Running Parallel Bytecode Interpreters on Heterogeneous Hardware

Juan Fumero
juanfumero@acm.org
The University of Manchester
Manchester, UK

Athanasios Stratikopoulos
{first.last}@manchester.ac.uk
The University of Manchester
Manchester, UK

Christos Kotselidis
christos.kotselidis@manchester.ac.uk
The University of Manchester
Manchester, UK

ABSTRACT

Since the early conception of managed runtime systems with tiered JIT compilation, several research attempts have been made to accelerate the bytecode execution. In this paper, we extend prior attempts by performing an initial analysis of whether heterogeneous hardware accelerators in the form of Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAS) can help towards achieving higher performance during the bytecode interpreter mode. To answer this question, we implemented a simple parallel Java bytecode interpreter written in OpenCL and executed it across a plethora of devices, including GPUs and FPGAs. Our preliminary evaluation shows that under specific workloads, hardware acceleration can yield up to 17x better performance compared to traditional optimized interpreters running on Intel CPUs and up to 214x compared to ARM CPUs.

CCS CONCEPTS

• **Software and its engineering** → **Interpreters**; • **Hardware** → **Hardware accelerators**; • **Computing methodologies** → *Parallel programming languages.*

KEYWORDS

Bytecode, Interpreters, GPUs, FPGAs, Heterogeneous Architectures

ACM Reference Format:
Juan Fumero, Athanasios Stratikopoulos, and Christos Kotselidis. 2020. Running Parallel Bytecode Interpreters on Heterogeneous Hardware. In *Proceedings of 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Heterogeneous hardware acceleration of programs written in managed programming languages is currently feasible via different forms such as bindings of native GPU or FPGA code [1, 6, 7, 10] and/or dynamic JIT compilation of high-level programs to low-level GPU code or FPGA bitstreams [3–5, 14, 15]. In addition to being able to accelerate applications written in managed languages such as Java, heterogeneous hardware accelerators provide opportunities for accelerating key components of virtual machines such as interpreters [2] or parts of the Garbage Collection [8, 9] that due to their nature take longer a time to execute.

The current complexity of Virtual Machines (VMs) along with the plethora of features they support (e.g., reflection, dynamic code dispatch) indicate that not all of their parts can be accelerated on heterogeneous hardware since these devices are less capable compared to CPUs regarding access to Operating System (OS) functionalities and lack of unified memory address space. Despite the lack of these features, VMs can still benefit from the offloading of certain parts of the runtime systems onto parallel accelerators, that usually remain underutilized.

In this work [2], we present our work in progress towards running a parallel bytecode interpreter on heterogeneous hardware. We tackle the challenge of accelerating and parallelizing a simple bytecode interpreter on different heterogeneous hardware, and provide a first insight under which circumstances such acceleration is meaningful. In contrast to prior works, we make use of parallel thread identifiers in the bytecode interpreter to execute SIMD workloads on OpenCL-compatible devices. Additionally, we present a multiple-heap VM that can exploit the different memory tiers, present in hardware accelerators, optimizing memory accesses while improving data coalescing. In detail, this paper makes the following contributions:

- It presents a prototype of a parallel bytecode interpreter implemented in OpenCL capable of being executed on a wide range of hardware accelerators such as GPUs and FPGAs.
- It presents a device multiple-heap configuration that takes advantage of all memory tiers presented on OpenCL devices.
- It evaluates the parallel bytecode interpreter across three classes of devices on two different platforms, showcasing the performance benefits of parallel execution on heterogeneous hardware accelerators.

2 BYTECODE INTERPRETERS FOR HETEROGENEOUS HARDWARE

As a proof of concept, we implemented a subset of the Java bytecodes in C++ and OpenCL. We chose the C++ bytecode interpreter as a baseline for an initial OpenCL parallel prototype. The bytecodes defined correspond to an extension of the small-subset of Java bytecode explained by Terence Parr, from the University of San Francisco of how to build a simple Virtual Machine [1]. We extend this simple bytecode interpreter to study the feasibility of running, as efficiently as possible, parallel bytecode interpreters on heterogeneous computer architectures.

