JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA

Yonghong Yan, Max Grossman, and Vivek Sarkar

Department of Computer Science, Rice University
{yanyh,jmg3,vsarkar}@rice.edu

Abstract. A recent trend in mainstream desktop systems is the use of general-purpose graphics processor units (GPGPUs) to obtain order-of-magnitude performance improvements. CUDA has emerged as a popular programming model for GPGPUs for use by C/C++ programmers. Given the widespread use of modern object-oriented languages with managed runtimes like Java and C#, it is natural to explore how CUDA-like capabilities can be made accessible to those programmers as well. In this paper, we present a programming interface called JCUDA that can be used by Java programmers to invoke CUDA kernels. Using this interface, programmers can write Java codes that directly call CUDA kernels, and delegate the responsibility of generating the Java-CUDA bridge codes and host-device data transfer calls to the compiler. Our preliminary performance results show that this interface can deliver significant performance improvements to Java programmers. For future work, we plan to use the JCUDA interface as a target language for supporting higher level parallel programming languages like X10 and Habanero-Java.

1 Introduction

The computer industry is at a major inflection point in its hardware roadmap due to the end of a decades-long trend of exponentially increasing clock frequencies. It is widely agreed that spatial parallelism in the form of multiple homogeneous and heterogeneous power-efficient cores must be exploited to compensate for this lack of frequency scaling. Unlike previous generations of hardware evolution, this shift towards multicore and manycore computing will have a profound impact on software. These software challenges are further compounded by the need to enable parallelism in workloads and application domains that have traditionally not had to worry about multiprocessor parallelism in the past. Many such applications are written in modern object-oriented languages like Java and C#.

A recent trend in mainstream desktop systems is the use of general-purpose graphics processor units (GPGPUs) to obtain order-of-magnitude performance improvements. As an example, NVIDIA's Compute Unified Device Architecture (CUDA) has emerged as a popular programming model for GPGPUs for use by C/C++ programmers [1]. Given the widespread use of managed-runtime execution environments, such as the Java Virtual Machine (JVM) and .Net platforms, it is natural to explore how CUDA-like capabilities can be made accessible to programmers who use those environments.

In this paper, we present a programming interface called JCUDA that can be used by Java programmers to invoke CUDA kernels. Using this interface, programmers can write Java codes that directly call CUDA kernels without having to worry about the details of bridging the Java runtime and CUDA runtime. The JCUDA implementation handles data transfers of primitives and multidimensional arrays of primitives between the host and device. Our preliminary performance results obtained on four double-precision floating-point Java Grande benchmarks show that this interface can deliver significant performance improvements to Java programmers. The results for Size C (the largest data size) show speedups ranging from 7.70× to 120.32× with the use of one GPGPU, relative to CPU execution on a single thread.

The rest of the paper is organized as follows. Section 2 briefly summarizes past work on high performance computing in Java, as well as the CUDA programming model. Section 3 introduces the JCUDA programming interface and describes its current implementation. Section 4 presents performance results obtained for JCUDA on four Java Grande benchmarks. Finally, Section 5 discusses related work and Section 6 contains our conclusions.

2 Background

2.1 Java for High Performance and Numerical Computing

A major thrust in enabling Java for high performance computing came from the Java Grande Forum (JGF) [2], a community initiative to promote the use of the Java platform for compute-intensive numerical applications. Past work in the JGF focused on two areas: Numerics, which concentrated on issues with using Java on a single CPU, such as complex arithmetic and multidimensional arrays, and Concurrency, which focused on using Java for parallel and distributed computing. The JGF effort also included the development of benchmarks for measuring and comparing different Java execution environments, such as the JGF [3,4] and SciMark [5] benchmark suites.

The Java Native Interface (JNI) [6], Java's foreign function interface for executing native C code, also played a major role in JGF projects, such as enabling Message Passing Interface (MPI) for Java [7]. In JNI, the programmer declares selected C functions as `native` external methods that can be invoked by a Java program. The native functions are assumed to have been separately compiled into host-specific binary code. After compiling the Java source files, the `javah` utility can be used to generate C header files that contain stub interfaces for the native code. JNI also supports a rich variety of callback functions to enable native code to access Java objects and services.

