A Performance Analysis Framework for Optimizing OpenCL Applications on FPGAs

Zeke Wang Bingsheng He
Nanyang Technological University, Singapore

Wei Zhang
HKUST

Shunning Jiang[*]
Cornell University

ABSTRACT

Recently, FPGA vendors such as Altera and Xilinx have released OpenCL SDK for programming FPGAs. However, the architecture of FPGA is significantly different from that of CPU/GPU, for which OpenCL is originally designed. Tuning the OpenCL code for good performance on FPGAs is still an open problem, since the existing OpenCL tools and models designed for CPUs/GPUs are not directly applicable to FPGAs. In the paper, we present an FPGA-based performance analysis framework that can shed light on the performance bottleneck and thus guide the code tuning for OpenCL applications on FPGAs. Particularly, we leverage static and dynamic analysis to develop an analytical performance model, which has captured the key architectural features of FPGA abstractions under OpenCL. Then, we provide four programmer-interpretable metrics to quantify the performance potentials of the OpenCL program with input optimization combination for the next optimization step. We evaluate our framework with a number of user cases, and demonstrate that 1) our analytical performance model can accurately predict the performance of OpenCL programs with different optimization combinations on FPGAs, and 2) our tool can be used to effectively guide the code tuning on alleviating the performance bottleneck.

1. INTRODUCTION

FPGAs have been used as accelerators for a wide range of applications such as high performance computing [19], databases [23], bioinformatics [28] and “big-data". Microsoft has adopted the FPGAs to accelerate the Bing web search engine [26] and Baidu is using Altera FPGAs to accelerate convolutional neural network (CNN) algorithms for deep learning applications. Compared with other accelerators like GPUs, FPGAs usually have much better energy efficiency, and can deliver superb performance for some applications [8]. Despite the energy efficiency and performance advantages, programmability has been a major issue of FPGAs. High-level synthesis (HLS) on FPGAs has attracted decades of efforts to automate the design process: interpreting an algorithmic description in high-level language and then implementing that program on FPGAs [5]. Recently, FPGA vendors such as Altera and Xilinx provide OpenCL SDK as a series of HLS tools to allow users to write OpenCL programs for FPGAs. Altera OpenCL SDK provides the pipeline parallelism technology to simultaneously process data in a massively multithreaded fashion on FPGAs. This is a significant leap on the FPGA programmability, in comparison with low-level programming with hardware description languages (HDL) [1, 8, 13]. Although this paper focuses on Altera OpenCL SDK, our study can be applicable to other vendors like Xilinx, with minor modifications to our analysis framework.

Even with Altera OpenCL SDK, users are still facing the challenge of programming FPGAs with OpenCL efficiently. Although OpenCL has offered a multi-threaded abstraction to programmers, the architecture of FPGA is fundamentally different from that of CPU/GPU. For example, each instruction has its own custom circuit on FPGAs, while instructions share the common pipeline of CPUs/GPUs. Therefore, optimizing OpenCL code on FPGAs requires the awareness of FPGA architecture. On the one hand, there are a series of FPGA-centric optimizations and their combinations. On the other hand, programmers should consider the FPGA resource constraints since different combination consume different amounts of FPGA resources.

If the code tuning on FPGAs was done properly, the performance could be boosted significantly. Consider an example of Matrix multiplication (MM). We implement MM in OpenCL with four different optimizations (namely SM, CU, SIMD and UL). The exact meanings of these four optimizations are not important here, and we present more details about our implementation in Section 6.3. The normalized performance speedup after applying the optimizations and their combinations over baseline on FPGA is shown in Figure 1. The leftmost one is the baseline. The next four show the performance of the kernels with only one optimization. The remaining ones show the speedup of some example optimization combinations (i.e., when one more optimization is applied). The important observation here is, with careful tuning and optimization combinations, the optimized implementation can be over two orders of magnitude faster than the baseline implementation. It shows the importance of understanding to what extent each optimization affects the OpenCL program on FPGA, as well as guiding programmers to choose the optimal optimization combination under the limited resource budget of FPGA.

Existing performance models [3, 20, 29] and code tuning tools [4, 29, 33] are specifically designed and developed for CPU/GPU. They cannot be directly applicable to FPGA.

[*]Shunning’s work was done when he was a visiting student in Nanyang Technological University, Singapore.

