

# Boosting Java Performance Using GPGPUs

**DOI:**
10.1007/978-3-319-54999-6_5

**Document Version**
Accepted author manuscript

# Boosting Java Performance using GPGPUs

James Clarkson, Christos Kotselidis, Gavin Brown, and Mikel Luján

School of Computer Science
The University of Manchester
first.last@manchester.ac.uk

**Abstract.** In this paper we describe Jacc, an experimental framework which allows developers to program GPGPUs directly from Java. The goal of Jacc, is to allow developers to benefit from using heterogeneous hardware whilst minimizing the amount of code refactoring required.
Jacc utilizes two key abstractions: *tasks* which encapsulate all the information needed to execute code on a GPGPU; and *task graphs* which capture both inter-task control-flow and data dependencies. These abstractions enable the Jacc runtime system to automatically choreograph data movement and synchronization between the host and the GPGPU; eliminating the need to explicitly manage disparate memory spaces.
We demonstrate the advantages of Jacc, both in terms of programmability and performance, by evaluating it against existing Java frameworks. Experimental results show an average performance speedup of 19x, using NVIDIA Tesla K20m GPU, and a 4x decrease in code complexity when compared with writing multi-threaded Java code across eight evaluated benchmarks.

## 1 Introduction

Heterogeneous programming languages, such as CUDA [2] and OpenCL [3], enable developers to execute portions of their code on specialized hardware. Typically, this involves offloading work from a *host* onto a *device* such as a GPGPU, and doing this requires developers to be mindful of the different *contexts* their code may execute on. Hence, the developer is burdened with writing the application and the extra code to manage its execution over disparate devices. This paper describes a programming framework (JIT compiler and runtime system), which has been designed to eliminate, or automate, a large amount of this responsibility to help reduce the burden placed on developers.

Current established heterogeneous programming languages, such as CUDA and OpenCL, require developers to logically separate their applications into code that runs either on the host or on the device (known as a *kernel*). As a consequence, these approaches require additional code to co-ordinate execution between the host and kernels.

This paper describes a simplified heterogeneous programming model in the context of the Java language. We make use of implicit parallelism and task-based parallel execution. The **J**ava **Acc**eleration system, hereafter Jacc, is inspired by

and shares many similarities with directive-based approaches such as OpenMP 4.0 [15]. However, the true benefits of Jacc are derived from the Java programming language: modular, statically typed code and dynamic compilation. Thus, a Jacc application does not need to be ported across different operating systems or hardware devices and it is possible to compose complex processing pipelines from existing code. Overall the paper makes the following contributions:
(1) Provides an overview of Jacc, its components, and design rationale.
(2) Discusses how Jacc can be used to write concise data-parallel code in Java and the sub-set of the Java language supported.
(3) Analyzes the implementation of the internal components of Jacc. The Jacc JIT compiler, unlike most prior work, compiles Java bytecode directly to PTX code which can be executed directly by NVidia drivers.
(4) Provides an in-depth comparative performance analysis of Jacc and standard Java multi-threaded benchmarks.

## 2 The Jacc Framework

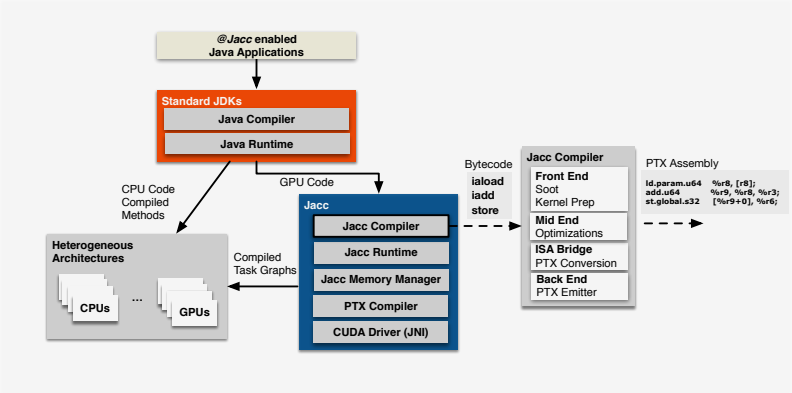



Fig. 1: Jacc system overview.