2.2 GPU Architecture and the CUDA Programming Model

Driven by the insatiable demand for realtime, high-definition 3D gaming and multimedia experiences, the programmable GPU (Graphics Processing Unit) has

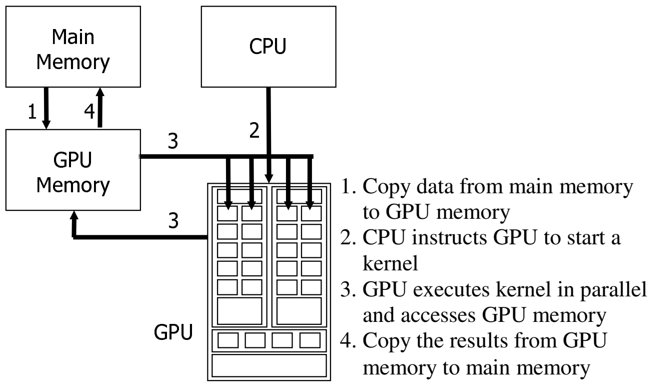

Main Memory
CPU
GPU Memory
GPU
1
2
3
4
1. Copy data from main memory to GPU memory
2. CPU instructs GPU to start a kernel
3. GPU executes kernel in parallel and accesses GPU memory
4. Copy the results from GPU memory to main memory

Fig. 1. Process Flow of a CUDA Kernel Call

evolved into a highly parallel, multithreaded, manycore processor. Current GPUs have tens or hundreds of fragment processors, and higher memory bandwidths than regular CPUs. For example, the NVIDIA GeForce GTX 295 graphics card comes with two GPUs, each with 240 processor cores, and 1.8 GB memory with 233.8 GB/s bandwidth, which is about 10× faster than that of current CPUs. The same GPU is part of the NVIDIA Tesla C1060 Computer Processor, which is the GPU processor used in our performance evaluations.

The idea behind general-purpose computing on graphics processing units (GPGPU) is to use GPUs to accelerate selected computations in applications that are traditionally handled by CPUs. To overcome known limitations and difficulties in using graphics APIs for general-purpose computing, GPU vendors and researchers have developed new programming models, such as NVIDIA's Compute Unified Device Architecture (CUDA) model [1], AMD's Brook+ streaming model [8], and Khronos Group's OpenCL framework [9].

The CUDA programming model is an extension of the C language. Programmers write an application with two portions of code — functions to be executed on the CPU host and functions to be executed on the GPU device. The entry functions of the device code are tagged with a `__global__` keyword, and are referred to as *kernels*. A kernel executes in parallel across a set of parallel threads in a Single Instruction Multiple Thread (SIMT) model [1]. Since the host and device codes execute in two different memory spaces, the host code must include special calls for host-to-device and device-to-host data transfers. Figure 1 shows the sequence of steps involved in a typical CUDA kernel invocation.

3 The JCUDA Programming Interface and Compiler

With the availability of CUDA as an interface for C programmers, the natural extension for Java programmers is to use the Java Native Interface (JNI) as a bridge to CUDA via C. However, as discussed in Section 3.1, this approach is

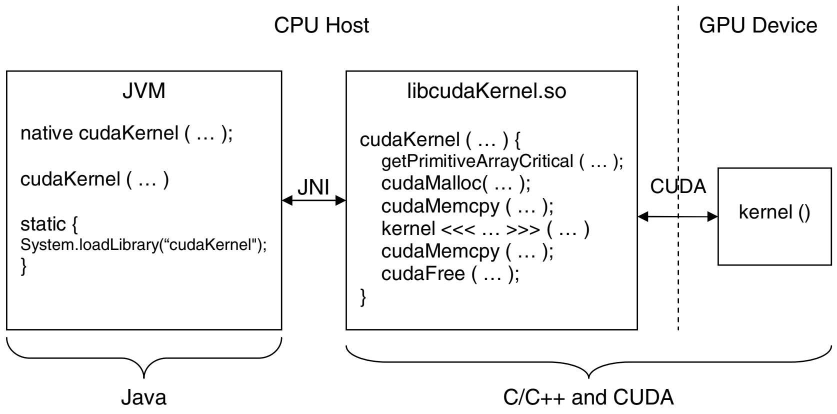


Fig. 2. Development process for accessing CUDA via JNI

neither easy nor productive. Sections 3.2 and 3.3 describe our JCUDA programming interface and compiler, and Section 3.4 summarizes our handling of Java arrays as parameters in kernel calls.