978-1-4673-9211-2/16/$31.00 ©2016 IEEE

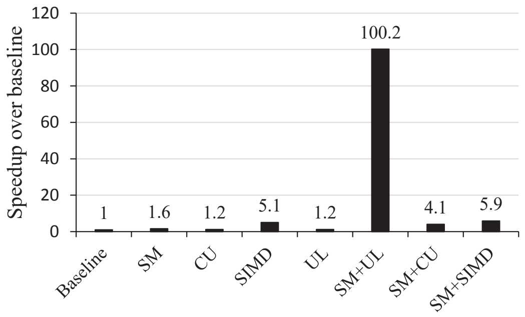


Figure 1: Matrix multiplication (MM) performance speedup over baseline with different optimization combinations

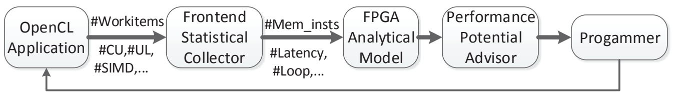


Figure 2: Overview of Performance analysis framework on FPGA

Though there are a few performance models [12, 14] for FPGAs, they are not designed for both performance analysis model and code tuning for OpenCL programs on FPGAs.

In this paper, we propose a performance analysis framework to assist programmers to optimize the OpenCL program on FPGAs. We assume that programmers adopt iterative and incremental development model [22] to tune and optimize their OpenCL programs. At the beginning of each iteration of the software development, our framework quantitatively estimates the performance potentials of the FPGA application via four metrics: global memory potential (B_{mem}), computing potential (B_{comp}), pipeline balance potential ($B_{balance}$), and inter-thread pipeline potential (B_{itpp}). These four metrics suggest the potential performance bottleneck, and then programmers can decide what types of optimizations that should be selected next for further performance optimization.

The performance analysis framework on FPGAs has three components: a frontend data collector, an analytical model, and a performance potential advisor, as shown in Figure 2. First, the frontend data collector takes an OpenCL application as an input and then performs the static statistical collection on the corresponding LLVM IR code as well as dynamic profiling of the OpenCL application execution on GPU. The static and dynamic statistics are fed into our analytical performance model. Second, the analytical model predicts the performance of OpenCL kernel by carefully analyzing and modeling the massive parallel execution mechanism of FPGAs. Third, the performance advisor digests the model information and provides the four potential metrics to understand the performance bottleneck. Thus, programmers are able to use the four potential metrics to diagnose the main performance bottleneck, and then apply the corresponding optimization to the OpenCL kernel.

To evaluate the proposed framework, we apply it to a few OpenCL programs from existing benchmarks [6] or source code from authors [9, 10]. The results show that our framework is able to differentiate the performance impact of individual optimizations and their combinations as well as successfully provide programmers with understandable

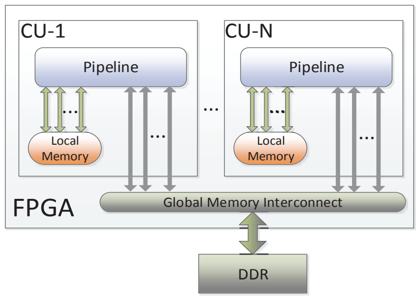


Figure 3: Architectural view of Altera OpenCL SDK

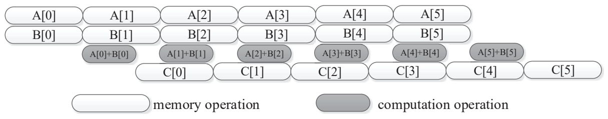


Figure 4: FPGA pipeline of VecAdd (C=A+B)

metrics to identify the performance bottleneck of the OpenCL kernel. Moreover, the optimization guided by our performance framework significantly outperforms that from Altera's resource driven optimization [2].

In summary, this paper makes the following two key contributions. First, we propose a FPGA-centric performance analysis framework that is able to predict the OpenCL kernel performance on FPGA with different optimization combinations. Second, we propose four FPGA-centric metrics to show the performance potentials and then guide the programmer to pinpoint the bottlenecks for OpenCL kernel on FPGAs. To the best of our knowledge, this study is the first of its kind in performance analysis and diagnosis for OpenCL programs on FPGAs.

