Speculative Execution of Parallel Programs with Precise Exception Semantics on GPUs

Akihiro Hayashi, Max Grossman, Jisheng Zhao, Jun Shirako, and Vivek Sarkar

Department of Computer Science, Rice University, Houston, TX, USA,
{ahayashi,jmg3,jisheng.zhao,shirako,vsarkar}@rice.edu

Abstract. General purpose computing on GPUs (GPGPU) can enable significant performance and energy improvements for certain classes of applications. However, current GPGPU programming models, such as CUDA and OpenCL, are only accessible by systems experts through low-level C/C++ APIs. In contrast, large numbers of programmers use high-level languages, such as Java, due to their productivity advantages of type safety, managed runtimes and precise exception semantics. Current approaches to enabling GPGPU computing in Java and other managed languages involve low-level interfaces to native code that compromise the semantic guarantees of managed languages, and are not readily accessible to mainstream programmers.

In this paper, we propose compile-time and runtime technique for accelerating Java programs with automatic generation of OpenCL while preserving precise exception semantics. Our approach includes (1) automatic generation of OpenCL kernels and JNI glue code from a Java-based parallel-loop construct (`forall`), (2) speculative execution of OpenCL kernels on GPUs, and (3) automatic generation of optimized and parallel exception-checking code for execution on the CPU. A key insight in supporting our speculative execution is that the GPU's device memory is separate from the CPU's main memory, so that, in the case of a misspeculation (exception), any side effects in a GPU kernel can be ignored by simply not communicating results back to the CPU.

We demonstrate the efficiency of our approach using eight Java benchmarks on two GPU-equipped platforms. Experimental results show that our approach can significantly accelerate certain classes of Java programs on GPUs while maintaining precise exception semantics.

1 Introduction

Programming models for general-purpose computing on GPUs (GPGPU), such as CUDA and OpenCL, can enable significant performance and energy improvements for certain classes of applications. However, these programming models provide system experts with low-level C/C++ APIs and require programmers to write, maintain, and optimize a non-trivial amount of application code.

In constrast, large numbers of programmers use high-level languages, such as Java, because these languages provide high-productivity features including type safety, a managed runtime, and precise exception semantics. However, the

performance of an application can often suffer due to runtime overheads caused by the additional logic required to enforce these guarantees. In addition, using heterogeneous systems to accelerate applications in these high-level languages is a difficult and error-prone task. Accessing OpenCL or CUDA's C/C++ API from Java requires the use of the Java Native Interface (JNI) API, immediately removing many of the programmability benefits of Java software development.

In our recent work [6], we introduced Habanero-Java [3] with OpenCL generation (HJ-OpenCL), an extension to the parallel HJ programming language. HJ-OpenCL enables execution of parallel `forall` loops on any heterogeneous processor in an OpenCL platform without any code change to the original HJ source. However, this approach requires programmers to use a `safe` language construct to explicitly specify conditions which are required to preserve Java exception semantics. With the `safe` construct, the programmer provides a boolean condition that ensures a parallel loop is not expected to throw an exception and can be safely executed outside of the JVM. However, the use of `safe` construct requires additional development effort. The runtime overhead of manual exception checking is not negligible when running applications which have indirect array access and non-affine array access.

In this work, we propose extensions to the compile-time and runtime techniques introduced in HJ-OpenCL which preserve precise exception semantics when executing a parallel `forall` loop outside the JVM. Unlike our previous work, the compiler automatically translates a `forall` loop into two parallel routines. The first routine contains an equivalent OpenCL implementation of the original `forall` loop, including all initialization, communication, and computation code required to transfer execution to an OpenCL device. The second routine is a transformation and subset of the instructions in the original forall loop which guarantees any runtime exception thrown by the original loop will also be thrown by the transformed version. If an exception occurs during execution of this specialized exception-checking code, execution transfers to a JVM-only implementation of the parallel loop. The runtime speculatively executes the specialized-checking code and the full OpenCL implementation in parallel to reduce the overhead of exception checking.

This paper makes the following contributions:

1. Automatic generation of OpenCL code from Habanero-Java for speculative execution on GPUs
2. Automatic generation of optimized and parallel exception-checking code for execution on the multiple CPU cores.
3. Performance evaluation of the proposed scheme on multiple heterogeneous platforms with CPU and GPU cores.

2 Motivation

While past evaluation of GPUs on extremely parallel and computationally heavy applications have demonstrated clear performance benefits for appropriate applications [18][22], there still remain application domains which could make use

of GPUs but do not. This missed opportunity is primarily caused by the sub-par programmability offered by existing GPU programming models: CUDA and OpenCL. In addition, these programming models are still only accessible from low-level programming languages, out of the scope of most high-level programmers' experience. To make the performance benefits of GPUs available to a wide range of developers it is necessary to build interfaces which are similar to and compatible with the managed languages in widespread use today. Arguably, the most pervasive example of this category of programming languages is Java.

