JCudaMP: OpenMP/Java on CUDA

Georg Dotzler, Ronald Veldema, Michael Klemm
University of Erlangen-Nuremberg
Computer Science Department 2
Martensstr. 3 • 91058 Erlangen • Germany
{georg.dotzler, veldema, klemm}@informatik.uni-erlangen.de

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*; D.3.4 [**Programming Languages**]: Processors—*Code generation, Run-time environments*

ABSTRACT

We present an OpenMP framework for Java that can exploit an available graphics card as an application accelerator. Dynamic languages (Java, C#, etc.) pose a challenge here because of their write-once-run-everywhere approach. This renders it impossible to make compile-time assumptions on whether and which type of accelerator or graphics card might be available in the system at run-time.

We present an execution model that dynamically analyzes the running environment to find out what hardware is attached. Based on the results it dynamically rewrites the bytecode and generates the necessary gpGPU code on-the-fly.

Furthermore, we solve two extra problems caused by the combination of Java and CUDA. First, CUDA-capable hardware usually has little memory (compared to main memory). However, as Java is a pointer-free language, array data can be stored in main memory and buffered in GPU memory. Second, CUDA requires one to copy data to and from the graphics card's memory explicitly. As modern languages use many small objects, this would involve many copy operations when done naively. This is exacerbated because Java uses arrays-of-arrays to implement multi-dimensional arrays. A clever copying technique and two new array packages allow for more efficient use of CUDA.

1. INTRODUCTION

Currently, many computers already have a graphics card that is capable of doing generic computations using, for example, CUDA [1] (nVidia). Since their core counts will grow and their cores will support even more generic computations, general purpose GPU (gpGPU) computing will become even more attractive. The common computing environment is therefore already heterogeneous and we need a programming environment to transparently adapt to this heterogeneity.

In this situation one can learn from Java's or .NET's write-once-run-everywhere approach: the programmer no longer has to deal with various native executables or architectural details of machines. The same is needed for (heterogeneous) multi-core architectures that might or might not be equipped with gpGPUs or other accelerators. The programmer cannot be expected to explicitly write code for any chip combination. Instead, the programmer will expect that for such modern, dynamic, object-oriented languages, the runtime environment will make sure that the execution will exploit (any) gpGPU hardware when it is available, without any significant changes to the source code. To do so, we need to annotate the bytecode (here Java bytecode) to mark parallel code so that it can be easily found at run-time. We also need to dynamically generate code for the current hardware architecture and hardware versions.

To circumvent the low-level parallel programming model of Java and to focus on SPMD-style parallelism that is suitable for graphics cards, we use JaMP [2] which provides OpenMP-like annotations to Java code. The JaMP compiler is an OpenMP compiler that translates OpenMP/Java code to multi-threaded Java bytecode. We extend the existing compiler to emit Java bytecode annotations. A new Java class loader reads the annotations and performs run-time code rewriting to adapt the bytecode to whatever multi-/many-core hardware is available in the system.

The new Java class loader detects the presence and different versions of CUDA (each version has different capabilities and memory sizes). Other hardware that falls in the same category of parallel co-processor hardware are the Cell-processor [3], Brook+ [4] (AMD's gpGPU platform), and Intel's future Larrabee [5] graphics/parallel processor.

Once we can detect and use CUDA dynamically, the two important problems are (1) to achieve good memory transfer performance and (2) to allocate huge data structures on the graphics card. We solve (1) by efficiently packing many objects into big chunks for bulk transfers and by providing specialized array classes that allocate their payload on the gpGPU (if available) and copy elements to the host only when needed. An OpenMP extension solves problem (2) by semi-automatically fragmenting an array into smaller tiles that are just big enough to fit onto the graphics card.

One could argue that a library based approach would be sufficient. Such a library would contain a number of pre-programmed, hand-optimized, compute kernels for a number of hardware types. There are two problems with such an approach. First, the pre-selected kernels will not fit all new problem domains or algorithms and second, new hardware will probably require new interfaces and so new libraries. However, in our approach, the programmer can write his own parallel region and let some lower-level software layer optimize and adapt it to available hardware. For new hardware, the programs themselves can thereafter remain untouched with only the class-loader/compiler changed.

IWMSE '10, May 1 2010, Cape Town, South Africa

The main contributions of this paper are a technique to dynamically invade an available gpGPU (Sec.2), ways to minimize host-gpGPU object transfers (Sec. 3), and finally, a buffering that semi-automatically hides memory limits of attached graphics cards and enables the processing of large data structures (Sec. 4).

2. CODE GENERATION

2.1 Architecture

To optimize the parallel region to the platform's hardware, our framework must dynamically generate specialized code at run-time because the target hardware is not known at compile-time. This code specialization can occur in the VM (e.g. the JIT compiler) or in an extended Java class loader. Changing a VM requires many low level system dependent patches (this includes VM recompilation for every platform) whereas specifying a new class loader only requires a new JVM command line argument and is portable to all VMs. We therefore use a class loader based approach.

Fig. 1 shows our compilation and execution pipeline.