3.1 Current Approach: Using CUDA via JNI

Figure 2 summarizes the three-stage process that a Java programmer needs to follow to access CUDA via JNI today. It involves writing Java code and JNI stub code in C for execution on the CPU host, as well as CUDA code for execution on the GPU device. The stub code also needs to handle allocation and freeing of data in device memory, and data transfers between the host and device. It is clear that this process is tedious and error-prone, and that it would be more productive to use a compiler or programming tool that automatically generates stub code and data transfer calls.

3.2 The JCUDA Programming Interface

The JCUDA model is designed to be a programmer-friendly foreign function interface for invoking CUDA kernels from Java code, especially for programmers who may be familiar with Java and CUDA but not with JNI. We use the example in Figure 3 to illustrate JCUDA syntax and usage. The interface to external CUDA functions is declared in lines 90–93, which contain a static library definition using the `lib` keyword. The two arguments in a `lib` declaration specify the name and location of the external library using string constants. The library definition contains declarations for two external functions, *foo1* and *foo2*. The `acc` modifier indicates that the external function is a CUDA-accelerated kernel function. Each function argument can be declared as IN, OUT, or INOUT to indicate if a data transfer should be performed before the kernel call, after the kernel call or both. These modifiers allows the responsibility of device memory allocation and data transfer to be delegated to the JCUDA compiler. Our current

```
1     double[ ][ ] l_a = new double[NUM1][NUM2];
2     double[ ][ ][ ] l_aout = new double[NUM1][NUM2][NUM3];
3     double[ ][ ] l_aex = new double[NUM1][NUM2];
4
5     initArray(l_a); initArray(l_aex); //initialize value in array
6
7     int [ ] ThreadsPerBlock = {16, 16, 1};
8     int [ ] BlocksPerGrid = new int[3];   BlocksPerGrid[3] = 1;
9     BlocksPerGrid[0] = (NUM1 + ThreadsPerBlock[0] - 1) / ThreadsPerBlock[0];
10    BlocksPerGrid[1] = (NUM2 + ThreadsPerBlock[1] - 1) / ThreadsPerBlock[1];
11
12    /* invoke device on this block/thread grid */
13    cudafoo.foo1 <<<< BlocksPerGrid, ThreadsPerBlock >>>> (l_a, l_aout, l_aex);
14    printArray(l_a); printArray(l_aout); printArray(l_aex);

...             ... ...

90    static lib cudafoo (”cfoo”, ”/opt/cudafoo/lib”) {
91      acc void foo1 (IN double[ ][ ]a, OUT int[ ][ ][ ] aout, INOUT float[ ][ ] aex);
92      acc void foo2 (IN short[ ][ ]a, INOUT double[ ][ ][ ] aex, IN int total);
93    }
```

Fig. 3. JCUDA example

JCUDA implementation only supports scalar primitives and rectangular arrays of primitives as arguments to CUDA kernels. The OUT and INOUT modifiers are only permitted on arrays of primitives, not on scalar primitives. If no modifier is specified for an argument, it is default to be IN. As discussed later in Section 5, there are related approaches to CUDA language bindings that support modifiers such as IN and OUT, and the upcoming PGI 8.0 C/C++ compiler also uses an `acc` modifier to declare regions of code in C programs to be accelerated. To the best of our knowledge, none of the past efforts support direct invocation of user-written CUDA code from Java programs, with automatic support for data transfer (including copying of multidimensional Java arrays).

Line 13 shows a sample invocation of the CUDA kernel function `foo1`. Similar to CUDA's C interface, we use the `<<<<...>>>>`[1] to identify a kernel call. The geometries for the CUDA grid and blocks are specified using two three-element integer arrays, `BlocksPerGrid` and `ThreadsPerBlock`. In this example, the kernel will be executed with $16 \times 16 = 256$ threads per block and by a number of blocks per grid that depends on the input data size (`NUM1` and `NUM2`).

3.3 The JCUDA Compiler

The JCUDA compiler performs source-to-source translation of JCUDA programs to Java program. Our implementation is based on Polyglot [10], a compiler front end for the Java programming language. Figures 4 and 5 show the Java static class declaration and the C glue code generated from the `lib` declaration in Figure 3. The Java static class introduces declarations with mangled names for native functions corresponding to JCUDA functions `foo1` and `foo2` respectively, as well as a static class initializer to load the stub library. In addition, three