Today, Java programmers can manually utilize GPUs using CUDA and OpenCL through JNI. However, native, OpenCL, or CUDA execution through JNI eliminates one of the primary safety benefits of the Java development environment: exceptions. Java's precise exception semantics provide a Java programmer with safety guarantees in regards to the correct execution of their application code. For example, these guarantees include checks for null pointer references, out-of-bounds array accesses, and division-by-zero. On the other hand, many natively compiled language provide no guarantees that a reference does not jump into a completely separate array, or object. As a result, incorrect application behavior can be difficult to diagnose in the absence of precise exception semantics.

As an illustrative example we consider the case of sparse matrix-matrix multiplication. Executing this computation the JVM would ensure that the row and column indices stored to represent a sparse matrix are within the bounds of a full output matrix. However, if the Java programmer took advantage of JNI to achieve improved performance through native execution all exception semantics would be forfeit. To maintain the same guarantees, the programmer would have to manually insert exception checking code in their Java or native code which checked the stored row and column indices against the bounds of the output matrix before submitting the kernel to the GPU. Doing so would increase code complexity and future maintainability.

This work addresses the problem of melding the performance characteristics of native GPU execution with the safety guarantees of JVM execution. It does so by enabling execution of a parallel Java application on any OpenCL hardware platform without any hand-written native code. This approach removes the pain points of JVM-OpenCL applications while providing the benefits of both managed and native execution.

3 Habanero Java Language

This section describes features of the Habanero-Java (HJ) parallel programming language and compilation flow for supporting OpenCL code generation.

3.1 Overview of HJ language

The Habanero Java (HJ) parallel programming language under development at Rice University [3] provides an execution model for multicore processors that builds on four orthogonal constructs, and was derived from early experiences with the X10 [5] language:

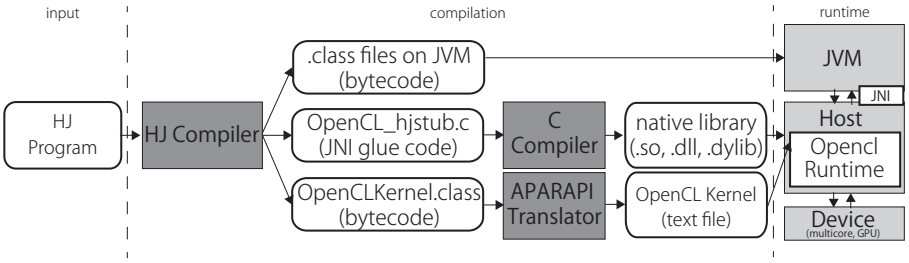

Fig. 1: Compilation and Runtime Flow

1. Lightweight *dynamic task creation and termination* using *async* and *finish*, *future* and *forall* constructs [19].
2. *Locality control* with task and data distributions using the *place* construct [15].
3. Mutual exclusion and isolation among tasks using the *isolated* construct [21].
4. Collective and point-to-point synchronization using the *phasers* construct [10] along with their accompanying *phaser accumulators* [11].

In HJ-OpenCL, programmers use the `forall` language feature to identify parallel loops as candidates for OpenCL execution. The statement "`forall(point p :` *region*`)` ⟨*stmt*⟩" indicates a parallel loop whose iteration space is defined by a *region*. The region can be one- or multi-dimensional space, e.g., `[0:M-1,0:N-1]` for a 2-D iteration space. Each iteration instance executes the loop body ⟨*stmt*⟩ for a distinct *point* in the iteration space. All `forall` loops end with an implicit barrier. In addition, HJ-OpenCL supports all-to-all synchronization points in those parallel loops [6] through the *n*ext statement. The HJ-OpenCL compiler and runtime trust these annotations when generating and executing code on GPUs.

3.2 Compilation Flow

Fig. 1 illustrates the HJ compilation and runtime flow for HJ-OpenCL. The HJ-OpenCL compiler leverages APARAPI [1] , a comprehensive, open-source framework for executing computational kernels from Java applications on OpenCL devices. For this work we extended the APARAPI component that generates OpenCL code from Java bytecode. In addition to OpenCL kernels, glue code must be automatically generated to transfer execution and data from the JVM to the OpenCL device and back. This functionality is provided internally by the HJ-OpenCL compiler, and includes the generation of JNI functions, OpenCL API calls, and transformed bytecode.