[1] https://www.youtube.com/watch?v=OjaAToVkoTw

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.1 Baseline C++ Bytecode Interpreter

Similarly to a standard Java Virtual Machine, our prototype executes bytecodes in a virtual stack-machine. The interpreter contains 25 bytecodes, grouped in the following categories:

- **Arithmetic operations:** We provide integer arithmetic operations such as `IDIV, IADD, IMUL, IDIV`. Additionally, we provide shift operations in the bytecode interpreter such as `RSHIFT`, and `LSHIFT`. Finally, we include comparison operations for integers, such as top of the stack less-than (`ILT`), and top of the stack equal-to (`IEQ`).
- **Memory operations**: We support loads and stores from the global heap memory to the stack, using the bytecodes `GLOAD, GSTORE` as well as loads and stores from/to different positions of the stack `LOAD/STORE`. To load and store data using the array format, we provide the bytecodes `GLOAD_INDEXED` and `GSTORE_INDEXED`. In our interpreter, the index is retrieved from the top of the stack. Additionally, we include fast and common load/store operations such as `ICONST`, that loads a constant value, and `ICONST1`, that loads the value 1.
- **Control flow:** We provide different bytecodes for defining control flow. These bytecodes include `BR, BRT, BRF, HALT, RET,` and `CALL`.
- **Interpreter control:** We include bytecodes such as POP, DUP.
- **Auxiliary bytecode:** We provide the `PRINT` bytecode as a utility to ease debugging and print the internal state.

Listing 1 shows an example for the vector multiplication operation between two vectors using our bytecodes. At first, a constant is loaded to be used as a loop bound, over which the two vectors are iterated. Then, two numbers that are loaded from the heap onto the stack are multiplied and the result is stored in a separate region of the heap.

2.2 Single Threaded OpenCL Interpreter

Since plain C++ code cannot be executed on GPUs, it has to be ported to a heterogeneous-friendly programming language such as OpenCL, CUDA, or SYCL. In contrast to GVM [2] that provides a Java interpreter implemented in CUDA for running on NVIDIA GPUs, we choose OpenCL because it is a standard and it can be executed on many heterogeneous hardware including GPUs, multi-core CPUs and FPGAs, while using the same source code.

Listing 1: Vector multiplication using our bytecode interpreter.

```
1  ICONST, 0,
2  DUP,
3  ICONST, SIZE,    // Define the vector size
4  IEQ,
5  BRT, 23, // jump if true to bc=23
6  DUP,     // offset for each array to load
7  DUP,     // offset for each array to load
8  GLOAD_INDEXED, SIZE, // topStack=heap[SIZE+offset]
9  LOAD, 1,   // load from position 1
10 GLOAD_INDEXED, SIZE * 2,
11 IMUL,
12 GSTORE_INDEXED, BASE, // heap[BASE+offset] = topStack
13 ICONST1,
14 IADD,
15 BR, 2,   //jump to bc=2
16 POP,
17 HALT
```

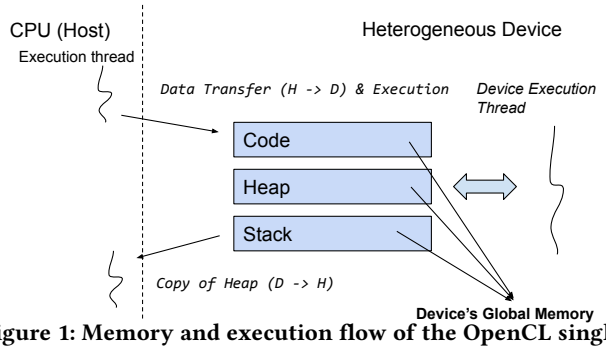

Figure 1: Memory and execution flow of the OpenCL single threaded interpreter.