[1] We use four angle brackets instead of three as in CUDA syntax because the ">>>" is already used as unsigned right shift operator in Java programming language.

```
  private static class cudafoo {

    native static void HelloL_00024cudafoo_foo1(double[ ][ ] a,
        int[ ][ ][ ] aout, float[ ][ ] aex, int[ ] dimGrid, int[ ] dimBlock, int sizeShared);

    static void foo1(double[ ][ ] a, int[ ][ ][ ] aout, float[ ][ ] aex, int[ ] dimGrid, int[ ] dimBlock, int sizeShared) {
      HelloL_00024cudafoo_foo1(a, aout, aex, dimGrid, dimBlock, sizeShared);
    }

    native static void HelloL_00024cudafoo_foo2(short[ ][ ] a,
        double[ ][ ][ ] aex, int total, int[ ] dimGrid, int[ ] dimBlock, int sizeShared);

    static void foo2(short[ ][ ] a, double[ ][ ][ ] aex, int total, int[ ] dimGrid, int[ ] dimBlock, int sizeShared) {
      HelloL_00024cudafoo_foo2(a, aex, total, dimGrid, dimBlock, sizeShared);
    }

    static { java.lang.System.loadLibrary("HelloL_00024cudafoo_stub"); }
  }
```

Fig. 4. Java static class declaration generated from `lib` definition in Figure 3

```
extern __global__ void foo1(double * d_a, signed int * d_aout, float * d_aex);

JNIEXPORT void JNICALL
Java_HelloL_00024cudafoo_HelloL_100024cudafoo_1foo1(JNIEnv *env, jclass cls, jobjectArray a,
        jobjectArray aout, jobjectArray aex, jintArray dimGrid, jintArray dimBlock, int sizeShared) {
  /* copy array a to the device */
  int dim_a[3] = {2};
  double * d_a = (double*) copyArrayJVMToDevice(env, a, dim_a, sizeof(double));

  /* Allocate array aout on the device */
  int dim_aout[4] = {3};
  signed int * d_aout = (signed int*) allocArrayOnDevice(env, aout, dim_aout, sizeof(signed int));

  /* copy array aex to the device */
  int dim_aex[3] = {2};
  float * d_aex = (float*) copyArrayJVMToDevice(env, aex, dim_aex, sizeof(float));

  /* Initialize the dimension of grid and block in CUDA call */
  dim3 d_dimGrid; getCUDADim3(env, dimGrid, &d_dimGrid);
  dim3 d_dimBlock; getCUDADim3(env, dimBlock, &d_dimBlock);

  foo1 <<< d_dimGrid, d_dimBlock, sizeShared >>> ((double *)d_a, (signed int *)d_aout, (float *)d_aex);

  /* Free device memory d_a */
  freeDeviceMem(d_a);

  /* copy array d_aout->aout from device to JVM, and free device memory d_aout */
  copyArrayDeviceToJVM(env, d_aout, aout, dim_aout, sizeof(signed int));
  freeDeviceMem(d_aout);

  /* copy array d_aex->aex from device to JVM, and free device memory d_aex */
  copyArrayDeviceToJVM(env, d_aex, aex, dim_aex, sizeof(float));
  freeDeviceMem(d_aex);

  return;
}
```

Fig. 5. C glue code generated for the `foo1` function defined in Figure 3

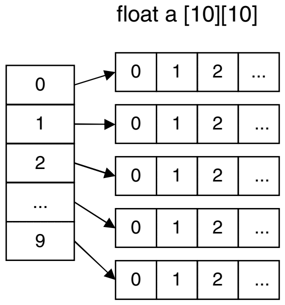


Fig. 6. Java Multidimensional Array Layout

parameters are added to each call — *dimGrid*, *dimBlock*, and *sizeShared* — corresponding to the grid geometry, block geometry, and shared memory size. As we can see in Figure 5, the generated C code inserts host-device data transfer calls in accordance with the IN, OUT and INOUT modifiers in Figure 3.

3.4 Multidimensional Array Issues

A multidimensional array in Java is represented as an array of arrays. For example, a two-dimensional float array is represented as a one-dimensional array of objects, each of which references a one-dimensional float array as shown in the Figure 6. This representation supports general nested and ragged arrays, as well as the ability to pass subarrays as parameters while still preserving pointer safety. However, it has been observed that this generality comes with a large overhead for the common case of multidimensional rectangular arrays [11].