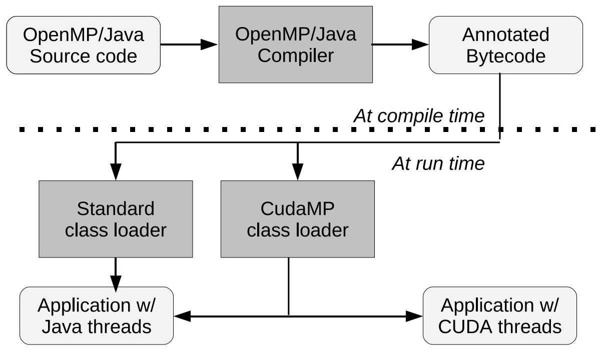

Figure 1: Execution model of OpenMP/Java programs in CUDA environments.

First, a programmer creates Java code enriched with OpenMP/-Java pragmas to tag code that is amenable to parallelization. For example:

```
// Original code:
class Demo {
  void work(int []arr) {
    //#omp parallel for
    for (int i = S;i < E; i += K) {
      arr[i]++;
    }
  }
}

// Generated for Demo.work
class DemoWorker extends Worker {
  int arr[],S,E,K; ...
  void run() {
    Job w; JCudaMP.init(S,E,K);
    while ((w=JCudaMP.getWork()) != NULL) {
      for (int i=w.start;i<w.end;i+=w.step){
        arr[i]++;
} } } }
// patched Demo:
class Demo {
 void work(int []arr) {
  JCudaMP.start(new DemoWorker(S,E,K, arr));
} }
```

The pragma tells the OpenMP compiler that the `for` loop can be executed in parallel, as each loop iteration is completely independent from others. Without prescribing how to do the parallelization, this code is then translated by the OpenMP/Java compiler to bytecode, effectively by generating the code below (multi-threaded code). If the class loader discovers CUDA-enabled hardware, it will rewrite it later on-the-fly.

First, the loop's body is placed in a new method (here `DemoWorker.run`) and surrounded with logic that requests chunks of the iteration space from the runtime. A sequential loop then processes the chunk's iterations. The original parallel code is replaced by a call to `JCudaMP.start` and a `DemoWorker` instance as argument.

A standard class loader will ignore the special bytecode section and will process the regular threaded implementation. In contrast, our JCudaMP class loader analyzes the original loop body in `DemoWorker.run` to see if it is CUDA-amenable. From the body of the inner loop in `DemoWorker.run` we generate CUDA code.
The `JCudaMP.start` call is replaced by a call to `DemoWorker.runCuda` which calls the generated CUDA code (via JNI).

We currently do not allow CUDA execution if the code contains meta-object access, native calls, try-catch statements, general object allocations, or calls to methods that the class loader has not previously marked as being CUDA-amenable. Meta-object access includes class casts, instance-of tests, and accesses to `Class` and its associates in the `java.lang.reflect` package. Object allocations are not allowed as memory is scarce on graphics cards. Arbitrary method calls are not allowed as CUDA-threads have very limited (call) stack space. Only Methods of the `CudaRuntimeFunctions` class can be used. For example, this class contains functions like `sinf`, `cosf`. In Java mode `Math.sin`, `Math.cos` are used. In CUDA mode, the class loader replaces them by the `sinf` and `cosf` functions provided by CUDA. Unfortunately, depending on hardware precision, the results of these functions could differ from the Java version. In addition, the OpenMP functions `omp_get_num_threads` and `omp_get_thread_num` are allowed. This is necessary for programs that expect certain thread IDs. If code is used inside the parallel region that prevents the execution on CUDA the compiler and classloader report a warning.

At run-time, we invoke the CUDA compiler to generate a shared library and load it into the running JVM. However, this compilation process imposes some overhead. It requires to write the CUDA source code to disk, to invoke the (heavy weight) CUDA compiler to generate assembly code, to pass that to the assembler and linker, and finally to load the result into the VM by the shared library loader. The generated C/CUDA code is then called from Java by an inserted JNI call that replaces the call to `JCudaMP.start`. To reduce this cost, the class loader uses a second thread to compile all the parallel regions of a class in one go, so that compilation, assembling and linking are shared.

There is a variety of different CUDA-capable cards, each with different capabilities (support for `double, long,` etc.) and memory sizes (from 256 MB to 4 GB). To deal with these differences, the class loader performs a test at start-up to determine the capabilities of the available hardware. The results of this test are used for the code generation decisions that are described later in this paper.

2.2 Introduction to CUDA

To better understand CUDA code generation, some CUDA details need to be known. CUDA extends C by some flavor of a parallel `for` loop that invokes a function (called a kernel) a number of times in parallel on the graphics card. For example:

```
foo<<<block_size, grid_size,
      smem_size>>>(a, b, c);
```

causes `block_size` × `grid_size` parallel invocations of `foo` with arguments `a, b, c`. A group of `block_size` parallel invocations operate in SPMD fashion and can access `smem_size` bytes of fast shared memory. These groups of parallel invocations are then effectively repeated `grid_size` times (although with enough resources, some of these groups could also be invoked in parallel to each other as well).