The remainder of this paper is organized as follows. We give the background and preliminaries in Section 2. We present the details on our data collector in Section 3. In Section 4, we present the design detail of the FPGA-centric performance analysis model. In Section 5, we present the four potential performance metrics. In Section 6, we present our experimental results. Section 7 describes the related work. Finally, we conclude this paper in Section 8.

2. PRELIMINARIES AND BACKGROUND

In this section, we present OpenCL architecture for FPGA and OpenCL execution model on FPGA, followed by an overview of our proposed framework.

2.1 OpenCL Architecture for FPGAs

Altera provides the OpenCL SDK [1] to abstract away the hardware complexities from the FPGA implementation. Figure 3 illustrates the architecture for Altera OpenCL SDK, and their interconnects with global memory and local memory. Different from viewing FPGAs as pure hardware resources such as LUTs, DSP blocks and memory blocks, the OpenCL SDK views the FPGA as a massive parallel architecture. An OpenCL kernel can contain multiple kernel pipelines, *i.e.*, Compute Units (CUs). Each CU implements a massive pipelined execution for the OpenCL program. Figure 4 shows the pipelined execution of a simplified vector addition ($C = A + B$). Memory and computation operations can overlap with each other for efficiency.

From the perspective of OpenCL, the memory component of FPGA computing system contains three layers. First, the *global memory* resides in DDRs of the FPGA board. The accesses to global memory have long latency, and its bandwidth is shared by all the CUs on the FPGA. Second, the *local memory* is a low-latency and high-bandwidth scratchpad memory, and it has four banks for each CU. Third, the *private memory*, storing variables or small arrays for each *work item* (i.e., the basic execution unit in OpenCL), is implemented using completely-parallel registers. Compared to CPU/GPU, the private memory is rather plentiful in FPGA.

Each CU has its own local memory interconnect while all the pipelines share the global memory interconnect. In particular, load/store operations to local memory access from each CU can be combined together to arbitrate for the local memory. In contrast, load/store operations to global memory access from all CUs compete for the on-board global memory bandwidth [1]. The absence of dedicated cache hierarchy further makes the global memory transactions of FPGAs significantly less efficient than those of CPUs/GPUs.

The OpenCL SDK exposes a number of key hardware and software features to programmers so that they are able to perform performance optimizations on the OpenCL kernel.

Local Memory (SM): To alleviate the stalls on the global memory, the local memory can be used to reduce the number of global memory accesses.

Memory Coalescing (MC): To reduce the number of global memory transactions, MC is used to refine the global memory access pattern so that global memory operations from consecutive work items can be coalesced into a single wide memory transaction.

Loop Unrolling (UL): If a large number of loop iterations exist in the kernel pipeline, the loop iterations could potentially be the critical path of the kernel pipeline. UL can increase the pipeline throughput by allocating more hardware resources to the loop. Also, UL might have a side-product benefit on FPGAs. The load/store operations can be coalesced so that the number of global memory transactions is reduced.

Kernel Vectorization (SIMD): To achieve higher throughput of OpenCL kernel execution, SIMD can be applied to translate multiple scalar arithmetic operations to a single vector arithmetic operation. With SIMD, the number of total work items (*#WI*) can be reduced, given the same amount of workloads per work item.

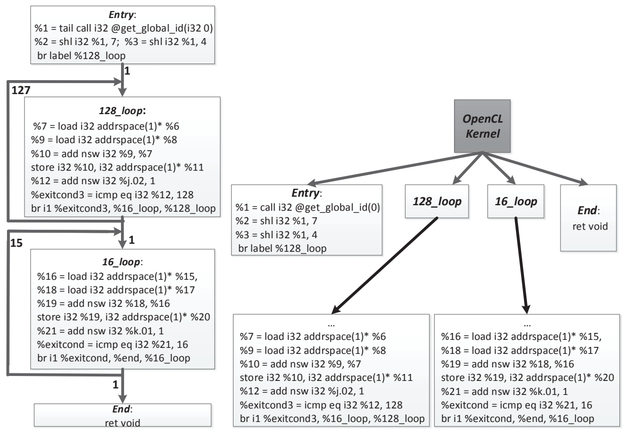
Kernel pipeline replication (CU): If the hardware resource is sufficient on the FPGA, the kernel pipeline can be replicated to generate multiple CUs to achieve higher throughput. The inner hardware scheduler automatically dispatches the work groups among CUs. Since CUs consumes more hardware resources, the operating frequency of FPGA tends to be lower than that of one CU. That means, two CUs cannot always double the performance. Another issue is that the global memory load/store operations from multiple CUs compete for the global memory bandwidth and the total number of global memory operations stays the same.

