Heterogeneous Managed Runtime Systems: A Computer Vision Case Study

Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján
School of Computer Science
The University of Manchester
Oxford Road, Manchester, M13 9PL, UK
{christos.kotselidis,james.clarkson,andrey.rodchenko,andy.nisbet,john.mawer,mikel.lujan}@manchester.ac.uk

Abstract

Real-time 3D space understanding is becoming prevalent across a wide range of applications and hardware platforms. To meet the desired Quality of Service (QoS), computer vision applications tend to be heavily parallelized and exploit any available hardware accelerators. Current approaches to achieving real-time computer vision, evolve around programming languages typically associated with High Performance Computing along with binding extensions for OpenCL or CUDA execution.

Such implementations, although high performing, lack portability across the wide range of diverse hardware resources and accelerators. In this paper, we showcase how a complex computer vision application can be implemented within a managed runtime system. We discuss the complexities of achieving high-performing and portable execution across embedded and desktop configurations. Furthermore, we demonstrate that it is possible to achieve the QoS target of over 30 frames per second (FPS) by exploiting FPGA and GPGPU acceleration transparently through the managed runtime system.

Keywords GPU Acceleration, Java Virtual Machines, Heterogeneous Runtime Systems, SLAM, Computer Vision

1 Introduction

Computer Vision (CV) applications, and in particular real time 3D space understanding, are becoming increasingly prevalent in both desktop and mobile domains. A good example is the field of robotics, where researchers are developing complex applications which demand high levels of performance. Furthermore, such applications could be deployed across different scenarios with diverse characteristics. For instance, the same CV application could be used in isolation to map a room using a mobile phone [35] or as a sub-component of a navigation system within a self-driving car [8].

A common characteristic of CV applications, regardless of the scenario they are used, is their extreme computational demands. Typically, they are written in programming languages such as C++ and OpenMP with binding extensions for OpenCL or CUDA execution. For example SLAMBench [27], a widely available benchmark that implements the Kinect Fusion (KF) application [28], provides implementations for all the aforementioned programming languages. A common drawback of such implementations is the lack of portability since the applications have to recompiled and optimized for each underlying hardware platform. Building and optimizing CV applications on top of a managed runtime system such as the Java Virtual Machine (JVM) would enable single implementations to run across multiple devices such as desktop machines or Android powered devices.

In the context of this paper we describe our experiences related to achieving a high-performing Java implementation of the Kinect Fusion application. After implementing and validating SLAMBench in Java, we performed an initial evaluation to identify performance bottlenecks. Consequently, we revised two optimization techniques leveraging the underlying heterogeneous hardware resources in order to meet the QoS target of the CV application. The developed optimizations follow two approaches: 1) a general purpose OpenCL accelerator, and 2) an application specific FPGA accelerator.

In detail, the paper makes the following contributions:

- Describes the implementation of a complex CV application in Java.
- Describes our work on providing a high-performing research JVM (Maxine VM) on low-power ARM architectures.
- Introduces two novel hardware acceleration techniques via the JVM which leverage FPGAs and GPGPUs transparently.
- Showcases that with the proposed acceleration techniques we are able to meet the QoS target of 30 FPS of a common CV application achieving up to 47X speedup compared to the original C++ implementation.

VEE'17, China

The paper is organized as follows: Section 2 presents the Computer Vision application that forms the use case in this paper. Section 3 explains the novel acceleration techniques developed along with the experimentation infrastructure. Finally, Section 4 presents the performance evaluations while Sections 5 and 6 present the related work and the concluding remarks, respectively.

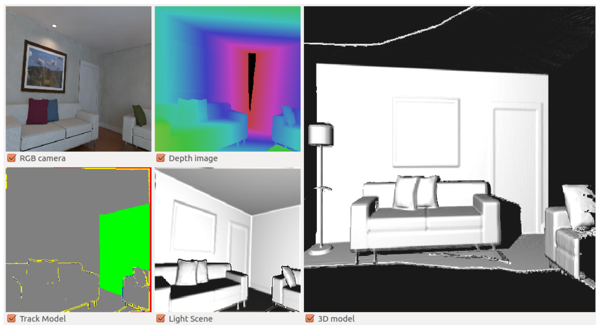


Figure 1. RGB-D camera combines RGB with Depth information (top left and middle). The tracking (left) results in the 3D reconstruction of the scene (right).

2 Kinect Fusion

Kinect Fusion (KF) is a Computer Vision application which reconstructs a three-dimensional representation from a stream of depth images produced by a RGB-D camera (Figure 1), such as the Microsoft Kinect. KF is described in [28] and a number of open-source implementation are provided by SLAMBench [27].

KF is a challenging application because in order to achieve its QoS target it needs to operate at the frame rate of the camera, which is 30 frames per second (FPS). Dropping below this frame rate means that pose changes, in both the camera and the subject, have the potential to become greater and, subsequently, finding correspondences between frames becomes increasingly difficult. KF is also interesting from an implementation perspective since there is an abundance of parallelism which can be exploited to improve its performance. Implementation-wise, some of the kernels are very large. For example, raycast is close to 250 lines of code (LOC) and expands to 1000 LOC in the OpenCL implementation while its performance is constrained by complex data dependencies.