Note that the host's memory is not directly accessible from CUDA. So before issuing a parallel call, we must allocate some graphics card memory with `cudaMalloc()` and copy data to the graphics card with `cudaMemcpy()`. After the parallel call it might be necessary to copy it back to main memory.

2.3 CUDA code generation

For CUDA-amenable parallel regions our class loader generates CUDA code. Because bytecode instructions are for a stack machine and maintaining an actual stack for execution is expensive (due to the resulting memory accesses required), we first translate the Java bytecode to a register assignment representation. From the register representation we then generate C/CUDA code.

The generation of code for array accesses needs a more detailed discussion since (1) array access can cause index-out-of-bounds exceptions, (2) each array has its length associated with it (the array.length field), and (3) multi-dimensional arrays in Java are specified as arrays-of-arrays instead of true multi-dimensional arrays.

We solve (1) by giving each CUDA-kernel a pointer to a (CUDA-side) variable which is set to '1' if an index violation occurs (initially zero). Upon returning from the kernel, we inspect the flag and throw the actual exception on the host.

We solve (2) by giving each array a companion data structure holding its length (or lengths if its a multi-dimensional array). We chose not to prefix each array with an integer holding its size as the CUDA hardware prefers aligned array access.

We do not solve (3) and use arrays-of-arrays at the CUDA side as well. This can cause the usual performance problems as each extra dimension access causes a memory access to get the pointer to a sub-array.

Taking it all together, for an array access to 'a[i][j]' we by default generate a CUDA kernel like this:

```
__global__
void generated_kernel(int *exception_type,
        int ***a_data, int** a_length,
        unsigned int i, unsigned int j) {
    int boundary_check_num;
    if ((i >= a_length[0][0] ||
        (j >= a_length[1][i])) {
        boundary_check_num = 1;
        goto exception_return;
    }
    int tmp = a_data[i][j];
    ..
    return;

    exception_return:
      *exception_type = boundary_check_num;
    return;
  }
```

The `a_data` holds the contents and the `a_length` array holds the lengths of the 1D and 2D sub-arrays.

To handle an index-out-of-bounds exception, `exception_type` is used. It is tested after return from the kernel to throw the proper Java exception on the host. The value returned is the index of the bounds check over all bounds checks in the kernel. This is used to throw the proper Java exception upon return from the kernel.

If multiple kernels throw index-out-of-bounds exceptions, one is chosen arbitrarily (fully accounting for all kernels would cause too much overhead to test afterward for small kernels).

3. MEMORY MANAGEMENT

Since CUDA kernels can only access the local memory of a graphics card, explicit transfers between host and graphics card are needed. Specifically, this requires first that GPU memory is allocated and released and second that data is copied to and from the GPU. Both operations are expensive as they involve context switches.

It turned out to be more efficient to implement a wrapper around CUDA's `malloc`. The wrapper allocates a big chunk of GPU memory and then sub-allocates within that bigger chunk on the host. The CUDA runtime system does not see this sub-allocation.

To copy memory (via DMA) to and from the GPU over the PCIe bus involves expensive context switches that reduce the available bandwidth considerably. This is especially noticeable for smaller data structures due to the latency.

In the next few subsections we will therefore examine a number of ways to reduce the number of host-to-GPU transfers.

3.1 Avoid transfers with data-sharing clauses

As observed in [6], some copying can be avoided by properly tagging the data with OpenMP data-sharing clauses. For example, if the data is marked `firstprivate`, the data needs only to be copied from host to card but not back again. If the data is `lastprivate`, the data needs only to be copied out of the card, never into the card. Only if the data is marked `shared` does it require both a copy-in and a copy-out of the card. Future versions could try determine the variable scope automatically. This is also discussed in [7].

3.2 Efficiently creating a CUDA data image

Since CUDA kernels can not only access flat data structures but can also access data structures with pointers we need to take care of pointer translations when performing data transfers from and to the GPU. Java's normal object serialization copies a graph of data objects into a buffer and makes sure to ship objects only once, even if they are referred to more than once. The recipient will unpack all the objects and revector the pointers according to the objects' new addresses. In JCudaMP we reuse the idea to bulk transfer a buffer of data. But in contrast to normal object serialization, we create the complete CUDA image on the host, i.e., all objects references in the serialization buffer are made into CUDA-side pointers, so that the shipped objects can be used without any further processing on the CUDA-side.

In addition, JCudaMP's serialization exploits the fact, that CUDA-amenable code may not use all kinds of classes, but only instances of `final` subclasses of `CudaObject` are permitted to be used. The current prototype does now allow a `CudaObject` subclass to contain any references to other objects or arrays. This greatly simplifies (and speeds up) the serialization process as no cycle detection needs to be performed to see if an object has been serialized before. Also, this simplifies the compilation process as we can beforehand easily determine which methods need to be compiled to CUDA to be available for calling on the graphics card from inside a compute kernel.

3.3 GPU-based array data

By default, regular data is allocated on the host which makes it cheap to use by the CPU but expensive to use by CUDA (data needs to be copied to and from the GPU). Because we cannot be certain that the host will use the data after the kernel call, we need to conservatively copy all the data back to the host. For the next kernel call (perhaps the same call in a loop), we often need to copy the same data back into the GPU again even if the host did not use or change the data at all. Doing some form of compile-time analysis or host-side read- or write-detection is expensive and even unnecessary if no CUDA code was generated at all for a kernel.