In our work, we focus on the special case of dense rectangular multidimensional arrays of primitive types as in C and Fortran. These arrays are allocated as nested arrays in Java and as contiguous arrays in CUDA. The JCUDA runtime performs the necessary gather and scatter operations when copying array data between the JVM and the GPU device. For example, to copy a Java array of `double[20][40][80]` from the JVM to the GPU device, the JCUDA runtime makes $20 \times 40 = 800$ calls to the CUDA *cudaMemcpy* memory copy function with 80 double-words transferred in each call. In future work, we plan to avoid this overhead by using X10's multidimensional arrays [12] with contiguous storage of all array elements, instead of Java's multidimensional arrays.

4 Performance Evaluation

4.1 Experimental Setup

We use four Section 2 benchmarks from the Java Grande Forum (JGF) Benchmarks [3,4] to evaluate our JCUDA programming interface and compiler — Fourier coefficient analysis (Series), Sparse matrix multiplication (Sparse), Successive over-relaxation (SOR), and IDEA encryption (Crypt). Each of these benchmarks has three problem sizes for evaluation — A, B and C — with Size A being the smallest and Size C the largest. For each of these benchmarks, the compute-intensive portions were rewritten in CUDA whereas the rest of the code was retained in its original Java form except for the JCUDA extensions used for kernel invocation. The rewritten CUDA codes are parallelized in the same way as the original Java multithreaded code, with each CUDA thread performing the same computation as a Java thread.

The GPU used in our performance evaluations is a NVIDIA Tesla C1060 card, containing a GPU with 240 cores in 30 streaming multiprocessors, a 1.3 GHz clock speed, and 4GB memory. It also supports double-precision floating-point operations, which was not available in earlier GPU products from NVIDIA. All benchmarks were evaluated with double-precision arithmetic, as in the original Java versions. The CPU hosting this GPGPU is an Intel Quad-Core CPU with a 2.83GHz clock speed, 12MB L2 Cache and 8GB memory. The software installations used include a Sun Java HotSpot 64-bit virtual machine included in version 1.6.0_07 of the Java SE Development Kit (JDK), version 4.2.4 of the GNU Compiler Collection (gcc), version 180.29 of the NVIDIA CUDA driver, and version 2.0 of the NVIDIA CUDA Toolkit.

There are two key limitations in our JCUDA implementation which will be addressed in future work. First, as mentioned earlier, we only support primitives and rectangular arrays of primitives as function arguments in the JCUDA interface. Second, the current interface does not provide any direct support for reuse of data across kernel calls since the parameter modes are restricted to `IN`, `OUT` and `INOUT`.

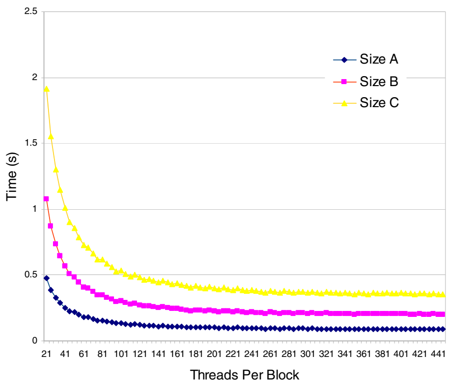
4.2 Evaluation and Analysis

Table 1 shows the execution times (in seconds) of the Java and JCUDA versions for all three data sizes of each of the four benchmarks. The Java execution times

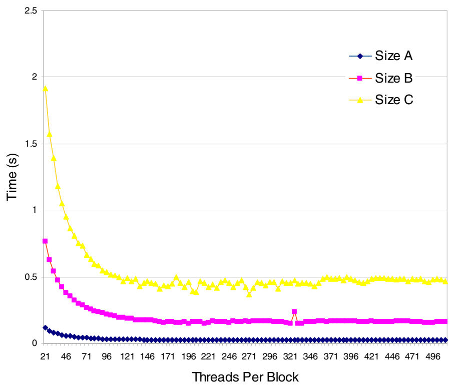
Table 1. Execution times in seconds, and Speedup of JCUDA relative to 1-thread Java executions (30 blocks per grid, 256 threads per block)

Benchmark	Series			Sparse		
Data Size	A	B	C	A	B	C
Java-1-thread execution time	7.6s	77.42s	1219.40s	0.50s	1.17s	19.87s
Java-2-threads execution time	3.84s	39.21s	755.05s	0.26s	0.54s	8.68s
Java-4-threads execution time	2.03s	19.82s	390.98s	0.25s	0.39s	5.32s
JCUDA execution time	0.11s	1.04s	10.14s	0.07s	0.14s	0.93s
JCUDA Speedup w.r.t. Java-1-thread	67.26	74.51	120.32	7.25	8.30	21.27