2.1 Processing Pipeline

KF uses a stream of depth images from a Kinect camera as input to a six-stage processing pipeline (Figure 2a):

1. acquisition obtains the next RGB-D frame - either from a camera or from a file.
2. pre-processing is responsible for cleaning and interpreting the raw data by: applying a bilateral filter to remove anomalous values, rescaling the input data to represent distances in millimeters and, finally, building a pyramid of vertex and normal maps using three different image resolutions.
3. tracking estimates the difference in camera pose between frames. This is achieved by matching the incoming data to an internal model of the scene using a technique called Iterative Closest Point (ICP) [6, 39].
4. integrate fuses the current frame into the internal model, if the tracking error is less than a predetermined threshold.
5. raycast constructs a new reference point cloud from the internal representation of the scene.
6. rendering uses the same raycasting to produce a visualization of the 3D scene.

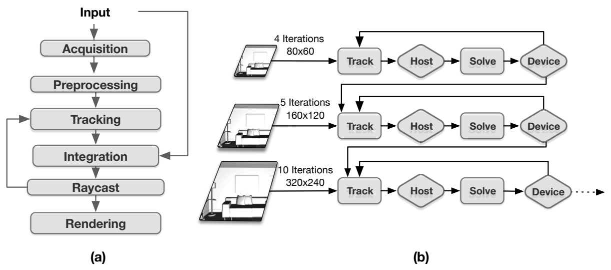


Figure 2. Kinect Fusion Pipeline stages.

Kernel	Stage	Invocations
mm2meters	preprocess	1
bilateral filter	preprocess	1
half sample	track	3
depth to vertex	track	3
vertex to normal	track	3
track	track	1 - 19
reduce	track	1 - 19
integrate	integrate	0 - 1
raycast	raycast	0 - 1
render depth	rendering	0 - 1
render track	rendering	0 - 1
render volume	rendering	0 - 1
Total	-	18 - 54

Table 1. List of KF kernels.

Each stage of the KF pipeline, as implemented in SLAMBench, is composed of a series of kernels. The breakdown of pipeline stages and invocation counts per kernel are shown in Table 1. We can see that a single frame of RGB-D data will require the execution of 18 to 54 kernels; in the best and worst case scenarios respectively. The variation is due to the performance of the tracking algorithm. If it is able to estimate the new camera pose quickly then fewer kernels will be executed. Therefore, to achieve a frame rate of 30 FPS, the application must sustain the execution of between 540 and 1620 kernels every second.

2.2 Tracking Algorithm

The tracking stage of the KF algorithm, depicted in Figure 2b, is the most complex in the pipeline and it uses an Iterative Closest Point (ICP) algorithm to estimate the difference in camera pose between two point clouds. The algorithm has two stages: 1) it finds correspondences between the incoming frame and its internal model - returning the error associated with each correspondence and, 2) it uses a least-squares approach to identify a new camera pose which minimizes this error. Finally, the algorithm iterates until the error is below a pre-configured threshold.

The complexities of implementing the tracking stage are compounded by using an iterative multi-scale ICP algorithm. In cases where KF is able to use a hardware accelerator, such as a GPGPU, the tracking algorithm is split so that the correspondences are found on the accelerator and the error minimization on the host. Consequently, in this situation, each iteration of the algorithm is required to transfer data between two memory spaces on the host and device (illustrated as diamonds in Figure 2b). The tracking kernel performs regular data transfers of 2.34 MB (320x240), 600 KB (160x120), and 150 KB (80x60) to the host. On the contrary, the host needs to transfer the new pose to the device after each iteration. Since each pose is represented by a $4{\times}4$ matrix, 64 bytes are required.

2.3 Measuring Performance and Accuracy

A challenge when comparing different implementations of KF, and CV algorithms in general, is that performance and accuracy measures are subjective. Normally, this is due to the real measure of the algorithmic quality being the user experience: does the user notice slow performance and is it accurate enough for their needs? Nevertheless, we must ensure that each implementation of KF does the same work and produces the same answer. Therefore, out of a number of KF implementations we have selected the ones provided by SLAMBench since they provide ready-made infrastructure to measure the performance and accuracy, enabling reliable comparisons between different implementations.

The accuracy of each reconstruction is determined by comparing the estimated trajectory of the camera against a provided ground truth, and is reported as an absolute trajectory error (ATE). The ground truths are provided by the synthetically generated ICL-NUIM dataset [18]. Finally, the performance is measured as the average frame rate achieved when processing the entire dataset.

2.4 Portability Issues

SLAMBench provides implementations in C++, OpenMP, CUDA, and OpenCL, enabling various performance points depending on the hardware unit executed. However, to achieve portability, current SLAMBench implementations, at the very least, have to be re-compiled on every platform; something that is not always possible e.g. OpenMP on OSX.

Another issue is that users may have to re-write key kernels for each target device. For example, reduction-style kernels are implemented to exploit a specific internal organization of a device. This means that the code will have to be re-written if the organization changes, or more subtly if some physical characteristics of the device changes, such as the amount of local memory or the maximum number of work items in a work group. This is a problem which affects all languages since developers may need to change tile sizes and thread scheduling for each device.

Programming languages implemented on top of managed runtime systems, such as the JVM, allow application execution regardless of operating system or hardware architecture. However, such implementations may perform slower especially in scenarios where heterogeneous acceleration is involved. In the remainder of the paper we discuss a number of techniques that can be used to accelerate the performance of our portable Java Kinect Fusion implementation in order to achieve our desired QoS level.