Consider the following scenario; a fluid dynamics problem iteratively updates a large 2D array:

```
void work(float[][] data) {
  for (int i=0;i<NUM_ITERATIONS;i++) {
    update_data(data);
  }
}
```

For a GPU kernel called in `update_data`, `data` needs to be copied to and from the GPU even though only code from the CUDA-amenable parallel region accesses it. Because we cannot a priori know that this is the case, we are forced to do the copying. Static analysis of the bytecode by the class loader is too expensive (as class loading happens at run time). Static analysis of the Java code at compile time will not work either as this analysis requires a closed-world assumption which does not hold for Java. Also, the class loader can not 'undo' transformations it made on code that it has already loaded.

Our solution is an array package that keeps the data on the GPU at all times. Because we want both performance and ease of implementation, we supply an array class for each combination of primitive type and number of dimensions (1–4) (e.g. `Int2DArray`, `Float3DArray`, etc.) Each of these classes has a constructor to pass the required sizes and each has `get(index)` and `set(index, value)` methods to access array elements. If the host code wants to access an array element of an array managed by the package, it will temporarily cache a block of array elements. Array elements remain cached on the host until either the (fixed-size) cache is full or a CUDA-call is made. Cached elements are then copied to CUDA.

The example from above is now rewritten to use this array package. First the OpenMP version; second the corresponding CUDA kernel.

```
//#omp parallel for
for(int x=0;x<WIDTH;x++) {
  for(int y=0;y<HEIGHT;y++) {
    data.set(x, y, compute_updated_value());
  }
}
// generated CUDA/C code:
void cuda_kernel() {
  int x = ...
  int y = ...
  CUDA_ARRAY_SET_xxx(data, x, y,
                    compute_updated_value());
}
```

With a standard class loader or when no CUDA is available, the code works as described above; the array class adapts. The JCudaMP class loader, however, generates CUDA code and uses a form of semantic inlining [8] to improve performance. The get/set methods when invoked on the host, perform JNI calls to access GPU memory using `cudaMemcpy`. When the class loader sees a call to `set/get` when generating CUDA code, it replaces the call with direct memory accesses to array elements. Since the classes of our array package are final, `set/get` calls are easily detected and statically found (and not hidden by polymorphism). At run-time, the serialization code to copy objects to and from GPU memory is also aware of the array package and will instead pass the internal pointer of the array. Just like Java allows access to sub-arrays, our array classes provide sub-array-class access, since it can save many indirections and array bounds-checks in the inner loops.

To give the user a choice, we also added true multi-dimensional array versions of the array package. The multi-dimensional array classes use address arithmetic to map multi-dimensional accesses to a one-dimensional array. These array classes can be more efficient in certain applications and on certain hardware.

Deallocation of graphics card memory used by an array package, is done via the object's finalizer. If the object's finalizer is called by the garbage collector, it frees the associated graphics memory. Normal arrays/objects are copied in-and-out of the card immediately for every kernel invocation (and their graphics card memory freed) so that garbage collection is not an issue there.

3.4 Hiding the Array Packages

The array packages provide a good solution to avoid unnecessary copy operations to the GPU but they require some effort by the programmer. A developer has to select the appropriate package for a given problem (e.g. to choose between true multi-dimensional and normal packages). Furthermore, it is necessary to replace the array uses in the code with getter and setter methods.

To solve these problems, we provide an annotation syntax that the programmer can use to tell the compiler that a given array is *managed*. A managed array is to be implemented with an array package.

The compiler then selects a array package and replaces the Java array with the package. The programmer can optionally provide hints to aid the compiler in the selection task, for example by providing a hint that a rectangular array would be preferred.

For example:

```
class ArrayAnnotationDemo {
  void foo() {
    //#omp managed(a:rectangular)
    int[][] a = new int[16][16];

    //#omp parallel for
    for (int i=0;i<16;i++)
      for (int j=0;j<16;j++)
        a[i][j] = 12345;

    Another.zoo(a);
  }
}

class Another {
  //#omp managed(a:rectangular)
  static void zoo(int[] a) {...}
}
```

This annotation causes the compiler to rewrite the code by changing the type of `'a'` to a multi-dimensional array package class. The `a[i][j]` access in the loop will be rewritten to a setter call and, as described above replaced by the class loader if necessary.

Note that this annotation changes the type of the listed arrays and all uses of the arrays outside the method also need to be annotated. For example, if `'a'` in the above example is passed to another function, that function is required to have the `managed` type annotation as well.

Using a type-annotation instead of a new array syntax for managed arrays is advantageous because a standard Java compiler that is unaware of our annotation will still accept the program as normal Java.

4. TILING OF HUGE ARRAYS

Compared to main memory (up to 128 GB) current graphics cards have only a small memory (up to 4 GB) that is too small to hold the complete working set of many scientific problems (that just barely fit into main memory). The data therefore has to be processed in a tiled fashion. Since, unfortunately, neither static analysis nor the class loader or JIT at run time can determine the sizes of the various arrays in a program beforehand. This would require the insertion of dynamic tests which incur runtime overhead. We therefore need the programmer help to identify huge arrays at compile time.