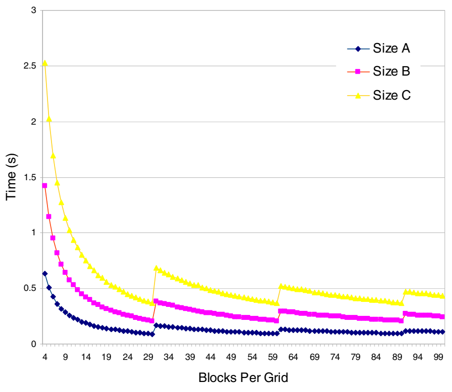
Benchmark	SOR			Crypt		
Data Size	A	B	C	A	B	C
Java-1-thread execution time	0.62s	1.60s	2.82s	0.51s	3.26s	8.16s
Java-2-threads execution time	0.26s	1.32s	2.59s	0.27s	1.65s	4.10s
Java-4-threads execution time	0.16s	1.37s	2.70s	0.11s	0.21s	2.16s
JCUDA execution time	0.09s	0.21s	0.37s	0.02s	0.16s	0.45s
JCUDA Speedup w.r.t. Java-1-thread	6.74	7.73	7.70	22.17	20.12	17.97



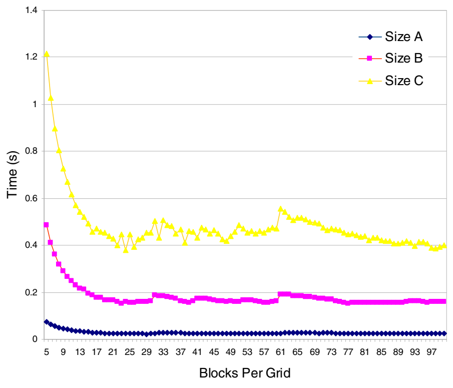
(a) SOR (30 blocks/grid)



(b) Crypt (30 blocks/grid)



(c) SOR (256 threads/block)



(d) Crypt (256 threads/block)

Fig. 7. Execution time by varying threads/block and blocks/grid in CUDA kernel invocations on the Tesla C1060 GPU

were obtained for 1, 2 and 4 threads, representing the performance obtained by using 1, 2, and 4 CPU cores respectively. The JCUDA execution times were obtained by using a single Java thread on the CPU combined with multiple threads on the GPU in a configuration of 256 threads per block and 30 blocks per grid. The JCUDA kernel was repeatedly executed 10 times, and the fastest times attained for each benchmark and test size are listed in the table.

The bottom row shows the speedup of the JCUDA version relative to the 1-thread Java version. The results for Size C (the largest data size) show speedups ranging from 7.70× to 120.32×, whereas smaller speedups were obtained for smaller data sizes — up to 74.51× for Size B and 67.26× for Size A. The benchmark that showed the smallest speedup was SOR. This is partially due to the fact that our CUDA implementation of the SOR kernel performs a large number of barrier (`__syncthreads`) operations — 400, 600, and 800 for sizes A, B, and C respectively. In contrast, the Series examples show the largest speedup because it represents an embarrassingly parallel application with no communication or

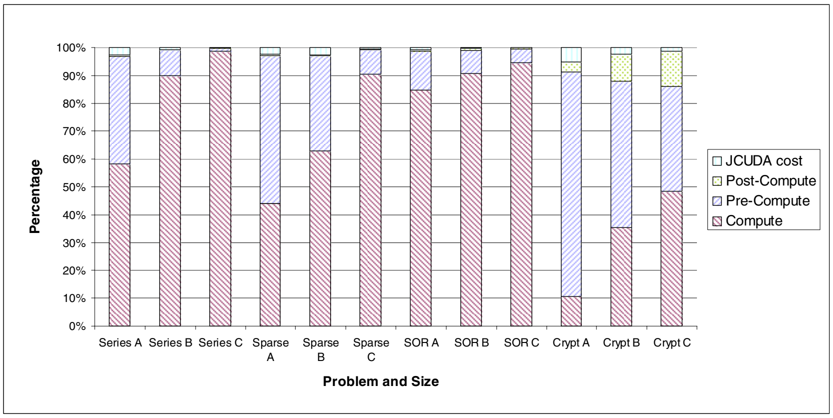


Fig. 8. Breakdown of JCUDA Kernel Execution Times

synchronization during kernel execution. In general, we see that the JCUDA interface can deliver significant speedups for Java applications with double-precision floating point arithmetic by using GPGPUs to offload computations.