Jacc is a Java based framework which simplifies the programming of heterogeneous hardware. At present, we have been able to use Jacc to program a wide range of devices such as multi-core processors, Xeon Phi, and both embedded and discrete GPGPUs. In this paper we describe our initial prototype that has been developed to program CUDA enabled GPGPUs. As depicted in Figure 1, the two major components of Jacc are: its API and the runtime system.

The Jacc API has been designed to make possible the creation of high performance data-parallel code without forcing developers to, unnecessarily, change their software engineering practices. In order to support the API, Jacc has a runtime system that is able to manage the execution of application code on disparate hardware. This typically requires support for generating and executing

code, moving data between devices, and synchronization. Using both components together, Jacc is able to automate and optimize many common housekeeping tasks involved in writing heterogeneous code; relieving the developer from a number of burdens that exist in languages such as CUDA and OpenCL.

The API is built on top of two basic abstractions: the *task* and the *task graph*. A task encapsulates all the information needed to perform some action on a disparate hardware device such as code execution, data transfers or synchronization. Hence, a task which executes some code will encapsulate: a reference to the code, references to all the data accessed by the code and some *meta-data*. The meta-data is used to pass task specific parameters to the runtime system - such as the device it should execute on, the number of threads, or the size of each thread group, allowing dynamic adaption of those parameters during runtime.

Tasks which perform data transfers and synchronization are implicitly handled by the runtime system - leaving the developer responsible for defining only those which execute code. These tasks can be created from any method in the application. Furthermore, their meta-data contain a mapping which associates each one of them with the device it should execute on. Typically, this mapping is defined when a task is inserted into a task graph, but as it is just an entry in their meta-data it can also be updated dynamically.

Executing tasks on a GPGPU requires a number of actions to be performed: compilation, data movement to the GPGPU, execution on the GPGPU, and data movement back to the host. Although this can be done synchronously, it is inefficient to execute tasks in this manner - especially when multiple tasks operate on shared data. To make task execution more efficient, Jacc provides the *task graph* abstraction - a mechanism which allows the runtime system to optimize task execution through lazy evaluation. After a task graph is created, the runtime system uses its meta-data to build an executable *directed acyclic graph* (DAG). Once built, the runtime system is able to optimize the DAG by inspecting task meta-data to remove redundant data transfers and re-organize the order in which tasks are executed.

Jacc exploits many features of the Java platform in order to simplify the development workflow. The GPGPU code is directly generated from Java bytecode which avoids the need to either: embed source code inside the application, like OpenCL, or re-parse the source code. This means that the code running on the GPGPU is created using a single type system, unlike OpenCL which introduces a second type system to the developer.

### 2.1 Writing Data Parallel Code

There are two ways in which developers can write parallel code: explicitly or implicitly. Although the Jacc framework supports both, implicit parallelism is strongly encouraged since the code will produce the same result whether executed serially or in parallel. This provides Jacc with the option to revert back to serial execution if an error is encountered whilst offloading onto the GPGPU.

Jacc provides an annotation based API, similar to OpenMP, which allows developers to statically define task meta-data. However, unlike OpenMP this

meta-data can also be provided or adapted dynamically at runtime. For instance, information such as the parallelization strategy and type of variable access, specified by the `@Jacc`, `@Read`, `@Write` and `@ReadWrite` annotations, is better defined statically. In certain circumstances it may be beneficial to override these settings - for example to ensure data is always fetched from the host and not cached on the GPGPU or if a specific device responds better to a different parallelization scheme. The only aspects of the API which cannot be overridden are the ones which directly influence code generation, such as `@Atomic` or `@Shared`, as they are embedded directly into Java bytecode.

To produce data-parallel code, the Jacc compiler has the ability to re-write certain classes of loop-nests so that each iteration of a loop is executed by a different thread. This can be done by adding to a method the `@Jacc` annotation and setting the `iterationSpace` parameter. The iteration space parameter defines how many levels of the loop-nest should be re-written. (e.g. A value of 2 will re-write the two outermost loops.) Since it is not possible to use annotations at a sub-method granularity in Java 7, the Jacc compiler will only parallelize the first loop-nest encountered in a method[1].

As some loop-nests communicate data between iterations, Jacc provides the ability to perform inter-thread communications via shared memory atomics. A field can be declared as `@Atomic` which forces the compiler to use atomic operations when reading from and writing to this field. To support reduction operations, it is possible to specify an operation that can be applied in each update of the field. In this case, the field is initialized with a default value at the start of execution and then updated with the result of applying the operation to the existing and incoming values.