Our newly introduced `tiled(X, ...)` clause extends OpenMP to state that array `X` is huge and should be processed in a tiled fashion. This annotation causes the array to be partitioned into tiles and automatically creates an additional `for` loop that iterates over the tiles. The size of a tile of one array depends on both the number of other (huge) arrays processed in the `parallel for` loop and the size of the available GPU memory.

Unfortunately, we can no longer support random access to elements of arrays marked as tiled, i.e., it is not allowed to access array elements outside of the current tile. To relax this restriction a bit, tiles can overlap for a few elements on their sides.

```
  //#omp parallel for tiled(
                         dst: { x: 0, 0; y:0, 0},
                         src: { x:-1, 1; y:0, 1})
  for(int x=1; x < WIDTH-1; x++) {
    for(int y=1; y < HEIGHT-1; y++) {
      float value1 = src[x - 1, y + 1];
      float value2 = src[x + 1, y];
      dst[x, y] = value1 + value2;
    }
  }
```

In the example `dst` and `src` are two-dimensional and so two nested `for` loops must follow the `tiled` clause. The first argument to the annotation indicates that `dst` should be tiled and that all iterations access it only by index expressions `x` and `y`. The second argument announces that `src` can be indexed by `x+1`, `x`, and `x-1` in the first dimension and by `y` and `y+1` in the second dimension.

Automatically the code is first transformed into two outer loops that iterate over the tiles. Two nested loops then (in parallel) execute over the elements of each tile to do the actual work:

```
BlockSizeCalculator calc =
                    new BlockSizeCalculator();
calc.add(dst, x, 0, 0);
calc.add(dst, y, 0, 0);

calc.add(src, x, -1, 1);
calc.add(src, y, 0, 1);

int tile_size_x=calc.compute_x_file_size();
int tile_size_y=calc.compute_y_file_size();

for (tmp_x = 1; tmp_x < WIDTH - 1;
                tmp_x += tile_size_x)
  for (tmp_y = 1; tmp_y < HEIGHT - 1;
                  tmp_y += tile_size_y)
        process_tile(tmp_x, tmp_y);

void process_tile(int tmp_x, int tmp_y) {
  // copy data to tile:
  float[][] tile_src = calc.copy_to_tile(
                    src, tmp_x, tmp_y);
  float[][] tile_dst = calc.copy_to_tile(
                    dst, tmp_x, tmp_y);
  int boundary_x = min(tmp_x + tile_size_x,
                         WIDTH);
  int boundary_y = min(tmp_y + tile_size_y,
                         HEIGHT);
  int dst_x_diff = calc.compute_diff_x(dst);
  int dst_y_diff = calc.compute_diff_y(dst);
  int src_x_diff = calc.compute_diff_x(src);
  int src_y_diff = calc.compute_diff_y(src);

  //#omp parallel for
  for(int x=tmp_x; x < boundary_x; x++) {
    for(int y=tmp_t; y < boundary_y; y++) {
      int new_x_src = x - src_x_diff;
      int new_y_src = y - src_y_diff;
      int new_x_dst = x - dst_x_diff;
      int new_y_dst = y - dst_y_diff;
      float value1 = src[new_x_src - 1,
                         new_y_src + 1];
      float value2 = src[new_x_src + 1,
                         new_y_src];
      tile_dst[new_x_dst, new_y_dst] =
                         value1 + value2;
    } }

  calc.copy_from_tile(tile_src, src,
                      tmp_x, tmp_y);
  calc.copy_from_tile(tile_dst, dst,
                      tmp_x, tmp_y);
}
```

Based on the arrays, the indexes, and their ranges, the `BlockSizeCalculator` object must figure out the appropriate tile sizes. It divides the available CUDA memory (established during the initial access of the hardware) between the fixed sized tiles of all arrays. The tiles of each array are made as large as possible in each dimension. Using large tiles reduces the number of DMA transfers to and from the GPU significantly. Note that this transformation occurs at compile time and not at run time in the class loader. If a programmer therefore writes the annotation, it is used for both the threaded version and the generated CUDA version. In the threaded version, `copy_data_to_tile` returns the original array reference to eliminate any copying overhead.

The `BlockSizeCalculator` can fail if the index ranges are chosen too large. For example, the programmer could have added the clause
`tiled(src: {x:-size,size; y:-size,size}).`
This forces the tile size to be at least $(size*2)^2$ which can be larger than the available GPU memory. Also, all huge arrays should be mentioned in the `tiled` clause as otherwise these would not be tiled and then would unnecessarily occupy scarce GPU memory.

5. PERFORMANCE

To test performance we use a simple matrix multiplication code as a micro benchmark and a number of bigger numerical applications. The machine used for testing is an eight core Intel Xeon (2.5 GHz, x86-64, 16 GB main memory) and a GeForce GTX 280 with 1 GB DDR3 memory (PCI Express 2.0 x16). The software environment is Linux kernel 2.6.24 with CUDA 2.2, GCC version 4.2.4 and Java Hotspot JIT in Version 1.6.0_13.