In summary, the HJ-OpenCL compiler takes an HJ program as input, and produces:

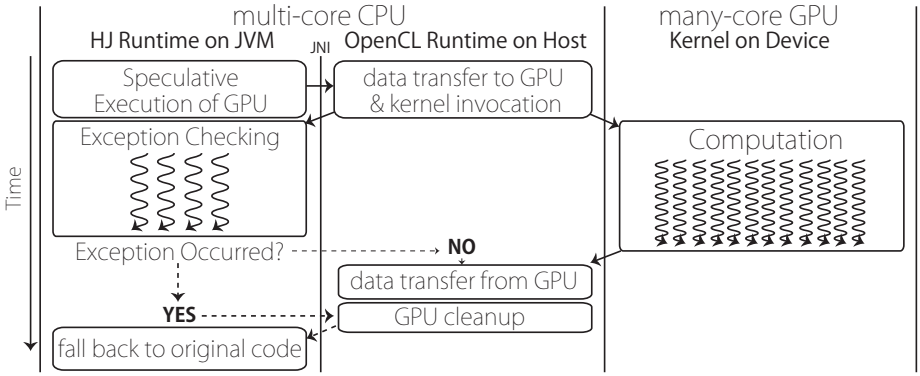

Fig. 2: The execution model of speculative exception checking

1. Java CLASS files for execution on the JVM;
2. JNI glue code to mediate between the JVM and OpenCL kernels;
3. A Java CLASS file which contains the bytecode to be translated to OpenCL kernels by the APARAPI bytecode translator.

4 Speculative Exception Checking Scheme

In the approach introduced in this paper, the HJ-OpenCL compiler from [6] is extended to automatically generate exception checking code and OpenCL kernel code for the `forall` loops in an HJ program. The exception checking code is a specialized version of the original `forall` loop which replaces all stores with loads. By enclosing this specialized loop with a Java *try-catch* block, HJ-OpenCL can detect all runtime exceptions generated by the original `forall` loop. The details of the code transformation algorithm for generating this exception loop is shown in Section 4.2. The exception checking loop is run in parallel on CPU cores in parallel with a speculatively and optimistically launched OpenCL kernel running on the GPU. Optimizing the exception checking code is important for cases where it is on the critical path of the application, i.e. where the exception checking code in the JVM finishes later than the OpenCL kernel.

The rest of this Section is organized as follows: Section 4.1 introduces the HJ-OpenCL runtime design. Section 4.2 describes optimizations applied to the exception checking loop by the HJ-OpenCL compiler.

4.1 Speculative Exception Checking Runtime

Fig. 2 illustrates the runtime interactions between the HJ runtime, the JVM, the OpenCL runtime on the host, and the OpenCL kernel on the device. The following steps (illustrated by the example generated code in Fig. 3) explain

the basic workflow of speculatively executing a `forall` loop on the GPU while running exception checking code in the JVM.

Step 1: The HJ runtime invokes the first JNI function. In the callee, the OpenCL API is called to perform host-to-device data transfer and asynchronously launch the corresponding OpenCL kernel on a device. The host application immediately returns to JVM execution. Blocking data transfers are necessary so that Java objects can be released before returning to the JVM. This step is done in the native method call to *openCL_Kernel0_first()* on line 9 of Fig. 3.

Step 2: The exception checking loop is run in the JVM, as seen on lines, seen on lines 12-16 of Fig. 3. This loop is a transformation of the original `forall` loop which:

1. Reduces computational load.
2. Reduces I/O load and eliminates any externally visible state change caused by the loop
3. Guarantees the same exceptions thrown by the original loop would also be thrown by the transformed version.

Step 3: The HJ runtime invokes a second JNI call (line 21 of Fig. 3 which waits for the completion of computation on the OpenCL device[1], transfers data from the device, and performs OpenCL cleanup. If an exception occurs during execution of the exception checking loop, the OpenCL runtime does not transfer any state back to the host and the HJ runtime executes the original `forall` loop in the JVM, thus maintaining Java exception semantics.

4.2 Generation and Optimization of Exception Checking Code

This section describes how to generate and optimize the exception checking code. The generated exception checking code must meet two requirements. It must be side-effect free, i.e. have no memory store operations and no invocation of system APIs. It must also preserve all exceptions which would be triggered from the original `forall` loop. If either of these requirements cannot be met, the HJ-OpenCL compiler aborts code generation and reverts to parallel execution within the JVM.