Figure 7 shows the results of varying threads/block and blocks/grid for SOR and Crypt to study the performance variation relative to the geometry assumed in Table 1 (256 threads/block and 30 blocks/grid). In Figures 7(a) and 7(b), for 30 blocks per grid, we see that little performance gain is obtained by increasing the number of threads per block beyond 256. This suggests that 256 threads per block (half the maximum value of 512) is a large enough number to enable the GPU to maintain high occupancy of its cores [13]. Next, in Figures 7(c) and 7(d), for 256 threads per block, we see that execution times are lowest when the blocks/grid is a multiple of 30. This is consistent with the fact that the Tesla C1060 has 30 streaming multiprocessors (SM). Further, we observe a degradation when the blocks/grid is 1 more than a multiple of 30 (31, 61, 91, . . .) because of the poor load balance that results on a 30-SM GPU processor.

Figure 8 shows the percentage breakdown of the JCUDA kernel execution times into *compute*, *pre-compute*, *post-compute* and *JCUDA cost* components. The compute component is the time spent in kernel execution on the GPGPU. The pre-compute and post-compute components represent the time spent in data transfer before and after kernel execution. The JCUDA cost includes the overhead of the JNI calls generated by the JCUDA kernel. As the figure shows, for each benchmark, as the problem becomes larger, the *compute* time percentage increases. The JCUDA overhead has minimal impact on performance, especially for Size C executions. The impact of *pre-compute* and *post-compute* times depends on the communication requirements of individual benchmarks.

5 Related Work

In this section, we briefly discuss two areas of related work in making CUDA kernels accessible to other languages. The first approach is to provide library bindings for the CUDA API. Examples of this approach include the JCublas and JCufft [14] Java libraries that provide Java bindings to the standard CUBLAS and CUFFT libraries from CUDA. We recently learned that JCublas and JCufft are part of a new Java library called jCUDA. In contrast, the JCUDA interface introduced in this paper is a language-based approach, in which the compiler automatically generates glue code and data transfers.

PyCuda [15] from Python community is a Python binding for the CUDA API. It also allows CUDA kernel code to be embedded as a string in a Python program. RapidMind provides a multicore programming framework based on the C++ programming language [16]. A programmer can embed a kernel intended to run on the GPU as a delimited piece of code directly in the program. RapidMind also supports IN, and OUT keywords for function parameters to guide the compiler in generating appropriate code for data movement.

The second approach is to use compiler parallelization to generate CUDA kernel code. A notable example of this approach can be found in the PGI 8.0 x64+GPU compilers for C and Fortran which allow programmers to use directives, such as *acc*, *copyin*, and *copyout*, to specify regions of code or functions to be GPU-accelerated. The compiler splits portions of the application between CPU and GPU based on these user-specified directives and generates object codes for both CPU and GPUs from a single source file. Another related approach is to translate OpenMP parallel code to hybrid code for CPUs and GPUs using compiler analysis, transformation and optimization techniques [17]. A recent compiler research effort that is of relevance to Java can be found in [18], where the Jikes RVM dynamic optimizing compiler [19] is extended to perform automatic parallelization at the bytecode level.

To the best of our knowledge, none of the past efforts support direct invocation of user-written CUDA code from Java programs, with automatic support for data transfer of primitives and multidimensional arrays of primitives.

6 Conclusions and Future Work

In this paper, we presented the JCUDA programming interface that can be used by Java programmers to invoke CUDA kernels without having to worry about the details of JNI calls and data transfers (especially for multidimensional arrays) between the host and device. The JCUDA compiler generates all the necessary JNI glue code, and the JCUDA runtime handles data transfer before and after each kernel call. Our preliminary performance results obtained on four Java Grande benchmarks show significant performance improvements of the JCUDA-accelerated versions over their original versions. The results for Size C (the largest data size) showed speedups ranging from 6.74× to 120.32× with the use of a GPGPU, relative to CPU execution on a single thread. We also discussed the impact of problem size, communication overhead, and synchronization overhead on the speedups that were obtained.