For all benchmarks we use the same set of runs. We perform a run with only *one Java thread* and a run with *7 Java threads* (leaving one core free for operating system, JVM, etc.) with CUDA disabled. We then enable CUDA code generation and see what happens when we try to disable background compilation (*CUDA* row in the performance tables that follow) or enable it (*Async Comp* row). Note that the times mentioned in the *Async Comp* row do not reflect the total compilation times, but instead give only the time needed to actively wait for CUDA compilation to finish. In addition, the final two rows use the arrays-of-arrays package (*APkg* row) and the true multi-dimensional array package (*APkgRect* row). Both allocate the data on the GPU to reduce copying. The rectangular version wins due to the reduced number of memory accesses.

For each of the following performance tables, the *Wall* column is the total runtime from end to end with all overheads included. The next column (compile-time overhead) shows how long it took the CUDA compiler to generate a shared library for the kernel(s). Management overhead is the time required to allocate (and release) CUDA memory, serialize and map the data to the GPU's address space, and to copy the data to the graphics card. All times are in seconds.

5.1 Matrix Multiplication Micro Benchmark

Here we present the measurements for a few versions of matrix-multiplication for two matrix sizes (Table 1). Going from one to seven regular threads achieves a good speedup: 6.7 on 7 cores. With the larger matrix, the speedup decreases to 3.3 because the CPU caches are no longer effective with 7 threads. When generating CUDA code, performance increases dramatically. Note the overhead of a dynamic run of the CUDA compiler. This cost can be reduced slightly by compiling in the background (while other classes are being loaded by the class loader). Allocating the array data on the GPU helps (*APkg* and *APkgRect*). The array package versions reduce the number of DMA transfers. The multi-dimensional array package wins by more than a factor of two on the large matrix size. A manually optimized native CUDA version, however, is able to gain still more performance. Mainly because data alignment is optimized carefully and shared memory is used as cache. Our implementation would require low-level CUDA optimizations to reach this kind of performance.

Let us consider two huge matrices (750MB and 1.14GB total). In Table 2 the `tile` clause is used in the CUDA versions. The benchmark is run once with 1GB and once with 500 MB available GPU memory.

The pure Java version with 7 threads performs poorly (the sequential version even fails to run in our CPU-quota on our machine). On a GPU with 1 GB of memory the 1.14GB tiled version needs 10 tiles. This version delivers good speedup over the plain threaded version. If we artificially reduce the amount of available GPU memory to 500 MB, 23 tiles are needed.

5.2 Benchmarks

BlackScholes [9] is a model that predicts the evolution of an option price by solving a partial differential equation. A single parallel `for` loop calculates the option prices.

The BlackScholes application uses a single 1D array of floats with 4,000,000 elements. We then run 512 iterations, the results are shown in Table 3. The performance table below (left) shows that the threaded version with 7 threads wins over the naive CUDA implementation because the array is copied many times to and from the GPU. Again the array packages that copy the data only at program start to the GPU, yield the best performance.

2D Convolution is a straightforward convolution. In complex number space an $M \times M$ kernel is applied to an $N \times N$ data set. The outer loop of size $N \times N$ is parallelized. In each iteration one value of the result set is calculated by applying the kernel to the input data set. We use a 4096×4096 matrix with a 16×16 kernel.

Again, we see that the copy overhead to and from the GPU is large causing the threaded version to outperform the naive CUDA version (Table 3, Column 2). With an array-package added, CUDA use again shows a small speedup over the threaded version.

LBM [10] uses cellular automata to numerically simulate almost incompressible fluids at speeds of low Mach numbers. Space and time are discretized and normalized. The benchmark computes 50 time steps over the 3D grid and operates on a 3D domain divided into $N \times N \times N$ cells that hold a finite number of states (D3Q19 model). In one time step the complete set of states is updated synchronously by deterministic, uniform update rules. For parallelization, we use the `parallel for` construct for the time-stepping loop. The loop over the x-axis of the 3D grid is augmented with the `for` directive (default scheduling). For a 64×128×256×19 grid and 100 iterations, we get the results in Table 3, Column 3.

Because the working set is far bigger than the CPU caches, speedup of the plain threaded version is not that high (3.2 on 7 cores). With CUDA enabled, performance plummets as the overhead of serializing 4,194,304 small arrays and then copying the result to and from the GPU 100 times (once per iteration) dominates. Using any array package to keep the data on the GPU completely removes this overhead.

Mersenne Twister [11] has two parts: (1) a pseudo-random number generator and (2) a transformation of the random number sequence into a normal distribution. Both parts process 210,000,000 random numbers divided, into 4096 key rings. In both cases the loop over the key rings is parallelized. The results are shown in Table 4.

For both parts, the threaded version creates good speedups (5.5 and 6.2 resp.), whereas the CUDA versions are slow because of the high compilation and copying overheads. When using an array package, these costs are largely gone and CUDA shows good performance.

If $1,5 * 10^9$ random numbers are generated (see Table 5), tiling is necessary on CUDA. In the first phase of the application, the compute times of both CUDA and 7 Java threads are nearly equal. Due to the higher overheads, the tiled version does not achieve better performance. This is different in the second phase in which the compute time of the CUDA tiled version is lower than the time of the multi-threaded Java version.