2.5 Java Implementation

Our Java reference implementation is derived from the open-source C++ version provided by SLAMBench. During porting, we ensured that the Java implementation produces bit-exact results when compared to the C++ one[1]. This is highly important, and challenging, since Java does not support unsigned integers. Therefore, we had to modify the code to use signed representations and maintain correctness. Although all kernels produce near identical results during unit-testing, each implementation can produce slightly differing results when combined together due to the nature of floating-point arithmetic.

We have developed the Java implementation with minimal dependencies on third-party code and we do not use any form of Foreign Function Interface (FFI) or native libraries. Our only dependency is on the EJML library [15] for its implementation of SVD. During a preliminary performance analysis, we discovered that the C++ implementation is 3.4X faster than Java. Despite outperforming Java, the C++ implementation barely manages to achieve 4 FPS: which is much lower than the expected QoS target of 30 FPS. To achieve such high levels of performance, we have two options: 1) make better use of the available hardware resources, and/or 2) use a hardware accelerator.

[1]Even if this failed, it came within 5 Units of Last Place (ULP).

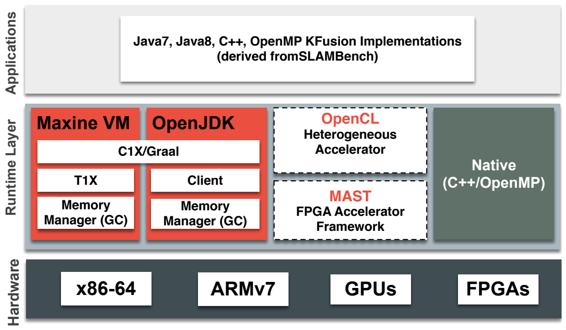


Figure 3. Architecture overview.

3 Heterogeneous Managed Runtime Systems

One of the key aspects of this work is to demonstrate that computationally intensive applications can be created in a hardware agnostic manner; they can be written once and run everywhere while achieving their performance targets. Therefore, we explore application performance over a wide range of devices ranging from desktop to embedded. Regarding low-power architectures, we experiment with both an industrial-strength and a research virtual machine; OpenJDK JVM [30] and Maxine VM [36] respectively.

After achieving a baseline, yet slow, Java implementation of the KF application, our next objective was to optimize it by accelerating its kernels using a range of hardware accelerators. We followed two approaches for acceleration: 1) general purpose acceleration where we provide a framework to re-compile the application for a target accelerator, such as a GPGPU and, 2) an application specific approach where key kernels are off-loaded onto an FPGA. Since no available production-quality JVM supports either GPGPU[2] or FPGA execution, out-of-the-box, we implemented: 1) an OpenCL accelerator for GPGPU offloading, and 2) an FPGA compatible library for dynamic offloading through the JVM.

The following subsections describe in detail the subsystems of the experimental infrastructure depicted in Figure 3: Section 3.1 presents the MREs while Sections 3.2 and 3.3 discuss the OpenCL and FPGA components respectively.

3.1 Maxine Research Virtual Machine

One of our key objectives is to enable JVM research on low-power ARM architectures. Therefore, besides using the production quality OpenJDK JVM we opted for a research JVM also. To overcome the lack of research-based JVMs on ARM systems, we have ported Maxine VM onto ARMv7 32 bit architectures[3].

The Maxine VM, a meta-circular Java-in-Java VM developed by Oracle Labs, has been adopted and augmented in the context of this paper. Since its last release from Oracle, we have enhanced it both in performance and functionality. In detail, the latest release of Maxine from Oracle had the following three compilers: 1) T1X: a fast template-based interpreter (stable), 2) C1X: An optimizing SSA-based JIT compiler (stable), and 3) Graal: an aggressively optimizing SSA-based JIT compiler. Furthermore, the Maxine VM could execute only on x86-64 bit architectures resulting in many of its part to be tightly coupled for 64 bit architectures.

Since our main objective is to provide to the community a state-of-the-art research JVM for both x86 and low-power ARM architectures we enhanced Maxine VM as follows:

1. T1X: Added profiling instrumentation enabling more aggressive profile-guided optimizations applicable to all underlying architectures.
2. T1X: Compiler ports to ARMv7 and Aarch64 enabling experimentation on low-power 32bit and 64bit architectures.
3. C1X: Compiler port to ARMv7 enabling experimentation on low-power ARM 32bit architectures.
4. Graal: Stability and performance improvements.
5. Maxine: Complete ARMv7 support, stability, and performance enhancements.

Furthermore, the work on providing an ARM-compatible research JVM entailed a number of modifications to the original Maxine VM implementations such as: 1) addition of an extra word in object headers in order to store hash codes, 2) a complete re-design of the locking schemes to accommodate for both `thin` and `thick` locks, and 3) augmentation of the register allocator to account for dual-register allocation for long and double types.

To provide a baseline performance comparison between Maxine and OpenJDK JVM we tested both JVMs on x86 and ARMv7 architectures. Figures 4 and 5 illustrate their performance differences on Dacapo9.12-bach [7] and SpecJVM2008 [33] respectively. Unfortunately, we could not compare against JikesRVM [1] since it can not run the Dacapo9.12-bach benchmarks on x86-64.