2.2 OpenCL Execution Model on FPGA

The Altera's compilation framework [13] is based on the LLVM framework. It first translates the OpenCL kernel (.cl file) into the intermediate representation (LLVM IR) via its specific C-Language parser, and then converts the LLVM IR

Algorithm 1: AN EXAMPLE WITH PIPELINE IMBALANCE PROBLEM

```
1 gid ← get_global_id(0);
2 for (j ← 0 to 127) do
3   | A[(gid ≪ 7) + j] ← B[(gid ≪ 7) + j] + C[(gid ≪ 7) + j];
4 end
5 for (k ← 0 to 15) do
6   | D[(gid ≪ 4) + k] ← E[(gid ≪ 4) + k] + F[(gid ≪ 4) + k];
7 end
```



```
%1 = tail call i32 @get_global_id(i32 0)
%2 = shl i32 %1, 7;  %3 = shl i32 %1, 4
 br label %128_loop
```

```
 %7 = load i32 addrspace(1)* %6
%9 = load i32 addrspace(1)* %8
%10 = add nsw i32 %9, %7
store i32 %10, i32 addrspace(1)* %11
%12 = add nsw i32 %j.02, 1
%exitcond3 = icmp eq i32 %12, 128
br i1 %exitcond3, %16_loop, %128_loop
```

```
%16 = load i32 addrspace(1)* %15,
%18 = load i32 addrspace(1)* %17
%19 = add nsw i32 %18, %16
store i32 %19, i32 addrspace(1)* %20
%21 = add nsw i32 %k.01, 1
%exitcond = icmp eq i32 %21, 16
br i1 %exitcond, %end, %16_loop
```

```
ret void
```

```
%1 = call i32 @get_global_id(0)
%2 = shl i32 %1, 7
%3 = shl i32 %1, 4
 br label %128_loop
```

```
ret void
```

```
...
%7 = load i32 addrspace(1)* %6
%9 = load i32 addrspace(1)* %8
%10 = add nsw i32 %9, %7
store i32 %10, i32 addrspace(1)* %11
%12 = add nsw i32 %j.02, 1
%exitcond3 = icmp eq i32 %12, 128
br i1 %exitcond3, %16_loop, %128_loop
```

```
...
%16 = load i32 addrspace(1)* %15,
%18 = load i32 addrspace(1)* %17
%19 = add nsw i32 %18, %16
store i32 %19, i32 addrspace(1)* %20
%21 = add nsw i32 %k.01, 1
%exitcond = icmp eq i32 %21, 16
br i1 %exitcond, %end, %16_loop
```

(a) Control flow graph (b) LLVM hierarchy

Figure 5: Control flow graph and LLVM hierarchy of Algorithm 1

into the Verilog HDL for the OpenCL kernel.

The LLVM IR of the corresponding OpenCL kernel can be represented as a hierarchy named control flow graph (CFG). The CFG consists of *basic blocks* connected by control-flow edges [13]. The basic block consists of a group of instructions in a contiguous sequence. The control flow (e.g., loops and if-else branches) among basic blocks form a CFG. Each edge is associated with a trip count, indicating the number of times that the execution of the basic block is along this control flow. An example OpenCL program with two loops is shown in Algorithm 1, whose LLVM IR is illustrated in Figure 5. In this example, there are four basic blocks (e.g., 128_*loop* and 16_*loop*) in the CFG. The first loop (basic block: 128_*loop*) at Lines 2-3 has a trip count of 128 for each work item (*gid*). The second loop (basic block: 16_*loop*) at Lines 5-6 has a trip count of 16.

Based on the specific properties of OpenCL implementation on FPGA, each instruction in a basic block is implemented with its own dedicated circuit. Thus, each basic block is implemented with the dedicated pipeline, and all the basic blocks in an OpenCL kernel can execute simultaneously when there is sufficient parallelism (in terms of work items) in the whole OpenCL pipeline. In other words, each basic block behaves like the execution of a separate core. That is, the performance of an OpenCL kernel is determined by the performance of the slowest core (basic block). Therefore, for the programmer, it is critical to keep the load balanced among basic blocks of the OpenCL kernel. In the example CFG of Figure 5, the four basic blocks can execute concurrently. For load balancing among the basic blocks, we should put more resources (via UL) in optimizing the first loop, since the performance is determined by the slowest part (the first loop: 128_*loop*).

