Sesame: A User-Transparent Optimizing Framework for Many-Core Processors

Jianbin Fang, Ana Lucia Varbanescu, Henk Sips
Parallel and Distributed Systems Group
Delft University of Technology, Delft, the Netherlands
Email: {j.fang, a.l.varbanescu, h.j.sips}@tudelft.nl

Abstract—With the integration of more computational cores and deeper memory hierarchies on modern processors, the performance gap between naively parallelized code and optimized code becomes much larger than ever before. Very often, bridging the gap involves architecture-specific optimizations. These optimizations are difficult to implement by application programmers, who typically focus on the basic functionality of their code. Therefore, in this thesis, I focus on answering the following research question: “How can we address architecture-specific optimizations in a programmer-friendly way?” As an answer, I propose an optimizing framework for parallel applications running on many-core processors (Sesame). Taking a simple parallelized code provided by the application programmers as input, Sesame chooses and applies the most suitable architecture-specific optimizations, aiming to improve the overall application performance in a user-transparent way. In this short paper, I present the motivation for designing and implementing Sesame, its structure and its modules. Furthermore, I describe the current status of Sesame, discussing our promising results in source-to-source vectorization, automated usage of local memory, and autotuning for implementation-specific parameters. Finally, I discuss my work-in-progress and sketch my ideas for finalizing Sesame’s development and testing.

Index Terms—Optimization, Many-Core Processors, OpenCL.

I. INTRODUCTION

In recent years, more and more many-core processors are superseding sequential ones. Increasing parallelism, rather than increasing clock rate, has become the primary engine of processor performance growth, and this trend is likely to continue [1]. Particularly, today’s GPUs (Graphic Processing Units) and Intel’s MIC (Many Integrated Cores), greatly outperforming traditional CPUs in arithmetic throughput and memory bandwidth, can use hundreds of parallel processor cores to execute tens of thousands of threads. However, programming many-cores is difficult, as it is a problem with multiple constraints: we want applications to deliver great performance, to be easy to program, and to be portable between architectures [2].

A. Motivation and Problem Statement

We measured the performance of parallelized code in two cases: (1) using OpenCL, a novel programming model, and (2) using OpenMP, a traditional programming model, on many-core processors. In [3], we have shown that OpenCL can achieve similar performance to CUDA on NVIDIA GPUs under a fair comparison. However, when testing our solutions on different architectures (e.g., AMD Radeon HD5870, Intel Core i7 920, and Cell/B.E.), we observed that although OpenCL does insure code portability on multiple families of devices, it has poor performance. The performance portability is hindered by platform-specific optimizations, as also confirmed in [4].

Working with OpenMP, a traditional shared memory programming model [5], we had similar results on Intel MIC. Using additional pragmas, OpenMP has been extended to support programming on many-core processors (like Intel MIC). Thus, the legacy code parallelized in OpenMP can run on the many-core processors with minor modifications. For example, we ported BT-MZ (the multi-zone version of the Block Tri-diagonal solver in the NAS Parallel Benchmark [6]) written in OpenMP onto Intel MIC. We got the performance illustrated in Figure 1. The maximum obtained performance is around 30 GFLOPS. We assume it is the architecture disparities (with the traditional CPUs on cache hierarchies and processing core inter-connection) that lead to the 3% of MIC’s theoretical peak performance (1TFLOPS).

Given all that, the poor performance of the parallelized code is due to the fact that the optimizations we apply on one platform are not portable on another, i.e., poor performance-portability. This phenomenon occurs both when using a novel dedicated programming model (e.g., OpenCL) and when using a traditional programming model (e.g., OpenMP). Typically, when implementing an application on a many-core processor,
we will experience two steps: porting and tuning/optimizing. Therefore, we will first get a naively parallelized code: a prototype that is functionally correct, but without any platform-specific optimizations. After performing a series of code transformations, the prototype will evolve into an (maximally) optimized version. Accordingly, we often do one of the two things: keep the implementation simple, at the naive level, or make some optimizations which end up as platform specific implementation. This leads to partially optimized, partially portable code, and fairly low productivity.

To tackle this issue, we need to (1) make a clearer separation between functional parallel (naive) code and optimized code, (2) identify all the relevant optimizations for the state-of-the-art many-core processors, and (3) find a way to automatically apply them. In the thesis, I propose a generic optimizing framework for many-core processors (Sesame), ensuring high productivity by letting naive code stay portable and high performance by performing code specializations.

B. Optimizing Framework: Sesame

Figure 2 shows the Sesame optimizing framework and its modules. Taking naive code as input, Sesame performs code transformations and generates specialized kernels for different platforms. The Sesame framework consists of four components: (1) feature identifier, (2) impact predictor, (3) source-to-source translator, and (4) auto-tuner.