Regarding x86-64, as illustrated in Figure 4, since Oracle's last release (Maxine-Graal-rev.20290 Original), performance has been increased by 64%[4] (Maxine-Graal-rev.20381 Current). Although Maxine's performance is half of the of industrial strength OpenJDK, which uses the more performant C2 and Graal (rev. 21075) compilers, our goal is to drive up performance until Maxine is on par with OpenJDK. Primarily, this will be achieved by enabling more aggressive Graal optimizations in Maxine such as escape analysis [34] and other compiler intrinsics.

[2]IBM's J9 JVM supports GPGPU acceleration but does not provide enough functionality to run SLAMBench.
[3]An AArch 64 bit port of Maxine VM is also underway. Furthermore, both ports will be open-sourced.

[4]Intel(R) Core(TM) i7-4770@3.4GHz, 16GB RAM, Ubuntu 3.13.0-48-generic, 16 iterations, 12GB heap.

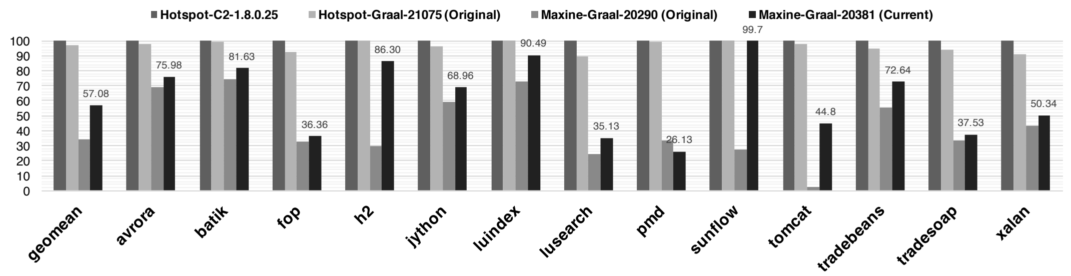


Figure 4. DaCapo-9.12-bach benchmarks (higher is better) normalized to Hotspot-C2-1.8.0.25, x86-64bit.

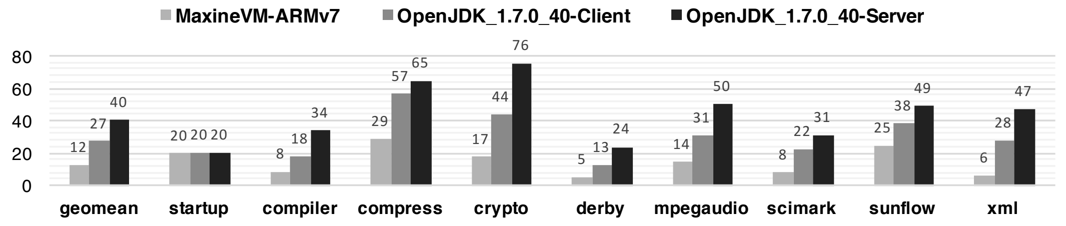


Figure 5. SpecJVM2008 benchmarks (higher is better) normalized to OpenJDK-Zero-IcedTea6_1.13.11, ARMv7-32bit.

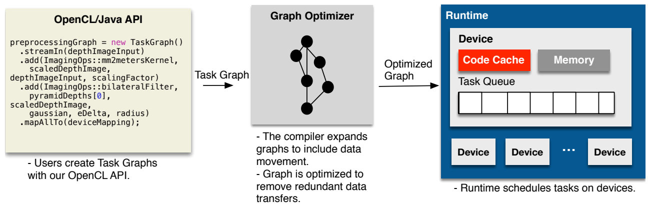
Regarding ARMv7, as depicted in Figure 5[5] the performance of Maxine VM falls between the performance of OpenJDK-Zero and OpenJDK-1.7.0-(Client, Server). Maxine VM outperforms OpenJDK-Zero by 12x on average across SpecJVM2008, while it is on average around 2.3x and 3.3x slower than the OpenJDK-1.7.0 client and server compilers respectively.

3.2 General Purpose OpenCL Acceleration

To improve the productivity of the developers, targeting heterogeneous hardware, we designed and developed an OpenCL accelerator. The key difference between the proposed OpenCL Accelerator and existing programming languages and frameworks is its dynamism; as such, developers do not need to make a priori decisions about their hardware targets. To achieve this, our framework exploits the new JVMCI (Java Virtual Machine Compiler Interface) [22] capabilities to Just-In-Time (JIT) compile Java bytecode to execute on OpenCL compatible devices.

As depicted in Figure 6, our API provides developers with a task-based programming model. A task can be thought of as being analogous to a single OpenCL kernel execution. This means that a task must encapsulate the code it needs to execute, the data it should operate on, and some meta-data. The meta-data can contain information such as the device it should execute on or profiling information. The mapping between tasks and devices is done at a task-level granularity; each task is capable of being executed on a different piece of hardware. These mappings can be provided either by the developer or by a runtime component.

[5]Samsung Chromebook, Exynos 5 Dual@1.7GHz, 2GB RAM, Ubuntu 3.8.11, 2GB heap. `Serial` was excluded from the evaluation.