When developing the performance analysis framework, we have considered three compilation levels for analysis: OpenCL source code, LLVM IR and Verilog HDL. To the end, we choose LLVM IR as the targeted level for the following reasons. First, LLVM IR has the fixed number of IR instructions, with a rich set of static analysis tools available. Second, LLVM IR allows us to pinpoint the bottleneck of the source code for further analysis. In comparison, the analysis of OpenCL source code is very difficult since the pattern of high-level language is flexible. On the other hand, the Verilog HDL level analysis is too low-level, because even the basic operations (e.g. floating point addition) are converted into the large-size Verilog code. Together with the automatic generated control logic, it is very tedious and programmer-unfriendly to map the Verilog code back to the OpenCL code, indicating that it is hard to pinpoint the performance bottleneck to programmers by analyzing the Verilog code.

2.3 Overview of Our Framework

The OpenCL SDK can resolve the FPGA programmability to a large extent. However, it is still a rather open problem on developing performance models and tuning tools to optimize the performance of OpenCL programs on FPGAs. As shown in Introduction, such FPGA-specific optimizations are essential for high performance OpenCL programs on FPGAs. That motivates us to develop a performance analyses framework for optimizing OpenCL applications on FPGAs, as shown in Figure 2. Our framework has combined static and dynamic analysis to offer performance predictions and optimization guides.

The basic idea behind the proposed performance framework is on *iterative and incremental development* in software engineering [22]. In this software development, programmers develop an OpenCL application through repeated cycles on a number of phases including planning, implementation, testing and evaluation. For optimizing the performance of an OpenCL application, programmers are facing many optimization choices, ranging from algorithmic design to architecture-specific optimizations. Most importantly, programmers need to know where the performance bottleneck is in their source code, and whether the applied optimization has already efficiently utilized the hardware resources of FPGA. Therefore, the proposed performance analysis framework facilitates programmers with understandable metrics to identify the performance bottleneck of the OpenCL kernel, and advises them with four potential metrics for further optimizations. That means, our framework mainly focuses on two phases in the iterative and incremental development cycle: the former for evaluation, and the latter for planning. Also, the framework does not pose any restriction on the other two phases: implementation and testing. For example, in order to resolve the memory bottleneck, programmers can implement a completely new memory-efficient algorithm. When programmers adopt architecture-specific optimizations (examples listed in Section 2.1), our framework can guide to select the suitable optimization combination and the parameter setting for each optimization in the combination.

3. INPUT DATA COLLECTORS

From this section on, we present the details of the components in our performance analysis framework. This section presents the details on input data collector, followed by the analytical model and performance potential advisor in Sections 4 and 5, respectively.

The analytical model on OpenCL-based FPGA requires several types of statistics to roughly predict the performance of the input OpenCL kernel and to identify the performance potentials. The data sources include standard inputs (e.g., according to vendor specification or from the programmer), static and dynamic analysis. Their notations are summarized in Table 1. For static and dynamic analysis, we have developed two data collectors: static statistical collector and run-time statistical collector accordingly.

3.1 Standard Inputs

The standard inputs include parameters that can be obtained from calibrations, vendor specification, compilation report and host code.

I_L, I_M and I_D represent the initial resource availability of logic blocks, memory blocks and DSP blocks in FPGA, respectively. Note, some resources are initially used by DDR and PCI-e controller. In our test FPGA, we calibrate the FPGA and obtain I_L, I_M and I_D to be 83%, 89% and 100% respectively.

From the Altera compilation report, we collect the frequency $\#Freq$ and resource utilizations ($\#U_{logic}$, $\#U_{mem}$ and $\#U_{dsp}$) of logic blocks, memory blocks, and DSP blocks, respectively. The resource utilizations are used to estimate the computation potential.

From the hardware specification, we obtain the values for *#Banks* and $\#Mem\_trans\_width$. Their values are 2 and 64 bytes, respectively.

The number of work items (*#WI*) is obtained from the function call of kernel invocation in the host code.