In cases where it is impossible to express a kernel using a single loop-nest, the developer has two choices: to split functionality across multiple kernels or to manually parallelize the code similarly to CUDA and OpenCL. The advantage of the latter approach is that developers can create highly optimized parallel code for a specific device. Unfortunately, this comes at the expense of reduced code re-use as Java applications cannot readily use this code.

Figure 2 (right) provides an example of how the data-parallel code is written while Figure 2 (left) demonstrates how a task is created and scheduled using a task-graph. Initially, we want each iteration of the outermost loop to be executed by independent threads — each thread will read a single element of the array and accumulate the value in `result`. To achieve this, a parallelization strategy is selected in line four, using the `@Jacc` annotation, to specify that only the outer-most loop should be parallelized. Finally, to handle the accumulation of partial results in the `result` variable, line 11, we use the `@Atomic` annotation which instructs the compiler to update this variable atomically.

In order to execute this code on a GPGPU, we need to define a task, add it to a task-graph, and schedule it. This is shown in Figure 2 (left) where the task is defined in lines 1-11. In this case, the task-graph consists of a single task which has been mapped onto the GPGPU. The number of threads used and the dimen-

[1] This problem is resolved in Java 8.

sions of each thread group are defined in lines 7-8, where `array.length` threads are specified - one for each iteration of the loop. On invoking the `execute` method of the task-graph, the runtime system will: compile the code for the GPGPU, move data to the GPGPU, execute the code, and synchronize the data between the host and the GPGPU.

```
1  DeviceContext gpgpu =
2   Cuda.getDevice(0).createContext();
3
4  Reduction r = new Reduction(...);
5  Task task = Task.create(
6      Reduction.class,methodName,
7      new Dims(array.length),
8      new Dims(BLOCK_SIZE));
9
10 task.setParameters(r, data);
11 tasks = new TaskGraph() {
12   @Override
13   public void create() {
14     executeTaskOn(task, gpgpu);
15   }
16 }
17 tasks.execute();
```

```
public class Reduction {
  @Atomic(op=ADD) float result;

  @Jacc(iterationSpace=ONE_DIMENSION)
  public void reduction(
      @Read float[] array) {
    float sum=0;
    for(int i=0;i<array.length;i++) {
      sum+=array[i];
    }
    result=sum;
  }
}
```

Fig. 2: Left: Reduction by generating a TaskGraph, Right: Reduction operation using implicit parallelism.

### 2.2 Current Subset of Java Supported for Execution on GPGPUs

**Objects:** Jacc provides object support and is able to freely access fields and invoke methods on objects or classes[2]. Jacc is not integrated directly with the garbage collector and, thus, it only supports the manipulation of existing objects on the GPGPU. However, due to escape analysis, stack allocated objects can be freely accessed. In practise, we have found that most tasks amenable for GPGPU offloading perform some form of volume reduction and object creation is often not needed. At present we do not maintain object headers in order to reduce storage requirements and improve serialization times. Consequently, we do not yet support reflection or the `instanceof` keyword[3].

**Arrays:** Use of arrays of primitives, objects and multi-dimensional arrays is supported as long as the element type is not an interface.

**Virtual and Static Method Calls:** Practically, the aggressive use of inlining removes all method calls except polymorphic calls which introduce indirection into the generated code. Tasks can be created from either static or virtual methods. The only difference between these two, is that the developer must remember to insert the `this` object reference as the first task parameter. The advantage of virtual methods is that the `this` object reference neatly encapsulates state that needs to be shared among multiple kernels.

[2] However, this can easily lead to a large number of indirect-memory accesses in the generated code - which will degrade performance on a GPGPU.

[3] There is no technical reason why support can not be added at a later date.

**Memory Allocations:** Jacc is able to support the `new` keyword under certain circumstances. The compiler will try to inline the constructor and any memory is allocated on the stack. Additionally, the use of inlining enables the elimination of a number of field accesses using scalar replacement. If the developer wishes to allocate memory in a certain memory space, the variable must be declared as a field with the declaration using the annotation specifying the memory space.

**Assertions and Exceptions:** Jacc has the ability to handle assertions and some limited exception checking on the GPGPU. Exception checks such as null pointer and array index out of bounds can be inserted by the compiler. If the runtime system detects that an exception has been thrown, it will attempt to run the same code within the JVM to produce a valid stack trace.