Our first approach is to port the whole C++ interpreter as a single OpenCL kernel that runs using one OpenCL thread. Our interpreter, similarly to the JVM or any other virtual machine, defines different memory regions that store the code, the stack, and the heap. Figure 1 shows a representation of how those memory regions are organized on the OpenCL device. As shown, the code region, the heap as well as the stack regions are copied to the device's global memory (e.g., to the GPU's global memory). Once data is copied, we launch the kernel using an OpenCL range of one thread for global and local dimensions. We choose this initial approach because it is the simplest way to run with OpenCL without having to specify data partitioning and thread-id configurations. As it will be explained in the evaluation section, this approach does not perform well due to the lack of efficient hardware resource utilization.

2.3 Parallel Interpreters

Having the single-threaded OpenCL implementation as our baseline, we introduce a number of bytecodes in the interpreter that allow us to access data and compute resources in parallel. This leads us to a better resource utilization of the heterogeneous devices, in the form of the parallelism available on GPUs or multiple compute units available on FPGAs.

Thread identifier. To allow parallel access of data and parallel computation, we introduce the concept of thread identifier (thread-id) in our interpreter. We create a new bytecode, called `TREAD_ID` that stores the OpenCL work-item index (`get_global_id`) onto the stack. In this configuration, we can run a bytecode interpreter using multiple threads. Each thread accesses the corresponding data items, following the Single Instruction Multiple Thread parallel (SIMT) model present in the OpenCL programming model.

Multi-heap configuration. Since the OpenCL memory model defines different tiers of memory, our interpreter exploits those regions with the aim of improving data locality and memory accesses. We achieve that by introducing the concept of "multiple heaps". Our OpenCL parallel interpreter handles access to different memory areas depending on the content of the data to be accessed. Those regions are as follows:

- **Code region:** Since the code is static, we copy the bytecode to the *constant memory* area of the heterogeneous device. Depending on the target device (e.g., a dedicated GPU), this region allows faster memory accesses with the promise that code (bytecodes) will not change during runtime.

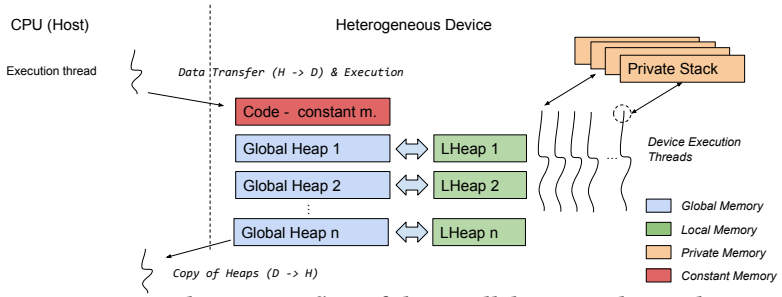

Figure 2: Memory and execution flow of the parallel OpenCL bytecode interpreter.

- **Local regions:** Since the interpreter accesses the heap frequently (for every load and store within the loop), we can make use of local memory. This area is a scratchpad memory that provides faster memory accesses and is located in the L3 cache of integrated GPUs and the L1 cache of discrete GPUs. In the case of an FPGA device, the local memory is the on-device block memory which has low latency.
- **Private regions:** data that can be accessed privately among all threads can be stored in private memory. Hence, we use this region to allocate the stack. Therefore, each device's thread (e.g., each GPU/FPGA thread) contains a private stack.

Having a single heap can lead to multiple memory accesses to obtain the data required to perform the computation, thereby causing memory contention. Since the OpenCL kernel is executed using hundreds, or even thousands of physical threads, we decided to provide multiple global heaps. Each global heap stores a subset of the data. The first time each thread accesses memory, it loads 128 bytes (typical cache-line size on GPUs) in a single transaction. In our prototype we use three different heap areas that are allocated on the GPU/FPGA's global memory. Note that we introduced three global heaps just to illustrate the concept, but it could be any number of global heaps. The bytecode interpreter can make use of those heaps (0-2) to improve data accesses and data coalescing. To do so, we introduce a pair of bytecodes, named `PARALLEL_GLOAD_INDEXED` and `PARALLEL_GSTORE_INDEXED`. These bytecodes receive a parameter that indicates the heap number from which to read/write from/to. For example, the bytecode sequence:

$$PARALLEL\_GLOAD\_INDEXED, 0$$

will load an integer value from heap 0 to the stack. The index used is the value that is on top of the stack:

$$topStack \leftarrow heap0[top]$$

On the other hand, the bytecode sequence:

$$PARALLEL\_STORE\_INDEXED, 2$$

will store onto heap 2 an integer value from the top of the stack. The index used is the value that is on top of the stack:

$$heap2[top] \leftarrow (top - 1)$$

Figure 2 shows a high-level representation of the different memory regions used in the parallel interpreter as well as an overview of the execution workflow between the host and the device. When the OpenCL kernel is executed, it first copies the data from the global memory of each heap to local memory. Then it performs the operation using local and private memory. When the kernel finishes, it copies back the data from local to global memory to obtain the final results.

Listing 2: Vector-mult in the parallel bytecode interpreter.

```
1  THREAD_ID,
2  DUP,
3  PARALLEL_GLOAD_INDEXED, 0,  // stack = heap0[threadID]
4  THREAD_ID,
5  PARALLEL_GLOAD_INDEXED, 1,  // stack = heap1[threadID]
6  IMUL,
7  PARALLEL_GSTORE_INDEXED, 2, // heap2[threadID] = (topStack - 1)
8  HALT
```

Listing 2 shows an example of vector multiplication using the explained bytecodes that allow parallel access using the thread-id and multi-heap configuration. Note that this example uses three global heaps in which input data is read from heaps 0 and 1, and heap 2 is used for the output.

3 EVALUATION

We evaluate our prototype on two different GPUs: an NVIDIA GP100 with 16GB of memory, and an Intel Integrated HD Graphics (HD 630). We used NVIDIA 384.111 and Intel OpenCL 19.52.15209 Gen9 NEO drivers. Additionally, we evaluate our prototype on a Xilinx KCU1500 FPGA with 16GB of memory. The CPUs used for running the baseline configurations are an Intel i7-7700K CPU at 4.20GHz and a ARMv7 processor at 1GHz.

Methodology. We execute the vector multiplication application in a C++ single-threaded bytecode interpreter on CPUs (Intel and ARM), and an OpenCL version of our parallel bytecode prototype on the Intel CPU, the GPUs and the FPGA. We report the OpenCL kernel time that takes to execute the whole parallel bytecode interpreter on the target device along with the execution time of the sequential bytecode interpreters on ARM and Intel CPUs. We execute each application 11 times and report the median value.

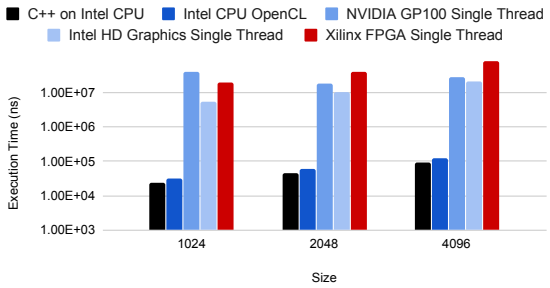

Figure 3: Total execution time of the vector multiplication application for the C++ implementation on CPU, OpenCL in an Intel CPU, OpenCL on NVIDIA GP100, Intel HD Graphics and Xilinx FPGA using a single thread.

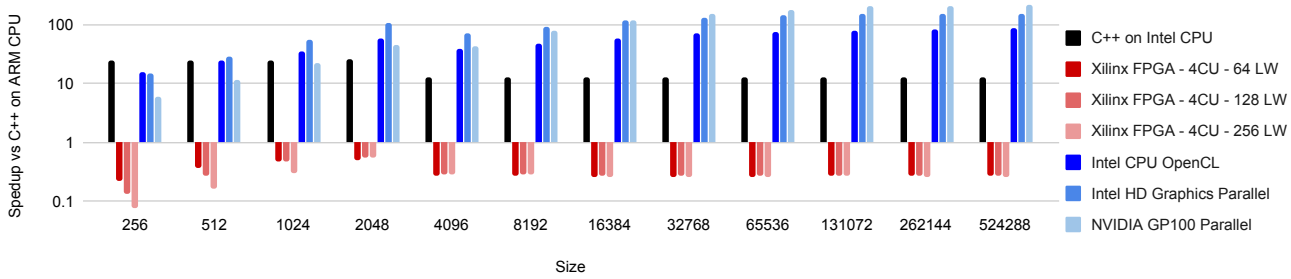

Figure 4: Performance comparison of parallel interpreters against single threaded C++ on ARM CPU.