3.2 Static Statistical Collector

Since Altera OpenCL SDK, which generates the intermediate representation (LLVM IR) file, is not open-source, our static statistical collector is implemented mainly on the open-source LLVM 3.4 [11] and Clang.

In the pipelined execution model, all basic blocks can be executed simultaneously. We therefore develop a data structure called *LLVM hierarchy* to capture the pipelined execution model. Given the CFG, we can construct the LLVM hierarchy in a recursive manner. The entry node of the CFG forms the root node of the LLVM hierarchy. Next, we find all the child nodes for the root node. We traverse the CFG by distinguishing node types, and add the child node to the root node from the left to right. For a basic block, we simply add it as a leaf child to the root node. Otherwise, we add an non-leaf child to the root node, and associate it with the relevant information such as trip count. The non-leaf node is further expanded in the later recursion steps. An example of LLVM hierarchy is shown on the right of Figure 5. A leaf node in the LLVM hierarchy corresponds to a basic block in the CFG. Thus, we also call the leaf node of LLVM hierarchy a basic block. In the following, we mainly use the LLVM hierarchy for our analytical model.

We design a built-in pass (based on the existing $llvm::BasicBlock$ class) obtained from the LLVM tool to analyze the static information (including $\#Mem\_insts_i$, $\#Mem\_bytes_i$, $\#Mem\_burst_i$ and $\#Cycle_i$) of the basic block i. Program-

Table 1: Summary of parameters

Model Parameter	Definition	Source
#WI	# of global work items	Standard input
#Mem_trans_width	# of maximum memory transaction width in bytes	Standard input
#Banks	# of number of memory banks	Standard input
#Freq	# of frequency of the OpenCL kernel.	Standard input
$\#U_{logic}, \#U_{mem}, \#U_{dsp}$	# utilization of logic blocks, memory blocks and DSP blocks.	Standard input
I_L, I_M, I_D	The initial resource availability of logic blocks, memory blocks and DSP blocks	Standard input
#CU	# of compute units	Static Analysis
#SIMD	# of vectorization factor	Static Analysis
$\#UL_i$	# of unrolling factor of the LLVM node *i*	Static Analysis
$\#N_B$	# of number of basic blocks	Static Analysis
$\#Mem\_insts_i$	# of memory instructions of the LLVM node *i*	Static Analysis
$\#Cycle_i$	# of cycles of the LLVM node *i*	Static Analysis
$\#Mem\_bytes_i$	# of average memory bytes of of the LLVM node *i*	Static Analysis
$\#Mem\_burst_i$	# of average memory burst length of the LLVM node *i*	Static Analysis
$\#Trip_i$	# of loop trip count of the LLVM node *i*	Dynamic Analysis

mers can enable these two optimizations (SIMD and UL) via annotation in the OpenCL source code. For the entire kernel, programmer can specify the degree of SIMD (*#SIMD*). For the loop of the basic block, programmer can specify the unrolling factor $\#UL_i$. Since both SIMD and UL affect the number of memory and computation instructions of the basic block and each instruction has its own custom circuit, we extend the LLVM tool to handle those two attributes. In particular, we define a *combined scale factor* $f = \#UL_i \times \#SIMD$ for the basic block *i* to reflect their combined effect.

$\#Mem\_insts_i$. It represents the total number of global memory instructions in the basic block *i*, as shown in Eq. 1. For each global memory operation *j* (with load/store bytes: $ls\_bytes_i^j$), the calculation is based on the combined scale factor f. If $f = 1$, the number of global memory operations ($ls\_num_i^j$) is 1. Otherwise, $ls\_num_i^j$ is given in Eq. 2. Again, for this case, we consider whether the memory accesses are coalesced or not (i.e., whether their memory addresses are consecutive or not). If so, memory operations can be coalesced into wide memory transaction (up to *#Mem_trans_width* bits per transaction). Therefore, $ls\_num_i^j$ can be reduced significantly by a factor of *#Mem_trans_width*. Otherwise, the number is f, since each instruction after UL or SIMD issues f memory transactions.

$$\#Mem\_insts_i = \sum_j ls\_num_i^j \tag{1}$$

$$ls\_num_i^j = \begin{cases} 1 + \frac{ls\_bytes_i^j \times (f-1)}{\#Mem\_trans\_width}, & Coalesced \\ f, & Uncoalesced \end{cases} \tag{2}$$