## 3 Runtime System

### 3.1 JIT Compiler

The Jacc JIT compiler, shown in Figure 1, unlike most prior work compiles Java bytecode directly to PTX code which can be executed directly by NVidia drivers. The compiler is organized in three layers: the front-end - responsible for parsing bytecode; the mid-end - responsible for transforming and optimizing the code for data-parallel execution; and the back-end - responsible for emitting the GPGPU specific machine-code. The front-end of the compiler has been implemented using the SOOT framework [17]. It generates various levels of IR from Java bytecode and leverages a number of advanced optimizations (e.g. common sub-expression elimination, loop invariant code motion, copy propagation, constant folding, straightening, and dead code elimination).

Initially, the IR is augmented with information about kernel entry points, exception handlers, and sets up accesses to the different memory spaces. Next, an optional transformation performs parallelization - this involves searching for loop-nests and updating the schedule of their induction variables so that iterations are assigned to different threads. This update is dependent on the value of the `iterationSpace` parameter specified in the meta-data of each task[4]. After parallelization, the remainder of the mid-end aims to generate high quality data-parallel code through a set of optimizations.

To optimize away costly functions calls, we search the IR for call-sites which map directly onto hardware instructions and replace them with appropriate intrinsics. If it is not possible to substitute a specific call-site, the compiler then tries to inline the code. If inlining is deemed infeasible the compiler will generate the code to support the call. If the compiler is unable to determine the actual method invoked at a particular call-site, the compilation will be terminated with an exception. Additionally, the compiler tries to minimize the number of branches in the IR. For example, it attempts to fully exploit the fact that PTX

[4] In our experience, the majority of kernels that we could not auto-parallelize using this scheme was due containing multiple loop-nests.

supports predicated execution by replacing simple branch statements with predicated instructions.

The mid-end is also responsible for handling code which access variables that are stored in different memory spaces or use shared memory atomics. Data-flow analyses are used to discover which loads/stores access a particular memory space and templated code is used to handle the initialization and update of variables accessed using atomics.

After passing through the mid-end, the IR goes through a lowering process which converts each statement of the IR into lower-level IR statements which generate one or more PTX instructions — this is marked as the ISA bridge. Finally, the PTX emitter converts each statement into valid PTX instructions.

### 3.2 Memory Management

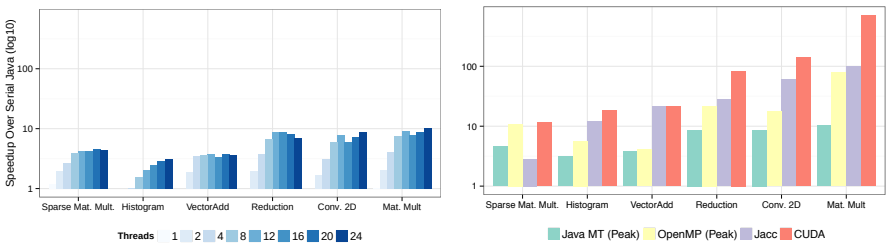
As a prerequisite to execution, data must be pre-loaded into the GPGPU memory by a memory manager (an instance is assigned to each device). To enable Jacc to target as many devices as possible, we have taken the decision that the Jacc runtime should be responsible for explicitly managing GPGPU memory; opposed to using CUDA's unified memory — as it is not yet available on all devices. This also has the secondary advantage of allowing Jacc to optimize data layout on a per-device basis. Typically, Jacc is able to avoid copying un-used data and minimizes the number of indirect memory accesses in the code. Hence, the memory manager is responsible for maintaining a custom data layout scheme. The format used is built dynamically, in concert between the memory manager and the compiler, and is communicated to the compiler and data serializer via a *data schema*. The generated schema maps each element of a composite type onto a specific memory location (relative to a given address). If the runtime system wishes to transfer data to the GPGPU it must serialize each object according to the schema provided by the memory manager.

A key design goal of Jacc is the ability to allow data to persist on the GPGPU. This feature makes possible to have multiple tasks or even task-graphs operate on the same data - avoiding the continual need to transfer data between host and device. However, as Jacc is unable to determine whether an object has been modified on the JVM, the developer is responsible for maintaining the state of persistent data. Typically, Jacc ensures shared state remains consistent by blocking until the task-graph has finished executing, at which point the memory managers will have synchronized any modified data with the host.