The basic workflow of the generation and optimization of exception checking code is described in Algorithm 1. It takes the original `forall` loop (L) as input and generates the optimized exception checking code (OCC) as output. Before applying Algorithm 1, some analysis and transformations are performed to verify the correctness of running the `forall` loop with speculative exception checking:

1. Side-effect analysis to identify procedures which potentially have side-effects.

[1] The OpenCL runtime has to wait for the completion of the kernel execution even in the event of an exception because there is no OpenCL API to terminate kernel on device currently.

```
1  public class Example {
2    static { System.loadLibrary(``libCalc''); }
3    public static native void openCL_Kernel0_first(...);
4    public static native void openCL_Kernel0_second(...);
5    public static void main(String[] args) {
6    ...
7    boolean excpFlag = false;
8    /* (1) Speculative GPU execution through JNI */
9    openCL_Kernel0_first(...);
10   /* (2) Exception Checking Code on JVM */
11   try {
12     forall (point [i]:[0:N-1]) {
13       int dummy1 = A[i];
14       int dummy2 = B[i];
15       int dummy3 = C[i];
16     }
17   } catch (Exception e) {
18     excpFlag = true;
19   }
20   /* (1) Second JNI Call */
21   openCL_Kernel0_second(excpFlag, ...);
22   if (excpFlag) {
23     /* (3) Original Implementaion */
24     forall (point [i]:[0:N-1]) {
25       A[i] = B[i] + C[i];
26     }
27   }
28 }
```

Fig. 3: Generated code for Vector Addition by the HJ-OpenCL compiler

2. Function inlining applied for all non-recursive functions invoked within the `forall` loop.
3. Alias analysis which works out may or must equality between any two object references.
4. Data dependence analysis which calculates def-use chains.

After pre-analysis and transformation, if the `forall` loop still contains unanalyzable array or procedures which may have side-effects, then it is not suitable for speculative OpenCL execution and the HJ-OpenCL compiler aborts exception code generation(Line 1).

Algorithm 1 begins by inspecting all array access statements in the `forall` loop to retain any statements which may throw an `ArrayIndexOutOfBoundsException`. For each array store statement *aStore*, the HJ-OpenCL compiler replaces the statment by an array read statement *aLoad* (lines 6-20) and traverses the def/use chain (built by pre-analysis) to check its users. In the case that the stored value is loaded by successors within the same loop iteration, the compiler applies scalar replacement on the load statement with the stored value and marks it as keep (lines 13-15). For the case that the store value is loaded in the successors which cross loop iterations, the HJ-OpenCL the compiler sets *excpFlag* with *true* (line 10-12) and aborts code generation. For the array load statement, the HJ-OpenCL compiler marks it as *keep* (line 24-27). The last step is to mark statements which derive denominator of division statement to keep statement, as they may trigger an `ArithmeticException`.

Algorithm 1: Exception Checking Code Generation and Optimization

```
   input  : L: One Forall Loop
   output: OCC: Optimized Checking Code
 1 if loop has unanalyzable array references or method calls then
 2 |   abort
 3 end
 4 // For Array Bounds Check;
 5 A ← getAllArrayAccessStatement(loop);
 6 foreach aStmt in A do
 7 |   // Get All Loop Index at Current Loop Nest;
 8 |   I ← getOuterLoopIndices();
 9 |   if aStmt is ArrayStore(A[i_1, i_2, ..., i_n] ← x) then
10 |   |   transform aStmt to dummy ← A[i_1, i_2, ..., i_n];
11 |   |   markedList ← aStmt;
12 |   |   if A[i_1, i_2, ..., i_n] is used in followed statements then
13 |   |   |   if A[i_1, i_2, ..., i_n] drives array subscript in the future iteration then
14 |   |   |   |   abort
15 |   |   |   end
16 |   |   |   else
17 |   |   |   |   rename A[i_1, i_2, ..., i_n] to x in each statement (as in scalar replacement);
18 |   |   |   end
19 |   |   end
20 |   |   foreach i_p such that 1 ≤ p ≤ n ∧ i_p ∉ I do
21 |   |   |   S ← statements which derive i_p (considering control flow);
22 |   |   |   markedList ← S ;
23 |   |   end
24 |   end
25 |   else if aStmt is ArrayLoad (x ← A[i_1, i_2, ..., i_n]) then
26 |   |   markedList ← aStmt ;
27 |   |   foreach i_p such that 1 ≤ p ≤ n ∧ i_p ∉ I do
28 |   |   |   S ← statements which derive i_p (considering control flow);
29 |   |   |   markedList ← S;
30 |   |   end
31 |   end
32 end
33 // For ArithmeticException;
34 markedList ← ∀stmt such that stmt derives denominator;
35 // Delete not marked statement;
36 OCC ← ∀stmt in L such that stmt ∈ markedList;
```

After applying Algorithm 1 to generate conservative exception checking code, HJ-OpenCL compiler performs two optimizatioin on the generated code to eliminate redundancy: loop invariant code motion (LICM) and redundant load elimination.

Fig. 4 provides an example of HJ-OpenCL code generation and optimzation. Fig. 4 (a) contains the `forall` loop of a sparse matrix multiply application in 3-address code. Fig. 4 (b) shows the same 3-address code, but optimized by the HJ-OpenCL compiler for exception checking. Because there are indirect accesses of arrays *row*, *Av*, *Aj* and *x*, the HJ-OpenCL compiler does not remove statements which derive array subscripts of these arrays.