Single Thread Interpreter. Figure 3 shows the total execution time, in nanoseconds, of the vector multiplication application for three input sizes. The first bar shows the total execution time on an Intel CPU using the C++ implementation. The second bar shows execution time of the OpenCL single-threaded interpreter on an Intel CPU. The following two bars show the performance of the single-threaded implementation in OpenCL on GPUs (NVIDIA and Intel HD Graphics), and the last bar shows the execution time on a Xilinx FPGA. As shown in Figure 3, the single-threaded OpenCL implementation is between 2 – 3 orders of magnitide slower than the C++ implementation on Intel CPU, meanwhile the OpenCL-CPU performs within 75% of the C++ interpreter on CPU. As we explain in Section 2.2, this is due to the underutilized computing and memory resources.

Performance of Parallel Interpreters. Figure 4 shows the speed-up of the OpenCL parallel interpreters for the vector multiplication for a wider range of input data sizes compared to the execution of the interpreter on an ARMv7 processor at 1GHz. We choose ARM as a baseline because it operates at a similar frequency to the heterogeneous devices. As shown, the multi-threaded CPU OpenCL interpreter achieves up to 84x speedup compared to the execution of the single-threaded interpreter running on ARM and 6x speedup compared to the execution on an Intel CPU (the black bar). GPU interpreters achieve speedups of up to 151x and 214x compared to the execution on ARM and 11x and 17x compared to the execution on an Intel. Regarding FPGAs, we show different configurations using 4 compute units and different local group sizes (64, 128 and 256 threads). However, none of those configurations achieve speedups. The observed slowdowns are attributed to the combination of data depenendencies, simple math operations, and the low frequency that our FPGA operates on (300MHz).

How this technique could be applied in current VMs? The presented prototype has demonstrated that it is feasible to execute bytecode interpreters on heterogeneous architectures, offering better performance, even for simple computations. Despite that the parallel bytecode interpreter is not sufficient to represent mainstream languages' bytecodes, this technique can be seen as a complementary strategy for running faster bytecodes of some subsections of applications. For example, for compute methods that follow SIMD and pipelining, VMs can transform original bytecodes to index data using the `THREAD_ID` and multi-heap configuration and run those parts of bytecodes on an interpreter that operates on a heterogeneous device. This can be suitable in the context of TornadoVM [4], in which FPGA compilation might take hours to perform the JIT compilation. During that time, TornadoVM can execute the user application on a parallel interpreter on GPUs. Additionally, VMs could make use of heterogeneous hardware for other operations, such as GC [9].

4 RELATED WORK

Limited prior works exist regarding the execution of bytecode interpreters on heterogeneous hardware. GVM [2] is a GPU bytecode interpreter for Java programs. Although GVM executes on GPUs, there are many differences compared to our approach. GVM executes a single-threaded CUDA interpreter and in order achieve better performance, it executes many bytecode interpreters on the GPU. In contrast, we propose launching one interpreter that has the notion of thread-parallelism through the `THREAD_ID` bytecode. Moreover, GVM stores all data in the global memory of the GPU, while our approach takes full advantage of the different memory tiers present on OpenCL devices, such as private, local, and constant memory. Additionally, our approach is fully implemented in OpenCL. Therefore, it can be executed not only on NVIDIA GPUs but on any OpenCL-compatible device, such as FPGAs, dedicated and integrated GPUs.

JopCMP [11, 13] is a real time Java shared-memory processor specialized for running Java applications. JopCMP is a stack-based processor that efficiently runs Java bytecodes and it is implemented using a low-level hardware description language. Since our approach is implemented in OpenCL, it is more generic than JopCMP, and it is able to run on multiple OpenCL device compatible devices, including GPUs, multi-thread CPUs and integrated GPUs.