Generally, variables or arrays of primitive types can be copied "as-is" and composite types are laid out according to the data schema provided by each device manager. In order to tackle the data serialization process of objects, we developed a novel compiler driven approach that dynamically builds data schemas during compilation. A schema starts empty, and as compilation progresses and new composite-types are discovered, dynamic new data schemas are built with

on-demand object references. This minimizes the number of objects transferred to the device during data serialization[5].

## 4 Evaluation




(a) (b)

Fig. 3: Left: The speedups obtained using multi-threaded Java code only, Right: The performance of GPGPU accelerated implementations normalized to the performance of the serial Java implementation.

The experimental hardware platform used has two Intel Xeon E5-2620 processors (12 cores / 24 threads total @2.0 GHz), 32GB of RAM and a NVIDIA Tesla K20m GPGPU with 5 GB of memory. Regarding the experimental software stack, CentOS 6.5, CUDA 6.5 and Java SDK 1.7.0_25 were used. All CUDA implementations are taken from the CUDA SDK except the matrix multiplications: SGEMM is taken from the `cuBLAS` library and SPMV from `cuSPARSE`. The benchmarks used for the performance evaluation are:
**Vector Addition** adds two 16,777,216 element vectors (300 iterations).
**Reduction** performs a summation over an array of 33,554,432 elements (500 iterations).
**Histogram** produces frequency counts for 16,777,216 values placing the results into 256 distinct bins (400 iterations).
**Dense Matrix Multiplication** of two $1024 \times 1024$ matrices (400 iterations)[6].
**Sparse Matrix Vector Multiplication** performs a sparse matrix-vector multiplication using a $44609 \times 44609$ matrix with 1029655 non-zeros (The bcsstk32 matrix from Matrix Market) (400 iterations).
**2D Convolution** of a $2048 \times 2048$ image with a $5 \times 5$ filter (300 iterations).
**Black Scholes** is an implementation of the Black Scholes option pricing model. The benchmark is executed to calculate 16,777,216 options over 300 iterations and is supplied as an example in the APARAPI source code.
**Correlation Matrix** is an implementation of the Lucene OpenBitSet "intersection count". The benchmark is executed using 1024 Terms and 16384 Documents

[5] The schema also tracks which fields are accessed and modified by the code, to minimize the cost of synchronizing data with the host after a task has been executed.
[6] The OpenMP implementation uses the OS supplied `libatlas` library.

and is supplied as an example in the APARAPI source code. Only a single iteration is performed.

Jacc is compared against: serial Java, multi-threaded Java, OpenMP, CUDA and the more mature APARAPI [1] framework that uses OpenCL [3]. The performance of each benchmark is calculated by measuring the time to perform the specified number of iterations of the performance critical section of the benchmark. Each quoted performance number is an average across a minimum of ten different experiments. The reported Jacc execution times are inclusive of a single data transfer to the device and a single transfer to the host but exclusive of JIT compilation times. This is done in order to demonstrate both the peak-performance of Jacc generated code and the low-overheads of the runtime system. In terms of programmability, we take the stance that code complexity is proportional to code size and that code can be accelerated, using a GPGPU, without requiring any significant increase in code complexity over a multi-threaded implementation. We assess this by measuring the number of source code lines required to express the data-parallel kernel(s).

### 4.1 Java Multi-Threaded Performance

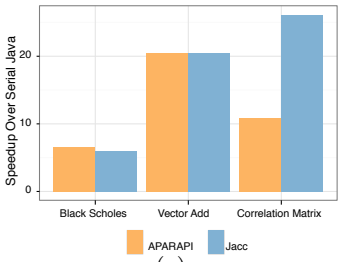
Figure 3a shows the speedups achieved by converting from serial to multi-threaded Java implementations. The results show that these benchmarks scale with increased thread counts. In this scenario, the largest performance increases are observed when the number of threads used is equal to or less than the number of physical cores in the system (up to 12 threads). Table 4b provides a summary of the peak performances of each benchmark and the number of threads used.