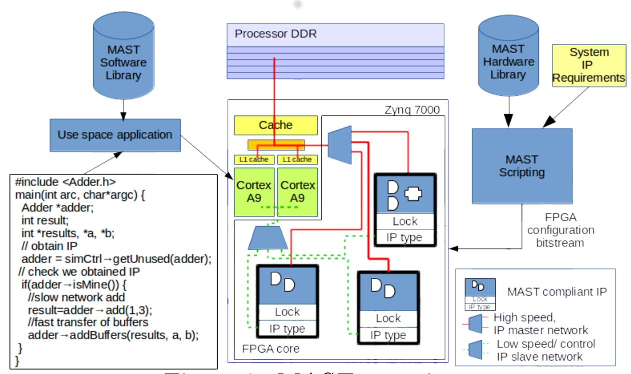

OpenCL/Java API
Task Graph
Graph Optimizer
Optimized Graph
Runtime
Device
Code Cache
Memory
Task Queue
Device
Device
...
Device
- Users create Task Graphs with our OpenCL API.
- The compiler expands graphs to include data movement.
- Graph is optimized to remove redundant data transfers.
- Runtime schedules tasks on devices.

```
preprocessingGraph = new TaskGraph()
  .streamIn(depthImageInput)
  .add(ImagingOps::mm2metersKernel,
    scaledDepthImage,
depthImageInput, scalingFactor)
  .add(ImagingOps::bilateralFilter,
    pyramidDepths[0],
scaledDepthImage,
    gaussian, eDelta, radius)
  .mapAllTo(deviceMapping);
```

Figure 6. OpenCL Accelerator outline.

Instead of focusing on scheduling individual tasks, we allow developers to combine multiple tasks together to form a larger schedulable unit of work (called a task-graph). This approach has a number of benefits: firstly, it provides a clean separation between the code which co-ordinates tasks execution and the code which performs the actual computation; and secondly, it allows the runtime system to exploit a wider range of runtime optimizations. For instance, the task-graph provides the runtime system with enough information to determine the data dependencies between tasks. By using this knowledge, the runtime system is able to exploit any available task parallelism by overlapping the execution of task execution and data movement. It also provides the runtime system with the ability to eliminate any unnecessary data transfers that would occur because of read-after-write data dependencies between tasks.

To increase developer productivity, we make offloading computation as transparent as possible. This is achieved via the runtime system which is able to automatically schedule data transfers between devices and handle the asynchronous execution of tasks. Moreover, the JIT compiler provides support for user-guided parallelization. As a result, the developers are able to rapidly develop portable heterogeneous applications which can exploit any OpenCL compatible device in the system.



```
#include <Adder.h>
main(int arc, char*argc) {
  Adder *adder;
  int result;
  int *results, *a, *b;
  // obtain IP
  adder = simCtrl→getUnused(adder);
 // check we obtained IP
  if(adder→isMine()) {
    //slow network add
    result=adder→add(1,3);
    //fast transfer of buffers
    adder→addBuffers(results, a, b);
 }
}
```

Figure 7. MAST overview.

3.3 Application Specific FPGA Acceleration

As depicted in Figure 3, we target a variety of hardware platforms and therefore significant effort is being placed in providing the appropriate support for the compilers and runtimes of choice. Besides targeting conventional CPU/GPU systems, it is also possible to target FPGA systems such as the Xilinx Zynq ARM/FPGA System on Chip (SoC). In order to efficiently program FPGAs using high level programming languages, we developed MAST: a Modular Acceleration and Simulation Technology (Figure 7).

MAST is a C++ software library, combined with Bluespec [3] hardware IP library, and tools designed to allow the rapid development of flexible hardware accelerators on Xilinx Zynq SoCs. MAST allows easy integration of accelerators into software systems, without the need to worry about drivers. This allows the decoupling of hardware and software engineers, allowing them to concentrate on hardware and user-space software development respectively. For systems exclusively containing MAST components, Xilinx Vivado scripting allows for the automatic implementation of systems from Verilog netlist to bitstream without user intervention. This allows software engineers to deploy new hardware configurations without the requirement to learn complex EDA tools and device specific features. The software library implements the `SimCtrl` controller: a module which allows the discovery of MAST compliant IP on the FPGA at run time. Any IP block can then be locked, at a thread or process level, or reserved by a process for future locking; protection is provided against IP being locked by terminated tasks. Users can request specific IP from the `SimCtrl` and it will, assuming availability, return a `SimObject` which allows them to manipulate the IP block using simple register access. MAST supports IP master transfers allowing memory access from the processor system memory. This typically operates via a coherency port, ACP in the case of Zynq 7000, allowing arbitrary pages of the parent processes to be read or written to from hardware. In this case, the hardware accelerator acts as a "virtual thread" being set off and synchronized at a later date whilst the processor continues operating on other tasks. The availability of both master and slave interfaces on IP allows for simple, flexible, and high performance links between the MAST software and IP libraries. Usually, for a specific IP block, the user will derive a new Class from the `SimObject`, allowing more complex operations at a higher level of abstraction from the hardware. Such abstraction is typically achieved with a few lines of simple code.