5 Performance Evaluation

This section presents experimental results for HJ-OpenCL on two platforms.

The first platform is an AMD *A10-5800K* APU. This APU includes an AMD Radeon HD 7660D GPU with 6 Streaming Multiprocessors(SMs). The CPU of

```
1  forall (point [id]:[0:M]){
2    i1 = row[id];
3    row_begin = i1
4    i2 = id + 1;
5    i3 = row[i2];
6    row_end = i3;
7    i4 = row_end - row_begin;
8    for (i = 0; i < i4; i++) {
9      for (j = 0; j < inter; j++)
           {
10       i5 = row_begin + i;
11       d1 = Av[i5];
12       i6 = row_begin + i;
13       i7 = Aj[i6];
14       d2 = x[i7];
15       d3 = d1 * d2;
16       d4 = sum + d3;
17       sum = d4;
18     }
19   }
20   y[id] = sum;
21 }
```

(a) Original 3-address Code

```
1  forall (point [id]:[0:M]){
2    i1 = row[id];
3    row_begin = i1
4    i2 = id + 1;
5    i3 = row[i2];
6    row_end = i3;
7    i4 = row_end - row_begin;
8    for (i = 0; i < i4; i++) {
9      i5 = row_begin + i;
10     d1 = Av[i5];
11     i6 = row_begin + i;
12     i7 = Aj[i6];
13     d2 = x[i7];
14   }
15   dummy = y[id];
16 }
```

(b) Optimized 3-address Code

Fig. 4: Optimization Example for Sparse Matrix Multiply

the *A10-5800K* includes 4 cores, 16KB of L1 cache per core, and 32MB of L2 cache. Each SM in the GPU has exclusive access to 32 KB of local scratchpad memory. The CPU and GPU can each access the same system memory, but share bandwidth when doing so. While physical memory is shared, it is partitioned between devices such that the CPU has 6GB and the GPU has 2GB. We conducted all experiments on this system using the Java SE Runtime Environment (build 1.6.0_21-b06) with Java HotSpot 64-Bit Server VM (build 17.0-b16, mixed mode).

The second platform has two hexacore Intel X5660 CPUs and two NVIDIA Tesla M2050 discrete GPUs connected over PCIe. There is a total of 48GB within a single node that is shared by all 12 cores. Each GPU also has approximately 2.5GB of global memory. Only 1 of the 2 available GPUs was used at a time to evaluate this work. In this platform, we used the Java SE Runtime Environment (build 1.6.0_25-b06) with Java HotSpot 64-Bit Server VM (build 20.0-b11, mixed mode).

The eight benchmarks shown in Table 1 were used in our experiments. Note that SparseMatMult, SAXPY and GEMVER have indirect array access. The baseline for this evaluation was sequential Java. We tested execution on OpenCL GPUs using HJ-OpenCL's code generation and runtime in the following modes:

- **No checking**: execute the full computation on the GPU without any exception checking, removing precise Java exception semantics.
- **Non-speculative execution**: run the unoptimized or optimized exception checking code in the JVM, followed by the full computation on the GPU.

Benckmark	Summary	Data Size
SparseMatmult	Sparse matrix multiplication from the Java Grande Benchmarks [20]	Size C with N = 500,000
Doitgen	Multi-resolution analysis kernel from PolyBench[24], ported to Java	128×128×128
Crypt	Cryptographic application from the Java Grande Benchmarks[20]	Size C with N= 50,000,000
Blackscholes	Data-parallel financial application which calculates the price of European put and call options	16,777,216 virtual options
MRIQ	Three-dimensional medical benchmark from Parboil[23], ported to Java	large size(64×64×64)
MatMult	A standard dense matrix multiplication: $C = A.B$	1024×1024
SAXPY	Sparse version of SAXPY from [12], ported to Java	25,000×25,000
GEMVER	Sparse BLAS function from [12], ported to Java	10,000,000

Table 1: Information on the benchmarks used to evaluate HJ-OpenCL

This mode retains precise Java exception semantics but serializes exception checking and computation, leading to higher overhead.

– **Speculative execution**: run the unoptimized or optimized exception checking code in the JVM in parallel with the full computation on the GPU. This mode retains precise Java exception semantics while minimizing overhead.

In the following sections, these five variants are referred to as HJ OpenCL GPU(No checking), HJ OpenCL GPU(Non-speculative, unoptimized), HJ OpenCL GPU(Non-speculative, optimized), HJ OpenCL GPU(Speculative, unoptimized) and HJ OpenCL GPU(Speculative, optimized) respectively. We run each benchmark 10 times and report the median value as the result. Note that we exclude the overhead of the OpenCL context and command queue creation from these measurements for precise measurements because we see timing in variation.