Figure 3b compares the same benchmarks against the Jacc implementations running on the GPGPU. As a sanity check, we have also implemented all benchmarks in OpenMP 3.2 and CUDA. By comparing the multi-threaded Java and OpenMP implementations, we see that our Java implementations have a number of inefficiencies. However, with the exception of the sparse matrix vector multiplication benchmark, Jacc still outperforms the OpenMP implementations. Furthermore, in order to provide a strong comparison point, the OpenMP version of SGEMM is provided by `libatlas`. Results indicate that even in this case Jacc is still able to outperform OpenMP, albeit by a reduced margin in comparison to Java multi-threaded implementations.

### 4.2 Performance and code size in heterogeneous environment

In terms of performance, Jacc is evaluated against: serial Java, multi-threaded Java, multi-threaded OpenMP and CUDA. The effect on programmability is studied by comparing the lines of code required to implement data-parallel code on the GPGPU against that required to write multi-threaded Java code.

Figure 4b summarizes the speedups obtained by Jacc against our Java implementations. We have normalized the speedups with the performance of two




(a)

| | Speedup | | | Lines of Code | | |
|---|---|---|---|---|---|---|
| Benchmark | Serial | Java MT | Peak | Java | MT Jacc | Reduction |
| Vector Add | 21.52 | 6.00 | (20) | 40 | 6 | 6.67x |
| Matrix Mult. | 98.56 | 13.08 | (24) | 46 | 16 | 2.88x |
| 2D Conv. | 60.31 | 10.18 | (24) | 66 | 33 | 2.00x |
| Reduction | 28.31 | 4.21 | (16) | 43 | 11 | 3.91x |
| Histogram | 11.86 | 7.53 | (24) | 61 | 8 | 7.62x |
| Sparse Mult. | 2.85 | 0.63 | (20) | 51 | 14 | 3.64x |
| Black Scholes | 5.93 | - | | - | - | - |
| Cor. Matrix | 26.16 | - | | - | - | - |
| Geo. Mean | 19.27 | 5.02 | | 50 | 13 | 4.01x |

(b)

Fig. 4: Left: Speedup obtained by APARAPI and Jacc over serial Java implementations, Right: A comparison of Jacc against Java based implementations.

different Java implementations: a serial Java implementation and the peak performing multi-threaded Java implementation. Results indicate that Jacc, on average, outperforms the serial and peak multi-threaded performance of all Java implementations by 19x and 5x respectively. Our pathological case is the sparse vector multiplication benchmark, where the irregular memory accesses pattern is not well suited to our current parallelization strategies. This can be resolved either algorithmically or through better code generation — assigning loop iterations on a per warp basis and making use of the texture cache. The table also contains the results of the difference in code complexity implementing a data-parallel kernel in multi-threaded Java or using Jacc. The results show that using Jacc to create data-parallel code requires 4x fewer lines of codes than writing them using Java threads.

Additionally, we compare against APARAPI [1], an alternative Java based framework, using three of their benchmarks: Vector addition, Black Scholes, and Correlation Matrix. A comparison of the results is shown in Figure 4a. To understand the impact of JIT compilation on performance, we conducted experiments that are both inclusive and exclusive of compilation times. Comparing the geometric mean of these speedups, we observe that both frameworks are very similar in terms of performance; APARAPI just incurs less overheads due to JIT compilation.

In contrast to our approach, APARAPI is built upon OpenCL and uses source-to-source translation to generate OpenCL C from Java bytecode. This approach provides APARAPI with two advantages: consistently low-compilation times, around 400 milliseconds, and a high quality of generated code. As our compiler matures, the cost of our JIT compilation will fall, so that it is comparable with APARAPI.

In the Correlation Matrix benchmark, Jacc significantly outperforms APARAPI because of its ability to easily tune the number of threads in each work group[7] and to replace an entire method with a single PTX instruction — `popc`.

[7] We found that changing Jacc's work group size, to match that of APARAPI, severely reduced performance but remained faster than APARAPI.

## 5 Related Work

Most prior work focuses on embedded support for heterogeneous programming inside existing languages targeting GPGPUs, FPGAs, vector units, and multi-core processors [1, 5, 10, 12, 9, 18, 16, 14, 8, 13]. Jacc is different from the majority of these efforts since it does not rely on translating bytecode into CUDA or OpenCL C to generate code for the GPGPU. Instead it generates PTX code which can be JIT compiled by the GPGPU driver. To the best of our knowledge, the most complete attempt at enabling the use of GPGPUs from Java is APARAPI [1] which translates Java bytecode into OpenCL C. We improve over APARAPI since we impose less restrictions on developers while making it easier to build complex multi-kernel codes. Jacc does not force developers to separate data-parallel code into singleton classes and our task-graph abstraction enables a series of runtime optimizations that are not possible in APARAPI. This work was being used as inspiration for the now defunct OpenJDK Sumatra project [4].