Opt Module	1: GPU Acceleration OpenJDK, Graal	2: FPGA Acceleration OpenJDK, Maxine
	Hardware	
CPU	Intel Xeon E5-2620 @ 2GHz	Xilinx Zynq 706 board ARMv7 Cortex A9
Cores	12 (24 Threads)	2
L1	32KB per core, 8-way	32KB per core
L2	256KB per core, 8-way	512KB per core
L3	15MB, 20-way	-
RAM	32GB	1GB
GPU	NVIDIA Tesla K20m @ 0.705GHz, OpenCL 1.2	- -
Ext.	OpenCL Accel.	MAST FPGA
	Software	
JVM	OpenJDK, Graal -	Maxine ARMv7 OpenJDK_1.7.0_40
OS	CentOS 6.8 (Kernel 2.6.32)	Linux 3.12.0-xilinx-dirty

Table 2. Hardware and Software configurations.

The hardware IP library consists of a set of parameterized IP blocks for performing not only basic tasks but also complex high level libraries. The low level blocks handle the IP / software interface, taking care of FPGA bus protocols, device discovery, locking and high speed memory interfaces. The higher level modules currently include memory system models and processor timing models for gathering statistics from either static execution traces or dynamically instrumented applications.

4 Evaluation

The following subsections describe the acceleration and optimization techniques applied to Kinect Fusion (KF) along with the experimental results. The hardware and software configurations for each optimization are shown in Table 2.

4.1 GPGPU Acceleration

GPGPU acceleration has been applied to KF through our OpenCL accelerator (Section 3.2). All, but one[6], kernels of KF have been dynamically compiled and offloaded for GPGPU execution through OpenCL code emission. Figures 8 and 9, illustrate the performance and speedup of the accelerated KF version respectively.

As depicted in Figure 8, the original validated version of Kinect Fusion can not meet the QoS target of

[6]Acquisition can not be accelerated because the input is obtained serially.

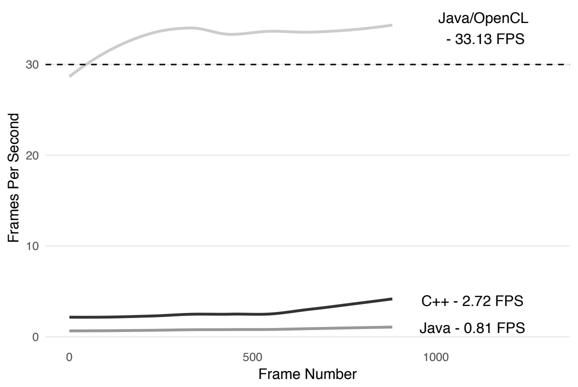


Figure 8. FPS of Java/OpenCL versus baseline Java and C++.

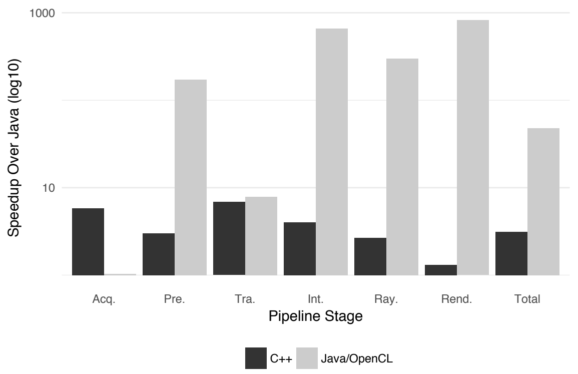


Figure 9. Java/OpenCL and C++ speedup over serial Java per KF stage.

real-time Computer Vision applications (0.71 FPS on average). Both the serial versions of Java and C++ perform under 3 FPS with the C++ version being 3.4x faster than Java. By accelerating KF through GPGPU execution we manage to achieve a constant rate of over 30 FPS (33.13 FPS) across all frames (882) from the ICL-NUIM dataset [17] (Room 2 configuration). In order to achieve 30 FPS, all kernels have been accelerated by up to 821.20x with an average of 47.84x across the whole application, as depicted in Figure 9. By utilizing our OpenCL acceleration infrastructure, we manage to dynamically accelerate a simple un-optimized serial Java version of a KF algorithm meeting its QoS requirements in a transparent to the developer manner.

4.2 FPGA Acceleration

FPGA acceleration has been applied to KF through the MAST acceleration functionality of our platform (Section 3.3). In our initial investigation into FPGA acceleration we selected the `preprocessing` stage for acceleration. This stage contains two computational kernels that: i) scale the depth camera image from `mm` to `meters`, and ii) apply a bilateral filter to produce a filtered scaled image. In particular, a filter is applied to the scaled image in order to reduce the effects of noise in depth camera measurements.

VM	No FPGA Acceleration	With FPGA Acceleration	Speedup
Maxine VM	2.20	0.05	43x
OpenJDK	0.66	0.03	22x

Table 3. Performance and speedup of KF's pre-processing stage with and without FPGA acceleration (mean execution time, in seconds, over 78 frames).

In order to improve the execution time in Java, we merged the two routines into a single routine reducing the streaming of data to and from the FPGA device. The offloading to the FPGA is accomplished by using the Java Native Interface (JNI) mechanism to interface with our MAST module (Section 3.3). The JNI stub extracts C-arrays of floating point values from the Java environment that represent the current input raw depth image from the camera, and the current output scaled filtered image. The JNI stub, in turn, converts the current raw depth image into an array of short integers which is memory allocated (through `malloc`) on the first execution of the JNI stub. The FPGA hardware environment is also initialized during first execution, and consequently the hardware performs the merged scaling and filtering operation. As a result, subsequent executions only need to perform a call to extract C-arrays and to, finally, release the output scaled and filtered image array back to the Java environment. The computational kernels selected for FPGA execution have been developed in Bluespec System Verilog [3] and synthesized on the Xilinx Zynq 706 board.