The feature identifier finds the architectural features that are sensitive to application performance from the state-of-the-art many-core processors. The impact predictor takes application information (e.g., memory access patterns) abstracted from naive kernels and quantifies the performance impact of these key features. The predictor can provide essential information on whether using such architectural features will benefit the overall performance. Once we know it is beneficial, a source-to-source translator transforms the naive code into a parameterized format with a simple mechanism (resembling on/off switches) for easy enabling/disabling the usage of certain architectural features. Further, programmers can perform more optimizations (e.g., implementation-related optimizations) on these optimized kernels. These switches together with the implementation-related parameters make up an Sesame optimization space. Using an auto-tuner, we find the optimal solution within this optimization space for a given target many-core processor.

After 24 months of my PhD career, I have implemented two modules in Sesame: S2S Vectorizer and ELMO (shown in Section IV). We are working on Cache-Tiler to make proper use of caches on many-core processors. Our Sesame framework is scalable and it can integrate more modules when new architectural features are identified.

The clash between productivity/portability and performance is not new in the multi-/many-core world. In fact, multiple approaches have been proposed to improve productivity while achieving high performance for many-core processors, which can be loosely classified into (i) new languages (e.g., OptiML [7]), (ii) auto-parallelizing compilers (e.g., OpenACC [8]), and (iii) libraries/APIs (e.g., Thrust [9]). In all these cases, programmers are isolated, in one way or another, from the difficult implementation details related to the platform architecture: they can focus on the functional parts of the application and leave these non-functional elements to be solved by run-times, compilers, or libraries. Different from the prior work, Sesame brings us with four new ideas:

- It works at a medium level of abstraction.
- It defines the boundaries of naive kernel implementations with optimized implementations.
- It estimates performance impact of key features.
- It automatically applies a mix of relevant optimizations to generate an efficient platform-tuned implementation.

II. Programming Model Selection

We investigate three (types of) programming models for many-core processors, already mentioned in Section I-A: OpenCL, CUDA, and OpenMP-like programming models.

OpenCL, managed by the Khronos Group [10], is a framework that allows parallel programs to be executed across various platforms. As a result, OpenCL can give software developers portable and efficient access to the power of diverse processing platforms. By comparison, CUDA is an NVIDIA-specialized programming model. Based on C, CUDA uses language extensions for separating device (i.e., GPU) from host code and data, as well as for launching CUDA kernels.

OpenMP-like programming models consist of directives, library routines, and designated compilers [11]. These models use a set of directives to provide useful information to the underlying compilers. For example, OpenACC [8] uses a collection of compiler directives to specify loops and regions of code in standard C, C++, and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators. Likewise, OmpSs [12] is an effort to integrate features from the StarSs programming model developed by BSC into a single programming model, based on extending OpenMP with new directives to support asynchronous parallelism and heterogeneity (devices like GPUs). Still, all the OpenMP-like languages are too high level and will hide too many of the parallelization decisions that need to be fully exposed to the programmers. Therefore, we consider they are less fit to allow for lower level optimizations like the ones we propose in our framework.
In our previous work, we thoroughly investigated the performance of OpenCL, versus CUDA and OpenMP [3], [4]. We showed that despite some performance gaps, OpenCL can give comparable performance with CUDA or OpenMP after we performed both the right optimizations related to the architecture features. Thus, OpenCL is a good alternative to CUDA and OpenMP in terms of performance.

Overall, I argue that using a unified programming model for all target architectures is beneficial for this work, as it allows Sesame to start from a common code base and machine representation. Further, as we require a level of abstraction that exposes the parallelism in the naive implementation, OpenMP-like programming models are not ideal solutions. Still, any directive-based programming model can be implemented on top of OpenCL and be orthogonal to OpenCL in terms of productivity. Therefore, we choose OpenCL as the language for both the input and the output of Sesame.

III. SESAME INPUTS

In this section, I describe the Sesame inputs: naive kernels, platform description, and application information.

A. Naive Kernels

As shown in Section II, current approaches for many-core programming are separated into high-level, user-transparent solutions (OpenMP-like) that hide both parallelism and low-level optimizations, and low-level user-driven solutions (CUDA and OpenCL) that force too many low-level optimizations included in the first prototype of any application.

I believe a solution with medium-level abstractions, that forces programmers to expose the parallelism decisions, but allows enough transparency in avoiding low-level architectural optimizations is necessary for more productivity and efficiency when searching for many-core performance. Therefore, Sesame starts from a naive application implementation, where only functional aspects and parallelism are exposed, and all other optimizations are carefully avoided. Then, Sesame framework will choose and apply the right optimizations for each specific target platform the application needs to run on.

B. Platform Models and Description

I present two (types of) machine models to serve Sesame: a simple platform model (Machine Model-I) for end-users, and multiple derived platforms (Machine Model-II) with specific architectural features for Sesame.

1) Machine Model-I: As illustrated in Figure 3(a), we present a simple machine model (MM-I) similar to the PRAM model. It consists of multiple compute units (CU) that can access the global memory space uniformly. This model can serve for end-users: (1) to design algorithms and analyze their complexity, and (2) to focus on functionality and parallelism.