6. RELATED WORK

The current prototype does not generate CUDA code for the following OpenMP constructs: `sections`, `single`, `master`, `critical`, `atomic`, `ordered`, `flush` and `barrier`. OpenMP/C [6] makes inroads into supporting these features. In [6] compile-time loop transformations such as parallel loop-swap are used to increase performance. In this paper, we focus on how to manage dynamic translation and problems specific to Java such as solving type-rigidity by using array packages. Their work is therefore orthogonal to ours.

There are numerous ways to use CUDA from Java. The simplest way is to use Java's native interface to call CUDA functions in C. For example, see [12]. Another way is to hide native GPU computing in libraries, for example, by creating a CUDA-BLAS binding [13] for Java. The libraries then interact with the graphics hardware via JNI calls. The disadvantage of such approaches is of course the relative inflexibility of the libraries. With our dynamic code generation, application specific code can be written and executed at full native speed. Our approach adapts to whatever GPU hardware is available on-the-fly and even works (and achieve speedups on a multi-core host) if no GPU is available. In the absence of a GPU, CUDA code could still be generated if a driver were provided to mimic CUDA's functionality on the host proces-

Table 1: Matrix multiplication.

	2048×2048			4096×4096		
	Wall	Compile Overh.	Mngmt. Overh.	Wall	Compile Overh.	Mngmt. Overh.
Java, 1 thread	105.0	-	-	855.0	-	-
Java, 7 threads	15.5	-	-	254.0	-	-
CUDA	5.2	3.7	0.1	17.4	3.6	0.5
Async Comp.	5.0	3.3	0.1	16.5	2.7	0.5
APkg	2.9	1.3	0.0	14.6	1.3	0.0
APkgRect	1.9	1.3	0.0	5.8	1.3	0.0
NVIDIA-manual	0.2	0.0	0.2	0.8	0.0	0.8

Table 2: Large Matrix multiplication.

	7500×7500			10000×10000		
	Wall	Compile Overh.	Mngmt. Overh.	Wall	Compile Overh.	Mngmt. Overh.
Java, 7 threads	1019.0	-	-	2776.5	-	-
CUDA tiled, 1000 MB	91.1	3.6	6.4	759.7	3.8	21.0
CUDA tiled, 500 MB	130.0	3.5	26.4	778.3	3.8	31.9

Table 3: BlackScholes, Convolution and LBM measurements.

	BlackScholes			Convolution			LBM		
	Wall	Compile Overh.	Mngmt. Overh.	Wall	Compile Overh.	Mngmt. Overh.	Wall	Compile Overh.	Mngmt. Overh.
Java, 1 thread	608.0	-	-	43.2	-	-	53.9	-	-
Java, 7 threads	87.0	-	-	6.7	-	-	16.7	-	-
CUDA	111.4	3.8	102.4	20.8	3.9	16.8	334.3	3.6	312.8
Async Comp.	113.8	3.1	107.5	17.1	0.0	15.5	347.0	0.0	329.4
APkg	7.3	1.3	0.0	4.1	1.5	0.0	12.0	1.6	0.6
APkgRect	7.5	1.3	0.0	4.0	1.5	0.0	10.8	1.6	0.6

Table 4: Mersenne measurements, small inputs.

	Mersenne, Part 1, small			Mersenne, Part 2, small		
	Wall	Compile Overh.	Mngmt. Overh.	Wall	Compile Overh.	Mngmt. Overh.
Java, 1 thread	27.6	-	-	45.3	-	-
Java, 7 threads	5.0	-	-	7.3	-	-
CUDA	6.7	2.5	2.6	4.1	1.4	2.6
Async Comp.	5.4	2.0	2.4	2.4	0.0	2.4
APkg	1.9	1.4	0.0	1.5	1.4	0.0
APkgRect	2.0	1.4	0.0	1.5	1.4	0.0

Table 5: Mersenne measurements, large input set.

	Mersenne, Part 1, $1, 5 * 10^9$			Mersenne, Part 2, $1, 5 * 10^9$		
	Wall	Compile Overh.	Mngmt. Overh.	Wall	Compile Overh.	Mngmt. Overh.
Java, 1 thread	195.4	-	-	327.8	-	-
Java, 7 threads	34.7	-	-	53.2	-	-
CUDA, tile	71.1	4.0	17.7	43.3	1.8	17.1

sor. One such an (optimizing) driver is MCUDA [14]. However, performance of such a solution, would depend on the efficiency of the underlying MCUDA.

The code of a computational kernel can also be assembled on-the-fly by building a Abstract-Syntax-Tree manually in the program and compiling that on-the-fly. For example, libSh [15] does so using C++. The created AST can then be used to generate CUDA code or normal threaded code. Our approach differs in that we do not build ASTs but rather, at pre-run-time, analyze and translate annotated bytecodes.

CuPP [16] creates data structures that can have different data representations on hosts and on GPUs. Our array packages do the same, and the CuPP approach could be used to partially implement our array packages. CUDA-lite [17] is a CUDA code optimizer. It rewrites loops and allocates data in the memory region most suitable for the accesses involved. Our CUDA class loader could use CUDA-lite for post-optimization.