5.1 Performance on AMD A10-5800K

Fig 5 shows the speedup numbers on the AMD *A10-5800K* APU relative to the sequential Java version. On the AMD APU system, exception checking is done in parallel on 4 cores. OpenCL(Speculative, optimized) approach shows speedups of up to 21.1× relative to sequential Java, while maintaining Java exception semantics. Only one benchmark (Polybench.doitGen) showed a slowdown due to OpenCL execution on this platform, though (as we will see later) it showed a speedup on the Westmere+Tesla platform. Performance differences between OpenCL(No Checking) and OpenCL(Speculative, optimized) range from 0.5%(Polybench.Doitgen) to 18.6%(JGF-Crypt). JGF-Crypt, BlackScholes, MRIQ and GEMVER each show significant improvement from exception checking code optimization. For these applications, exception checking takes longer than OpenCL execution. Additionally, deleting java.lang.Math method calls which does not derive array index dramatically accelerates exception checking code for BlackScholes and MRIQ.

Polybench Doitgen, MatMult, and SAXPY exception checking code, these benchmarks do not show speedup from optimization because the checking code is not on the critical path.

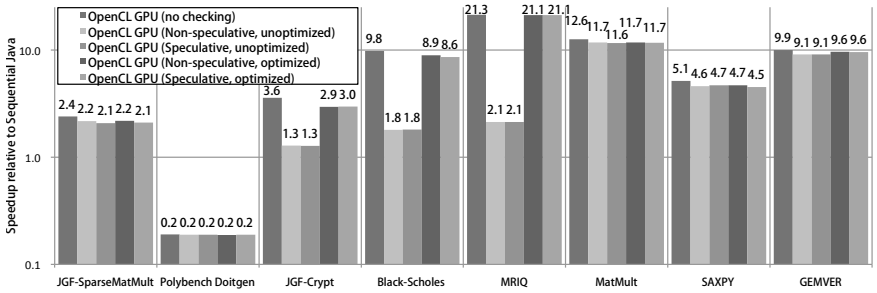

Fig. 5: Performance improvements relative to sequential Java on the *A10-5800K*

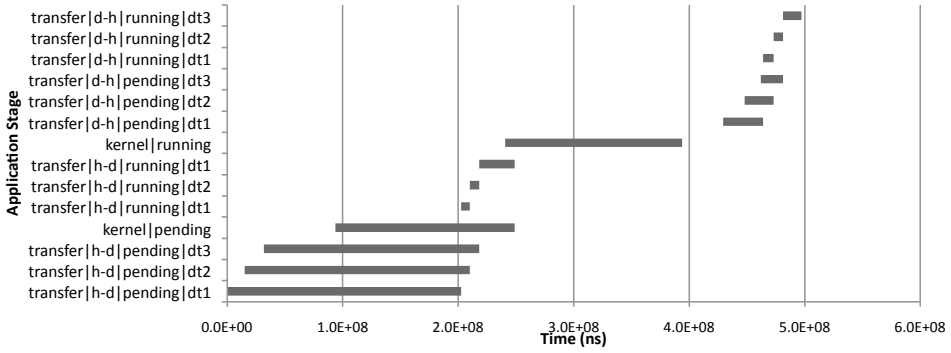

Fig. 6: Sample timeline of the Black-Scholes application on the *A10-5800K*

Fig. 6 shows a timeline of OpenCL exection on the AMD APU. Fig. 6 was gathered using the OpenCL *c*lGetEventProfilingInfo function to get information on when commands are submitted to the device for execution, when commands actually begin execution, and when commands complete execution. Each row in the figure is categorized as either a pending operation, which shows the time between a command being submitted and starting, or a running operation, which shows the time between a command starting and finishing. Each operation is also categorized as either a kernel or transfer operation, and transfer operations are broken down by the variable being transferred the direction of transfer. *h-d* indicates a copy from the host system to the GPU, and *d-h* indicates a copy from the GPU to the host. On the AMD APU, pending time accounts for a significant amount of execution time for both transfers and kernels. That is why no application shows speedup with speculative execution unlike NVIDIA GPU. For example, the AMD OpenCL runtime does not start three data transfers for the first 2.0E+08 (ns). This seems to be an issue with the AMD OpenCL libraries.