2) Machine Model-II: Due to the diversity of many-core architectures, we give a set of machine models (MM-II) with the architectural properties that are significant to the overall performance. Examples of such key features are VPUs (vector processing units), on-chip programmer-managed local memory, and user-oblivious caches, and are shown in Figure 3.

We identify these features from studying the state-of-the-art many-core processors. For example, Intel MIC uses wider vectors, and thus VPU plays a key role in the overall performance. Specifically, we use the (micro-)benchmarking approach to measure the performance impact of each architectural feature. Each target platform may have multiple performance-relevant features, and Sesame should be able to perform the corresponding code transformations based on these key features.

C. Application Information

In our work [13], we see that using local memory can significantly decrease execution time for the right applications (e.g., those that have data sharing). Thus, apart from hardware specifics, choosing to perform a hardware-specific optimization also depends on the input application. The application information can be a code template or application characteristics, abstracted from input kernels. Taking this information, the impact predictor calculates the performance gain/loss when using a hardware-related optimization.

IV. SESAME IMPLEMENTATION

Up to now, we have implemented multiple generic optimizations and implementation-related (auto-)optimizations in Sesame. The current status is summarized as follows.

A. Using Local Memory

Local memory is situated on-chip (Figure 3(c)), and it is much faster than the global memory. Thus, a proper use of local memory often leads to higher memory bandwidth. Nevertheless, using local memory is an error-prone and time-consuming process. Programmers often have to manually address, in their code, challenges like (1) geometry mismatch, (2) work-items masking and binding switches, and (3) inefficient local memory organization [14]. We argue that when solving these problems manually, programmers waste too much time on non-functional coding details, which hinders productivity.
1) An Impact Predictor: To tackle these challenges, the ultimate solution is to use automated code transformation, typically in the form of a compiler pass. This code translation consists of two steps: (1) predict the performance benefits of using local memory, and then (2) perform code translation. In the recent past, our work [15] has focused on addressing the issue of unpredictability of performance benefits from using local memory. Thus, we developed a microbenchmark-based approach to quantify the performance impacts of using local memory. We designed and evaluated the benchmarks on typically used platforms, and store the results into a performance database. Based on this database, we develop a query-based impact predictor to indicate the performance gain/loss of using local memory.

2) ELMO: For the second step, i.e., code transformation, we have designed and implemented a high-level API targeting the efficient usage of local memory on modern many-core processors [14]. Specifically, we propose ELMO, a collection of easy-to-use APIs that (1) present a friendly front-end to make the bindings/mappings transparent to users, and (2) provide implementations and perform several optimizations to ensure the efficiency of the local memory operations. Our results show that with ELMO the kernels can run by up to $3.7 \times$ faster than the naive kernels and deliver matching performance with hand-tuned kernels on NVIDIA Quadro5000.

B. S2S Vectorization

Currently, there are multiple many-core architectures that use SIMD cores (Figure 3(b)). For these architectures, vectorization is a mandatory optimization: without it, applications literally waste more than 50% of the processing power. In our work [16], we propose two different vectorization approaches, specifically designed for naive OpenCL code. Essentially, we are using a source-to-source translator that starts from a generic (scalar) kernel and applies step-by-step transformations to obtain a vectorized one. We note interesting performance results: in most cases, vectorization improves the application performance (improvements ranging between 4% and 300%), but can also slow-down certain applications. Thus, VPU is a performance-relevant feature and need to use it carefully for better performance.

C. Implementation-related Auto-tuning

We have also found that optimizations are not only architecture-dependent, but closely relevant to implementation. In [17], we start with the Cluster Data Stream (CDS) algorithm in Rodinia, and focus on optimizing its memory usage. Specifically, we propose a rake-based memory-efficient solution to CDS. The basic idea is to let each work-item work on multiple data elements. Then the question becomes: how many such elements should be assigned to one work-item (the rake-size).

As the rake-size has a significant effect on performance, it is difficult to maximize performance for all problem sizes by setting one fixed value. To address this issue, we present an auto-tuning solution to select the optimal rake-size per platform and problem-size. Experimental results show that our fully optimized implementation can perform $2.1 \times$ and $1.4 \times$ faster than the naive OpenCL implementation on NVIDIA GTX480 and AMD HD5870, respectively; it can also achieve $1.4 \times$ to $3.3 \times$ speedup relative to the original CUDA implementation solution on GTX480.

V. On-going Work and Conclusion

We have implemented two modules in Sesame: the optimized use of local memory and S2S vectorization. Further, I plan to extend the current work based on the following points.

- Identify more performance-changing architectural features. I want to investigate how the non-uniform caches impact the performance on Intel’s MIC, and implement the Cache-Tiler. In the future, I plan to trace the developments of processors and hopefully add more performance features to this list.
- Consider the performing order of code transformation, once we have got a set of key features in the framework.
- Integrate all tools into the full Sesame such that it does the analysis, code transformation, and tuning in one go.

REFERENCES