OpenCL [18] is a standard for data parallelism on GPUs and SPMD processors, e.g., the Cell [3]. It will be straightforward to extend our framework to also generate OpenCL code. Since OpenCL [18] proposes unified standard parallel processing, it does not expose a specific vendor's hardware features that might be the key to a code's performance. By directly targeting a vendor's lower-level API we dynamically avoid abstraction layers.

Our system uses tiling to allow large data structures, where others mostly use tiling for parallelization and/or locality optimizations [19]. HTA [20] is a library that encapsulates arrays. Our array packages could conceptually use HTA for managing non-GPU allocated arrays.

7. CONCLUSIONS AND FUTURE WORK

We observe that using the Java class loader to dynamically adapt a parallel loop to exploit available accelerators is feasible and leads to an excellent application speed-up. However, when naively generating CUDA code from Java/OpenMP, we have observed three obstacles. First, the significant overhead of invoking the CUDA compiler at run-time that can only be amortized with large compute times. Using fast JIT-style compilation of CUDA kernels would help to solve this. Second, there are high overheads for copying data to and from the GPU. The presented array packages encapsulate locality information provide a solution. Third, the small GPU memory that we hide with the tiling annotation.

At the moment our runtime is only capable of using either the CPU or one GPU. In the future it should be possible to share work between the CPU and GPU and to use more than one GPU at the same time. Another avenue for future work is load-balancing of data and code across a cluster of workstations, each equipped with OpenCL/CUDA-capable hardware.

8. REFERENCES

[1] Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable Parallel Programming with CUDA. Queue **6**(2) (2008) 40–53

[2] Klemm, M., Bezold, M., Veldema, R., Philippsen, M.: JaMP: An Implementation of OpenMP for a Java DSM. Concurrency and Computation: Practice and Experience **18**(19) (2007) 2333–2352

[3] Scarpino, M.: Programming the Cell Processor: For Games, Graphics, and Computation. Prentice Hall PTR, Upper Saddle River, NJ (2008)

[4] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: stream computing on graphics hardware. In: SIGGRAPH '04, Los Angeles, CA (2004) 777–786

[5] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Dubey, P., Junkins, S., Lake, A., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Abrash, M., Sugerman, J., Hanrahan, P.: Larrabee: A Many-Core x86 Architecture for Visual Computing. IEEE Micro **29**(1) (2009) 10–21

[6] Lee, S., Min, S.J., Eigenmann, R.: OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In: Symp. on Principles and Practice of Parallel Programming, Raleigh, NC (2008) 101–110

[7] Lin, Y., Terboven, C., an Mey, D., Copty, N.: Automatic scoping of variables in parallel regions of an openmp program. In Chapman, B.M., ed.: WOMPAT. Volume 3349 of Lecture Notes in Computer Science., Springer (2004) 83–97

[8] Midkiff, S., Moreira, J., Snir, M.: Java For Numerically Intensive Computing: From Flops To Gigaflops. In: Symp. on the Frontiers of Massively Parallel Computation, Annapolis, MA (1999) 251–261

[9] Black, F., Scholes, M.: The pricing of options and corporate liabilities. Journal of Political Economy **81**(3) (1973) 637–54

[10] Wolf-Gladrow, D.: Lattice-Gas Cellular Automata and Lattice Boltzmann Models. Number 1725 in Lecture Notes in Mathematics. Springer (2000)

[11] Matsumoto, M., Nishimura, T.: Mersenne Twister: a 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. ACM Trans. Model. Comput. Simul. **8**(1) (1998) 3–30

[12] JCuda. http://www.jcuda.org/

[13] Barrachina, S., Castillo, M., Igual, F., Mayo, R., Quintana-Orti, E.: Evaluation and tuning of the Level 3 CUBLAS for graphics processors. In: Intl. Parallel and Distributed Processing Symp., Miami, FL (2008) 1–8

[14] Stratton., J., Stone., S., Hwu, W.M.W.: MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, Edmonton, Canada (2008) 16–30

[15] cCool, M., Toit, S.D.: Metaprogramming GPUs with Sh. AK Peters Ltd (2004)

[16] Breitbart, J.: CuPP – A framework for easy CUDA integration. In: HIPS: High-Level Parallel Programming Models and Supportive Environments, Rome, Italy (2009) 1–8

[17] Ueng, S.Z., Lathara, M., Baghsorkhi, S., Hwu, W.M.W.: CUDA-Lite: Reducing GPU Programming Complexity. In: Languages and Compilers for Parallel Computing, Edmonton, Canada (2008) 1–15

[18] Khronos. http://www.khronos.org/opencl/

[19] Wolfe, M.: More iteration space tiling. In: Proc. of the 1989 ACM/IEEE conference on Supercomputing, Reno, Nevada (1989) 655–664

[20] Guo, J., Bikshandi, G., Fraguela, B.B., Garzaran, M.J., Padua, D.: Programming with Tiles. In: PPoPP '08: Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, Salt Lake City, UT (2008) 111–122