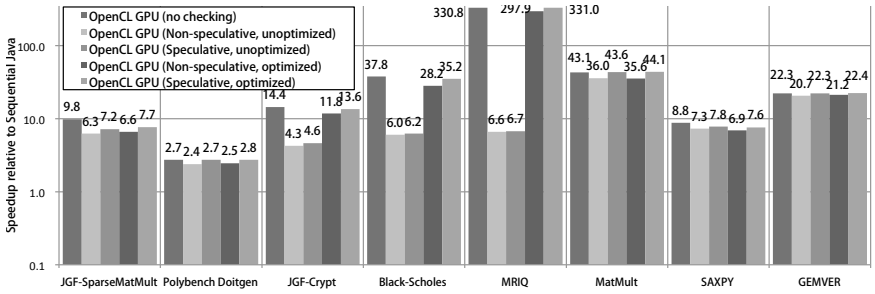

Fig. 7: Performance improvements relative to sequential Java on *Westmere*

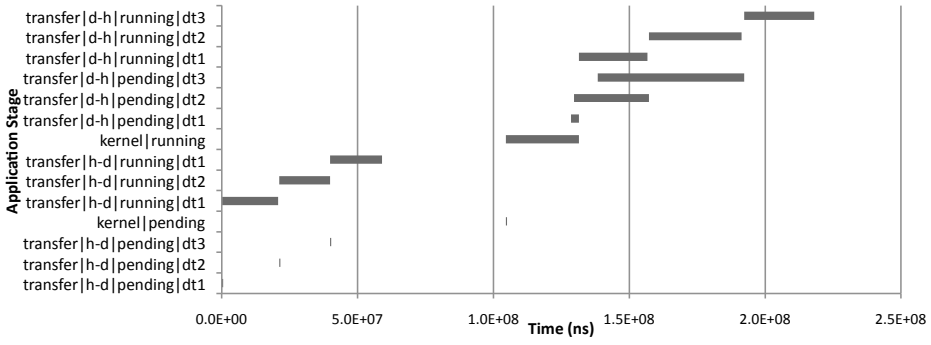

Fig. 8: Sample timeline of the Black-Scholes application on *Westmere*

Despite this, we see a gap between the completion of the *k*ernel—running operation and the start of *t*ransfer—d-h—pending—dt1 operation. This indicates that the critical path on the AMD APU for Black-Scholes is the exception checking code, which explains the significant improvements from optimized exception checking in Fig. 5.

5.2 Performance on Westmere

Fig. 7 shows the speedups on the *Westmere* platform with two NVIDIA Tesla GPUs (of which we currently only use one) and two hexacore Intel CPUs.

On this platform, exception checking is done by 12 cores. The proposed OpenCL(Speculative, optimized) mode shows speedups of up to 331.0× relative to sequential Java while maintaining Java's exception semantics. Performance differences between OpenCL(No Checking) and OpenCL(Speculative, op-

timized) vary from -2.4%(MatMult)[2] to 27.9%(JGF-Crypt). JGF-SparseMatMult, Polybench Doitgen, JGF-Crypt, BlackScholes, MRIQ and GEMVER shows speedups from optimization because exception checking is on the application's critical path. Further, there is no difference in time between command submission and actually starting work on it(See Table 8). Additionally, more cores enable a shorter exception checking time. As a result speculative execution enables performance improvement in the range of 2.0% to 24.7%.

6 Related Work

The GPU code generation has been widely supported in high level language compilation systems.

Lime [7] is a JVM compatible language which generates OpenCL code automatically. Lime provides language extensions that express coarse grain tasks, SIMD parallelism. Its compiler generates Java bytecode, JNI glue code, and OpenCL kernels.

RootBeer [13] compiles Java to CUDA by specifying the code region within *gpuMethod*. The RootBeer compiler translates *gpuMethod()* method in *Kernel* interface into CUDA kernel.

JCUDA [17] provides programming interface which can be used by Java programmers to invoke CUDA kernels. Programmers can write Java codes that call CUDA kernels with special interface and JCUDA compiler generates the JNI glue code between the JVM and CUDA runtime by using this interface.

Android RenderScript [4] provides C-like programming model for GPUs. Programmer manually write a kernel and invoke it by using provided Java APIs.

To the best of our knowledge, none of these three systems (Lime, RootBeer, JCUDA, RenderScript) preserve Java's precise exception semantics with speculative execution, as in our work.

There also been related work on eliminating redundant checks for null pointer and array bound exceptions by generating dual version code. In Artigas et al. [2] and Moreira et al. [9], Their work generates dual-version code which consists of exception-safe regions and -unsafe regions. In exception-safe regions the compiler can perform aggressive loop optimization such as loop tiling. In contrast, the automatically generated exception-checking code in our approach that sets excpFlag can express more general conditions than in this past work e.g., see Fig. 3.

There has also been related past work on array bounds check elimination. Würthinger et al. [16] proposed an algorithm for Static Single Assignment(SSA) form for the JIT compiler which eliminates unnecessary bounds checking. ABCD [14] provides powerful array bounds checking elimination algorithm by creating an SSA-based inequality graph. Jeffery et al. [8] proposed an static annotation framework to reduce the overhead of dynamic checking in the JIT compiler. These past results complement our work since the exception checking code generated by our compilation system can be further optimized by these techniques.