Rootbeer [16] is another attempt at exposing GPGPUs to Java developers. In contrast to APARAPI, it uses ahead of time code generation by extending SOOT [17] with support for emitting CUDA code. Other projects [5, 12, 10, 6, 13] use supersets of Java which include special syntax and language features to simplify the writing of data-parallel code, or advocate the use of a functional style programming on a specialized array class. Finally, projects such as [7, 11] use a new intermediate language (IL) which is enriched with support for parallel execution allowing the JIT to be embedded in domain-specific dynamic programming languages. Jacc is different from these approaches as we use an existing IL, Java bytecode, and we aim to support general purpose programming in Java.

## 6 Conclusions

Heterogeneous programming allows developers to improve performance by running portions of their code on specialized hardware resources. In this paper we have introduced the Jacc framework and shown how it is possible to write concise data-parallel code and execute it on GPGPUs. Moreover, our task-based programming model and runtime system means that a large amount of tedium associated with programming heterogeneous devices can be automated. Our experimental results demonstrate that Jacc is able to generate code which outperforms serial Java code by 19x on average and that it requires 4x less code than a multi-threaded Java implementation.

## 7 Acknowledgments

This work is supported by the AnyScale Apps and PAMELA projects funded by EPSRC EP/L000725/1 and EP/K008730/1. Dr Luján is supported by a Royal Society University Research Fellowship.

## References


1. Aparapi, http://developer.amd.com/tools-and-sdks/opencl-zone/aparapi/
2. CUDA, http://developer.nvidia.com/cuda-zone
3. OpenCL, https://www.khronos.org/opencl/
4. Project Sumatra, http://openjdk.java.net/projects/sumatra/
5. Auerbach, J., Bacon, D.F., Cheng, P., Rabbah, R.: Lime: A java-compatible and synthesizable language for heterogeneous architectures. In: OOPSLA '10: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. ACM (2010)
6. Catanzaro, B., Garland, M., Keutzer, K.: Copperhead: Compiling an embedded data parallel language. In: PPoPP '11: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming. PPoPP '11, ACM (2011)
7. Chafi, H., Sujeeth, A.K., Brown, K.J., Lee, H., Atreya, A.R., Olukotun, K.: A domain-specific approach to heterogeneous parallelism. p. 35. ACM Press (2011)
8. Chafik, O.: Scalacl: Faster scala: optimizing compiler plugin + gpu-based collections (opencl), http://code.google.com/p/scalacl
9. Dotzler, G., Veldema, R., Klemm, M.: JCudaMP. In: Proceedings of the 3rd International Workshop on Multicore Software Engineering (2010)
10. Fumero, J.J., Steuwer, M., Dubach, C.: A composable array function interface for heterogeneous computing in java. In: ARRAY'14: Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming. ACM (2014)
11. Garg, R., Hendren, L.: Velociraptor: An embedded compiler toolkit for numerical programs targeting cpus and gpus. In: PACT '14: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation. ACM (2014)
12. Hayashi, A., Grossman, M., Zhao, J., Shirako, J., Sarkar, V.: Accelerating habanero-java programs with opencl generation. In: Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (2013)
13. Herhut, S., Hudson, R.L., Shpeisman, T., Sreeram, J.: River trail: A path to parallelism in javascript. In: OOPSLA '13: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages Applications. ACM (2013)
14. Nystrom, N., White, D., Das, K.: Firepile: Run-time compilation for gpus in scala. In: GPCE '11: Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering. ACM (2011)
15. OpenMP Architecture Review Board: OpenMP Specification (version 4.0) (2014)
16. Pratt-Szeliga, P., Fawcett, J., Welch, R.: Rootbeer: Seamlessly using gpus from java. In: Proceedings of 14th International IEEE High Performance Computing and Communication Conference on Embedded Software and Systems (2012)
17. Vallèe-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Phong, C.: Soot - a java optimization framework. In: Proceedings of CASCON 1999 (1999)
18. Yan, Y., Grossman, M., Sarkar, V.: Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In: Sips, H., Epema, D., Lin, H.X. (eds.) Euro-Par 2009 Parallel Processing (2009)