As depicted in Table 3, FPGA acceleration of the selected kernels improves their performance by 43x and 22x on MaxineVM and OpenJDK respectively. The difference in both execution times and speedups from both VMs stem from the fact that OpenJDK produces more optimal code than MaxineVM (Section 3.1). Unfortunately, we could not combine both techniques to provide an end-to-end evaluation having simultaneous acceleration on FPGAs and GPGPUs because we could not get access to a system with both GPGPU and FPGA accelerators.

5 Related Work

Several related works proposed the exploitation of heterogeneous hardware from dynamic languages. The majority of them target GPGPUs, although attempts have also been for FPGAs, vector units, and multi-core processors. Amongst the targeted programming languages are Java [2, 4, 12, 14, 16, 19, 23, 25, 31, 37, 38], Python [5, 9, 24, 32], Haskell [11, 20, 26], Scala [10, 29], MATLAB [5, 13], and JavaScript [21].

To the best of our knowledge, this paper describes the most complex application to be written entirely in

Java and accelerated by GPGPUs to date. Our OpenCL accelerator differs from prior work by: 1) not using a super-set of the Java language [4, 19], 2) not using ahead-of-time compilation [14, 31], 3) not requiring developers to write heterogeneous code in another language [23], and 4) not requiring manual parallelization of kernels [2].

6 Conclusions and Future Work

In this paper, we showcased that it is possible to use a high-level language such as Java in order to implement complex Computer Vision applications. We extended our research to both industrial-strength and research Java Virtual Machines along with desktop and embedded systems. Also, we managed to accelerate the Kinect Fusion application by up to 47x achieving over 30 FPS with the use of GPGPUs and FPGAs.

Our next steps are to unify our OpenCL accelerator and the MAST technology in order to transparently offload on GPGPUs and FPGAs under the same executions. Finally, recent hardware such as Intel's Xeon and FPGA systems will enable us to unify both acceleration domains to provide Java a high-performing out-of-the-box acceleration suite.

Acknowledgments

This work is partially supported by EPSRC grants Anyscale EP/L000725/1, PAMELA EP/K008730/1, DOME EP/J016330/1, and EU Horizon 2020 ACTiCLOUD 732366 grant. Rodchenko is funded by a Microsoft Research PhD Scholarship, and Luján is funded by a Royal Society University Research Fellowship.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 2000. The JalapeñO Virtual Machine. IBM Systems Journal (2000).

[2] AMD-Aparapi. 2017. http://developer.amd.com/tools-and-sdks/heterogeneous-computing/aparapi/. (Feb. 2017).

[3] Arvind. 2003. Bluespec: A Language for Hardware Design, Simulation, Synthesis and Verification Invited Talk. In Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '03). IEEE Computer Society, Washington, DC, USA, 249–. http://dl.acm.org/citation.cfm?id=823453.823860

[4] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10). ACM, New York, NY, USA, 89–108. DOI: http://dx.doi.org/10.1145/1869459.1869469

[5] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: a CPU and GPU Math Expression Compiler. In Proceedings of the Python for Scientific Computing Conference (SciPy).

[6] P. J. Besl and H. D. McKay. 1992. A method for registration of 3-D shapes. IEEE Transactions on Pattern Analysis and Machine Intelligence 14, 2 (Feb 1992), 239–256.

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications. ACM Press.

[8] J. Butzke, K. Daniilidis, A. Kushleyev, D. D. Lee, M. Likhachev, C. Phillips, and M. Phillips. 2012. The University of Pennsylvania MAGIC 2010 multi-robot unmanned vehicle system. Journal of Field Robotics 29, 5 (2012), 745–761.

[9] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. 2011. Copperhead: Compiling an Embedded Data Parallel Language. In Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11). ACM, New York, NY, USA, 47–56. DOI: http://dx.doi.org/10.1145/1941553.1941562

[10] Olivier Chafik. 2017. ScalaCL: Faster Scala: optimizing compiler plugin + GPU-based collections (OpenCL). (Feb. 2017). Retrieved December 20, 2018 from http://code.google.com/p/scalacl

[11] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell Array Codes with Multicore GPUs. In Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming (DAMP '11). ACM, New York, NY, USA, 3–14. DOI: http://dx.doi.org/10.1145/1926354.1926358

[12] James Clarkson, Christos Kotselidis, Gavin Brown, and Mikel Luján. 2017. Boosting Java Performance using GPGPUs. In Proceedings of the 30th International Conference on Architecture of Computing Systems (ARCS '17).

[13] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In BigLearn, NIPS Workshop.

[14] Georg Dotzler, Ronald Veldema, and Michael Klemm. 2010. JCudaMP. In Proceedings of the 3rd International Workshop on Multicore Software Engineering. DOI: http://dx.doi.org/10.1145/1808954.1808959

[15] EJML. 2017. (Feb. 2017). Retrieved December 20, 2018 from http://ejml.org

[16] Juan José Fumero, Michel Steuwer, and Christophe Dubach. 2014. A Composable Array Function Interface for Heterogeneous Computing in Java. In Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14). ACM, New York, NY, USA, 44:44–44:49. DOI: http://dx.doi.org/10.1145/2627373.2627381

[17] A. Handa, T. Whelan, J.B. McDonald, and A.J. Davison. 2014. A Benchmark for RGB-D Visual Odometry, 3D Reconstruction and SLAM. In ICRA.