[2] Theoretically this is unlikely, This is due to variation in timing.

In the context of speculative execution for parallel processing GPGPU, Paragon [12] runs the C/C++ loop speculatively, while monitoring the dependencies. The runtime transfers the execution to the CPU in case a conflict is detected. In contrast, we generate exception checking code that is executed on the CPU.

7 Conclusions

In this paper, we introduce a new compile-time and runtime approach for accelerating Java programs through automatic generation of OpenCL while maintaining precise exception semantics. To maintain precise exception semantics, the HJ-OpenCL compiler automatically generates code for the speculative execution of OpenCL kernels on GPUs alongside optimized and parallel exception checking code for execution on the CPUs.

On an AMD APU, our results show speedups of up to 21.1× relative to sequential on the integrated GPU, only 0.8% slower than unsafe execution on the GPU. For a system with an Intel Xeon CPU and a discrete NVIDIA Fermi GPU, the speedups relative to sequential Java are up to 331.0× on the GPU, equivalent performance to unsafe execution. These experiments show that our approach can automatically and effectively accelerate the execution of Java programs on GPUs while maintaining precise exception semantics.

References

1. APARAPI. API for Data Parallel Java. http://code.google.com/p/aparapi/.
2. Pedro V. et al. Artigas. Automatic loop transformations and parallelization for java. In *Proceedings of the 14th international conference on Supercomputing*, ICS '00, pages 1–10, New York, NY, USA, 2000. ACM.
3. Vincent Cavé et al. Habanero-Java: the New Adventures of Old X10. In *PPPJ'11: Proceedings of 9th International Conference on the Principles and Practice of Programming in Java*, 2011.
4. Android Developers. Renderscript. http://developer.android.com/guide/topics/renderscript/index.html.
5. Kemal Ebcioğlu, Vijay Saraswat, and Vivek Sarkar. X10: Programming for hierarchical parallelism and nonuniform data access (extended abstract). In *Language Runtimes '04 Workshop: Impact of Next Generation Processor Architectures On Virtual Machines (colocated with OOPSLA 2004)*, October 2004. www.aurorasoft.net/workshops/lar04/lar04home.htm.
6. Akihiro Hayashi et al. Accelerating Habanero-Java Program with OpenCL Generation(under submission). In *PPPJ'13: Proceedings of 10th International Conference on the Principles and Practice of Programming in Java*, 2013.
7. Christophe Dubach et al. Compiling a high-level language for gpus: (via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12, New York, NY, USA, 2012. ACM.
8. Jeffery Von Ronne et al. Safe bounds check annotations. In *Concurrency and Computations: Practice and Experience, Vol. 21, No. 1*, 2009.
9. José E.Moreira et al. From flop to megaflops: Java for technical computing. *ACM Trans. Program. Lang. Syst.*, 22(2):265–295, March 2000.

10. Jun Shirako et al. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 277–288, New York, NY, USA, 2008. ACM.
11. Jun Shirako et al. Phaser accumulators: a new reduction construct for dynamic parallelism, 2009. IPDPS 2009.
12. Mehrzad Samadi et al. Paragon: collaborative speculative loop execution on gpu and cpu. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 64–73, New York, NY, USA, 2012. ACM.
13. P.C Pratt-Szeliga et al. Rootbeer: Seamlessly using gpus from java. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 375–380, June.
14. Rastislav Bodík et al. Abcd: eliminating array bounds checks on demand. *SIGPLAN Not.*, 35(5):321–333, May 2000.
15. Satish Chandra et al. Type inference for locality analysis of distributed data structures. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 11–22, New York, NY, USA, 2008. ACM.
16. Würthinger et al. Array bounds check elimination for the java hotspot client compiler. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, PPPJ '07, pages 125–133, New York, NY, USA, 2007. ACM.
17. Yonghong Yan et al. Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 887–899, Berlin, Heidelberg, 2009. Springer-Verlag.
18. Zhe Fan et al. GPU Cluster for High Performance Computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, page 47, Washington, DC, USA, 2004. IEEE Computer Society.
19. Yi Guo et al. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *IPDPS '09: International Parallel and Distributed Processing Symposium*, 2009.
20. JGF. The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.
21. Roberto Lublinerman et al. Delegated Isolation. In *OOPSLA '11: Proceeding of the 26th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 2011.
22. Svetlin A Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC bioinformatics*, 9(Suppl 2):S10, 2008.
23. Parboil. Parboil benchmarks. http://impact.crhc.illinois.edu/parboil.aspx.
24. PolyBench. The polyhedral benchmark suite. http://www.cse.ohio-state.edu/ pouchet/software/polybench.