerFactoring {

    public void BigMultInt(BigInteger[] array){

        List<Kernel> jobs = new ArrayList<Kernel>();
        for(int i = 0; i < array.length; ++i){
            jobs.add(new BigIntegerMult(array, i));
        }

        Rootbeer rootbeer = new Rootbeer();
        rootbeer.runAll(jobs);

    }
```

Rootbeer Sample Code

JCUDA

JCUDA is designed to be an interface for invoking CUDA kernels from Java code. The JCuda driver has the bindings to load and execute a CUDA kernel. Written mainly in Java, the sample reads a CUDA file which is written in C, compiles it to a PTX file. Then using NVCC, loads the PTX file as a module and executes the kernel function then copies the device output back to the host.

```
// Call the kernel function.

int blockSizeX = 256;
int gridSizeX = (int)Math.ceil((double)allexponents.length / blockSizeX);

System.out.println("Launching Kernel...");
long time0 = System.nanoTime();
cuLaunchKernel(function,
        gridSizeX,  1, 1,      // Grid dimension
        blockSizeX, 1, 1,      // Block dimension
        0, null,               // Shared memory size and stream
        kernelParameters, null // Kernel- and extra parameters
        );
long time1 = System.nanoTime();
System.out.println("cuCtxSynchronize()");
cuCtxSynchronize();
GPUTime = time1 - time0;
```

JCuda sample kernel call

```
#include "bigint.h"
extern "C"
__global__ void processBlock(int n, int* exps, int* primes, int* numbers, int* c)
{
        int tid = threadIdx.x + blockIdx.x * blockDim.x;

        if (tid < n)
        {

                        cuBigInt testnumber;
                        cuBigInt BuiltNum(0);
                        cuBigInt tentho(10000);

                        int pos = numbers[1]-1;

                        while (pos >= 0 && exps[tid*numbers[1]+pos] == 0)
                                pos--;

                        for(int k=pos; k >= 0; k--){
                                int part =(exps[tid*numbers[1]+k]);
                                BuiltNum = BuiltNum*(tentho) +(part);

                        }

                testnumber = BuiltNum;

                cuBigInt one(1);
                cuBigInt zero(0);
                int i;

                for (i = tid*numbers[1]; i < tid*numbers[1]+numbers[1]; i++)
                        exps[i] = 0;
```

The C Code it calls

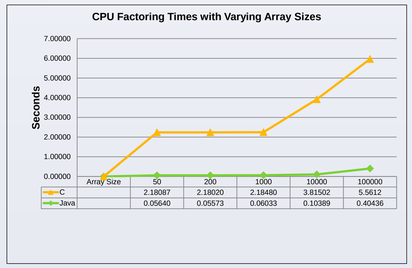
Results

Windows : Windows 7 Enterprise Edition (64-bit) Service Pack 1 (Build 7601)

Memory (RAM): 8184 MB

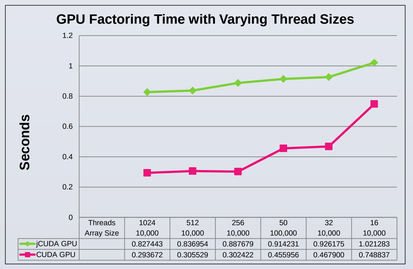
CPU Info: Intel(R) Xeon(R) CPU E5607 @ 2.27GHz

Display Adapters : NVIDIA GeForce GTX 670 / 1344 cores / 2 GB GDDR5 Memory / 7 multi-processors



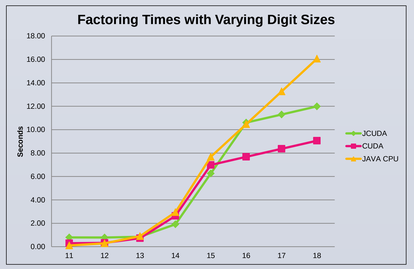
Array Size	50	200	1000	10000	100000
C	2.18087	2.18020	2.18480	3.81502	5.5612
Java	0.05640	0.05573	0.06033	0.10389	0.40436

- Surprising to anyone who has worked with Java and C/C++, the Java-CPU version of the algorithm was faster than the C implementation. This is due to the use of the built in Java Big Integer class which has been optimized to be the fastest versus the Big Int class that was written in C.



Threads / Array Size	1024 / 10.000	512 / 10.000	256 / 10.000	50 / 100.000	32 / 10.000	16 / 10.000
JCUDA GPU	0.827443	0.836954	0.887679	0.914231	0.926175	1.021283
CUDA GPU	0.293672	0.305529	0.302422	0.455956	0.467900	0.748837

- The results of factoring on the GPU were not surprising like the results on the CPU. As the number of threads increased, the speed on the GPU also increased. The JCUDA takes a hit due to the overhead of Java and also the data transfer time, which is naturally must faster on C.



- By changing the amount of digits of the number being factored, the times progressively got slower. JCUDA proved to still be slower than CUDA on the GPU. This is due to the data transfer from the CPU to the GPU in JCUDA. By chopping the Big Integers into an array of ints, I was able to send the big numbers to the GPU, this however was relatively very time consuming.

Pros/ Cons of JCUDA

Pros:

- JCUDA provides access to CUDA for Java programmers, exploiting the full power of GPU hardware from Java based applications. Using JCUDA you can create cross-platform CUDA solutions, that can run on any operating system supported by CUDA without changing your code
- Using Java allows access to the huge library. Specifically, the use of the Big Integer class increased the speed of the algorithm run on the CPU using Java

Cons: The overhead introduced by data transfers can overwhelm the benefits of fast GPU computation which is shown with the results of the GPU factoring time

Conclusion

By the results, I have seen that factoring in CUDA with C is faster than using JCUDA on the GPU. While the surprise came on the CPU side, the GPU results were what was expected. Using JCUDA and Java brings some overhead, especially during data transfer which causes the overall algorithm to be slower. By increasing the array size and the number of threads used on the GPU using both JCUDA and CUDA, I was able to make the algorithm more efficient. Using JCUDA allowed me to use Java and also explore using multiple languages on the GPU.

Advancement

To continue, I could revise the algorithms in the BigInt struct (class) to be more efficient, making the C code on the CPU side faster. Then, I could look into revising the C side of the code to use CUMP (The CUDA Multiple Precision Arithmetic Library). Finally, I can look at alternative ways to relate the Java BigInteger class to the C BigInt struct.

References

- Donald Knuth, Seminumerical Algorithms: the Art of Computer Programming, v.2 (Addison-Wesley Publishing Co.,Reading,MA, 1981.
- Wade Trappe and Lawrence Washington, Introduction to Cryptography with Coding Theory, ed. 2 (Pearson, 2006)
- Neal R. Wagner: The Laws of Cryptography: The RSA Cryptosystem
- Philip C. Pratt-Szeliga: " Rootbeer: Seamlessly using GPUs from Java"
- Connelly Barnes: "Integer Factorization Algorithms"
- CUDA by Example: An Introduction to General-Purpose GPU Programming (29 July 2010) by Jason Sanders, Edward Kandrot
- jcuda.org

Acknowledgements

- NSF REU (Research Experiences for Undergraduates)

EXERCISE - EXplore EmeRging Computing In Science and Engineering program. Award #1156509

- Salisbury University: Henson School of Science & Technology