5 CONCLUSION

This paper presented a work-in-progress towards exploiting bytecode interpreters on heterogeneous hardware, such as GPUs and FPGAs. To efficiently run managed language bytecode interpreters on GPUs and FPGAs, this paper introduced the concept of parallel bytecode interpreters, in which the runtime has the notion of parallel indexing. In this way, bytecode interpreters can exploit data parallelism. Additionally, we introduced the concept of multiple heaps for heterogeneous architectures in which objects and bytebuffers can access different levels of the tiered memory. In future work, we plan to extend the evaluation with more benchmarks and include comparisons against existing interpreted programming languages, such as Python, R or Ruby. Finally, we aim to exploit parallel interpreters using the SYCL parallel programming standard for C++ [12].

ACKNOWLEDGMENTS

This work is partially supported by the European Union's Horizon 2020 E2Data 780245 grant.

REFERENCES

[1] P. Bellows and B. Hutchings. 1998. JHDL-an HDL for reconfigurable systems. In *IEEE 6th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*.

[2] Ahmet Celik, Pengyu Nie, Christopher J. Rossbach, and Milos Gligoric. 2019. Design, Implementation, and Application of GPU-Based Java Bytecode Interpreters. *Proc. ACM Program. Lang.* 3, OOPSLA, Article Article 177 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360603

[3] James Clarkson, Juan Fumero, Michail Papadimitriou, Maria Xekalaki, and Christos Kotselidis. 2018. Towards Practical Heterogeneous Virtual Machines. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming (Programming'18 Companion)*. Association for Computing Machinery, New York, NY, USA, 46–48. https://doi.org/10.1145/3191697.3191730

[4] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. 2019. Dynamic Application Reconfiguration on Heterogeneous Hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2019)*. Association for Computing Machinery, New York, NY, USA, 165–178. https://doi.org/10.1145/3313808.3313819

[5] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. 2017. Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '17)*. ACM, New York, NY, USA, 60–73. https://doi.org/10.1145/3050748.3050761

[6] Steve Guccione, Delon Levi, and Prasanna Sundararajan. 1999. JBits: Java based interface for reconfigurable computing. In *Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*.

[7] JOCL 2017. Java bindings for OpenCL. http://www.jocl.org/.

[8] Martin Maas, Krste Asanoviundefined, and John Kubiatowicz. 2018. A Hardware Accelerator for Tracing Garbage Collection. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, 138–151. https://doi.org/10.1109/ISCA.2018.00022

[9] Martin Maas, Philip Reames, Jeffrey Morlan, Krste Asanoviundefined, Anthony D. Joseph, and John Kubiatowicz. 2012. GPUs as an Opportunity for Offloading Garbage Collection. In *Proceedings of the 2012 International Symposium on Memory Management (ISMM '12)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/2258996.2259002

[10] Thierry Moreau, Tianqi Chen, and Luis Ceze. 2018. Leveraging the VTA-TVM Hardware-Software Stack for FPGA Acceleration of 8-bit ResNet-18 Inference. In *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-designing Pareto-efficient Deep Learning*.

[11] Christof Pitter and Martin Schoeberl. 2010. A Real-Time Java Chip-Multiprocessor. *ACM Trans. Embed. Comput. Syst.* 10, 1, Article Article 9 (Aug. 2010), 34 pages. https://doi.org/10.1145/1814539.1814548

[12] Ruyman Reyes and Victor Lomüller. 2015. SYCL: Single-source C++ accelerator programming.. In *PARCO*. 673–682.

[13] Martin Schoeberl. 2005. Design and Implementation of an Efficient Stack Machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*. IEEE. https://doi.org/10.1109/IPDPS.2005.161

[14] Oren Segal, Martin Margala, Sai Rahul Chalamalasetti, and Mitch Wright. 2014. High Level Programming for Heterogeneous Architectures. *CoRR* (2014).

[15] Wojciech Zaremba, Yuan Lin, and Vinod Grover. 2012. JaBEE: Framework for Object-oriented Java Bytecode Compilation and Execution on Graphics Processor Units. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*.