[18] A. Handa, T. Whelan, J.B. McDonald, and A.J. Davison. 2014. A Benchmark for RGB-D Visual Odometry, 3D Reconstruction and SLAM. In IEEE Intl. Conf. on Robotics and Automation, ICRA. Hong Kong, China.

[19] Akihiro Hayashi, Max Grossman, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2013. 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. DOI: http://dx.doi.org/10.1145/2500828.2500840

[20] Sylvain Henry. 2013. ViperVM: A Runtime System for Parallel Functional High-performance Computing on Heterogeneous Architectures. In Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC '13). ACM, New York, NY, USA, 3–12. DOI: http://dx.doi.org/10.1145/2502323.2502329

[21] Stephan Herhut, Richard L. Hudson, Tatiana Shpeisman, and Jaswanth Sreeram. 2013. River Trail: A Path to Parallelism in JavaScript. In Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications (OOPSLA '13). ACM, New York, NY, USA, 729–744. DOI: http://dx.doi.org/10.1145/2509136.2509516

[22] JEP 243: Java-Level JVM Compiler Interface. 2017. http://openjdk.java.net/jeps/243. (Feb. 2017).

[23] Java bindings for OpenCL. 2017. (Feb. 2017). Retrieved December 20, 2018 from http://www.jocl.org/

[24] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation. Parallel Comput. 38, 3 (March 2012), 157–174.

[25] Christos Kotselidis, Andrey Rodchenko, Colin Barrett, Andy Nisbet, John Mawer, Will Toms, James Clarksonand Cosmin Gorgovan, Amanieu d'Antras, Yaman Cakmakci, Thanos Stratikopoulos, Sebatian Werner, Jim Garside, Javier Navaridas, Antoniu Pop, John Goodacre, , and Mikel Luján. 2016. Project Beehive: A Hardware/Software Co-designed Stack for Runtime and Architectural Research. In Proceedings of the 9th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG '16).

[26] Geoffrey Mainland and Greg Morrisett. 2010. Nikola: Embedding Compiled GPU Functions in Haskell. In Proceedings of the Third ACM Haskell Symposium on Haskell (Haskell '10). ACM, New York, NY, USA, 67–78. DOI: http://dx.doi.org/10.1145/1863523.1863533

[27] Luigi Nardi, Bruno Bodin, M. Zeeshan Zia, John Mawer, Andy Nisbet, Paul H. J. Kelly, Andrew J. Davison, Mikel Luján, Michael F. P. O'Boyle, Graham Riley, Nigel Topham, and Steve Furber. 2015.. Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. In ICRA.

[28] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. 2011. KinectFusion: Real-time Dense Surface Mapping and Tracking. In Proceedings of the 2011 10th IEEE International Symposium on Mixed and Augmented Reality (ISMAR '11). IEEE Computer Society, Washington, DC, USA, 127–136. DOI: http://dx.doi.org/10.1109/ISMAR.2011.6092378

[29] Nathaniel Nystrom, Derek White, and Kishen Das. 2011. Firepile: Run-time Compilation for GPUs in Scala. In Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering (GPCE '11). ACM, New York, NY, USA, 107–116. DOI: http://dx.doi.org/10.1145/2047862.2047883

[30] OpenJDK. 2017. http://openjdk.java.net/. (Feb. 2017).

[31] P.C. Pratt-Szeliga, J.W. Fawcett, and R.D. Welch. 2012. Rootbeer: Seamlessly Using GPUs from Java. In Proceedings of 14th International IEEE High Performance Computing and Communication Conference on Embedded Software and Systems. DOI: http://dx.doi.org/10.1109/HPCC.2012.57

[32] Alex Rubinsteyn, Eric Hielscher, Nathaniel Weinman, and Dennis Shasha. 2012. Parakeet: A Just-in-time Parallel Accelerator for Python. In Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism (HotPar'12). USENIX Association, Berkeley, CA, USA, 14–14.

[33] SpecJVM2008. 2017. https://www.spec.org/jvm2008/. (Feb. 2017).

[34] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14). ACM, New York, NY, USA, 165:165–165:174. DOI: http://dx.doi.org/10.1145/2544137.2544157

[35] Tango. 2017. (Feb. 2017). Retrieved December 20, 2018 from https://get.google.com/tango/

[36] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. ACM Trans. Archit. Code Optim. (January 2013).

[37] Yonghong Yan, Max Grossman, and Vivek Sarkar. 2009. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In Euro-Par 2009 Parallel Processing, Henk Sips, Dick Epema, and Hai-Xiang Lin (Eds.), Vol. 5704. Springer Berlin Heidelberg.

[38] 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 (GPGPU-5). ACM, New York, NY, USA, 74–83. DOI: http://dx.doi.org/10.1145/2159430.2159439

[39] Zhengyou Zhang. 1994. Iterative Point Matching for Registration of Free-form Curves and Surfaces. Int. J. Comput. Vision 13, 2 (Oct. 1994), 119–152.