To address the limitations listed in Section 4.1, we are considering to add a KEEP modifier for an argument to specify GPU resident data between kernel calls. To support the overlapping of computation and data transfer, we are developing memory copy primitives for programmers to initiate asynchronous data copy between CPU and GPU. Those primitives can use either the conventional *cudaMemcpy* operation or the page-locked memory mapping mechanism introduced in the latest CUDA development kit. In addition to those, there are also several interesting directions for our future research. We plan to use the JCUDA interface as a target language to support high-level parallel programming language like X10 [12] and Habanero-Java in the Habanero Multicore Software Research project [20]. In this approach, we envision generating JCUDA code and CUDA kernels from a single source multi-place X10 program. Pursuing this approach will require solving some interesting new research problems such as ensuring that offloading X10 code onto a GPGPU will not change the exception semantics of the program. Another direction that we would like to explore is the use of the language extensions recommended in [21] to simplify automatic parallelization and generation of CUDA code.

Acknowledgments

We are grateful to Jisheng Zhao and Jun Shirako for their help with the Polyglot compiler infrastructure and JGF benchmarks. We would also like to thank Tim Warburton and Jeffrey Bridge for their advice on constructing the GPGPU system used to obtain the experimental results reported in this paper.

References

1. Nickolls, J., Buck, I., Garland, M., Nvidia, Skadron, K.: Scalable Parallel Programming with CUDA. ACM Queue 6(2), 40–53 (2008)
2. Java Grande Forum Panel, Java Grande Forum Report: Making Java Work for High-End Computing, Java Grande Forum, SC 1998, Tech. Rep. (November 1998)
3. Bull, J.M., Smith, L.A., Pottage, L., Freeman, R.: Benchmarking Java Against C and Fortran for Scientific Applications. In: Proceedings of the 2001 joint ACM-ISCOPE Conference on Java Grande, pp. 97–105. ACM Press, New York (2001)
4. Smith, L.A., Bull, J.M., Obdrzálek, J.: A Parallel Java Grande Benchmark Suite. In: Proceedings of the 2001 ACM/IEEE conference on Supercomputing, p. 8. ACM Press, New York (2001)
5. SciMark Java Benchmark for Scientific and Numerical Computing, `http://math.nist.gov/scimark2/`
6. Liang, S.: Java Native Interface: Programmer's Guide and Specification. Sun Microsystems (1999)
7. Carpenter, B., Getov, V., Judd, G., Skjellum, A., Fox, G.: MPJ: MPI-Like Message Passing for Java. Concurrency - Practice and Experience 12(11), 1019–1038 (2000)
8. AMD, ATI Stream Computing - Technical Overview. AMD, Tech. Rep. (2008)
9. Khronos OpenCL Working Group, The OpenCL Specification - Version 1.0. The Khronos Group, Tech. Rep. (2009)
10. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An Extensible Compiler Framework for Java. In: Kahng, H.-K. (ed.) ICOIN 2003. LNCS, vol. 2662, pp. 138–152. Springer, Heidelberg (2003)

11. Moreira, J.E., Midkiff, S.P., Gupta, M., Artigas, P.V., Snir, M., Lawrence, R.D.: Java Programming for High-Performance Numerical Computing. IBM Systems Journal 39(1), 21–56 (2000)
12. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. In: OOPSLA 2005: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 519–538. ACM, New York (2005)
13. NVIDIA, NVIDIA CUDA Programming Guide 2.2.plus 0.5em minus 0.4emNVIDIA (2009), `http://www.nvidia.com/cuda`
14. Java Binding for NVIDIA CUDA BLAS and FFT Implementation, `http://www.jcuda.de`
15. PyCuda, `http://documen.tician.de/pycuda/`
16. Matthew Monteyne, RapidMind Multi-Core Development Platform. RapidMind Inc., Tech. Rep (2008)
17. Lee, S., Min, S.-J., Eigenmann, R.: Openmp to gpgpu: a compiler framework for automatic translation and optimization. In: PPoPP 2009: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 101–110. ACM, New York (2009)
18. Leung, A.C.-W.: Thesis: Automatic Parallelization for Graphics Processing Units in JikesRVM. University of Waterloo, Tech. Rep. (2008)
19. Alpern, B., Augart, S., Blackburn, S.M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., McKinley, K.S., Mergen, M., Moss, J.E.B., Ngo, T., Sarkar, V.: The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. IBM Systems Journal 44(2), 399–417 (2005)
20. Habanero Multicore Software Project, `http://habanero.rice.edu`
21. Shirako, J., Kasahara, H., Sarkar, V.: Language Extensions in Support of Compiler Parallelization. In: Adve, V., Garzarán, M.J., Petersen, P. (eds.) LCPC 2007. LNCS, vol. 5234, pp. 78–94. Springer, Heidelberg (2008)