$\#Mem\_bytes_i$. It denotes the average number of bytes of the global memory instructions in the basic block *i*, as given in Eq. 3, where $f \times \sum_j ls\_bytes_i^j$ represents the total size of data accessed by all the memory instructions after considering the combined effect from SIMD and UL.

$$\#Mem\_bytes_i = \frac{f \times \sum_j ls\_bytes_i^j}{\#Mem\_insts_i} \tag{3}$$

$\#Mem\_burst_i$. We collect the burst length $\#Mem\_burst_i^j$ of each global memory instruction *j* in the basic block *i* by analyzing its memory access pattern. It is larger than 1 when the address of global memory access increases with work item and *#Mem_trans_width* is larger than $ls\_bytes_i^j \times f$. That is, memory access order is sequential. A few memory transactions can be merged into a single wide

Table 2: Latency (cycles) of computation instruction

fp_sqrt	fp_mul	fp_add	local memory instruction	fp_div
28	5	7	7	14
int_mul	int_div	branch	int_add/bitwise	
3	32	1	1	

memory transaction [2], as shown in Eq. 4. So, in order to achieve better performance, the sequential memory access is recommended, not the random memory access. Then, $\#Mem\_burst_i$ is computed as the average burst length of the global memory operations, as shown in Eq. 5.

$$\#Mem\_burst_i^j = \begin{cases} \frac{\#Mem\_trans\_width}{ls\_bytes_i^j \times f}, & Coalesced \\ 1, & Uncoalesced \end{cases} \tag{4}$$

$$\#Mem\_burst_i = \frac{\sum_j \#Mem\_burst_i^j}{\#Mem\_insts_i} \tag{5}$$

$\#Cycle_i$. It denotes the total number of cycles of the critical path in the basic block *i*, including all the computation instructions, where the local memory load/store instructions are considered to be computation instructions since they only access the on-chip buffers, not the global memory. We employ dynamic programming approach to compute the critical path [21]. The cycles for the basic kind of instructions are summarized as shown in Table 2. We have obtained the unit costs from profiling the FPGA IP cores of the Altera OpenCL SDK.

3.3 Run-time Statistical Collector

In this subsection, we determine the run-time statistics from dynamic profiling, such as $\#Trip_i$.

We design another LLVM built-in pass (based on *llvm::LoopPass* class) to determine the structure of each node in the LLVM hierarchy. Then, we log down the trip count for each LLVM node ($\#Trip_i$) when running the same OpenCL kernel with the same input parameters on GPU, since the trip count can vary with the input data. For a loop, when loop unrolling $\#UL_i$ is applied, the number of trip counts $\#Trip_i$ is reduced by $\#UL_i$ times, since the dedicated pipeline is replicated by $\#UL_i$ times.

In summary, with static and dynamic analysis, we have constructed the LLVM hierarchy (as illustrated in Figure 5) and generated the corresponding static and dynamic statistics. Then, these statistics are fed into the analytical model which determines the execution time of the OpenCL kernel and the corresponding four potential metrics.

4. ANALYTICAL MODEL

In this section, we firstly present the overall estimation of the input OpenCL kernel, secondly the performance estimation of each leaf node (basic block) of the input OpenCL kernel, and thirdly the number of active work items residing on each basic block.

4.1 Overall Estimation

The estimated time of executing the input OpenCL kernel is given in Eq. 6, where C_S is the estimated number of cycles, and S denotes the root node (i.e., the entrance node) of the CFG of the OpenCL kernel.

$$T_S = \frac{C_S}{\#Freq} \tag{6}$$

According to the structure of the LLVM hierarchy, the performance estimation of C_S is done in a recursive manner.

Sequential type. If S consists of multiple child nodes, denoted as S_i (i is from 1 to N_S). Thus, according to the pipelined execution model of the FPGA, we define C_S as the largest number of cycles among child nodes, as shown in Eq. 7.

$$C_S = Max(C_{S_1}, C_{S_2}, ..., C_{S_{N_S}}) \tag{7}$$

If S is a basic block, we estimate its execution cycles according to the computation and memory instructions. Then, the recursion finishes. The detailed estimation of a basic block are provided in subsection 4.2.

If-else type. It is estimated to be the larger number of cycles between its IF child node and $ELSE$ child node of S, as shown in Eq. 8. Then, the recursion continues.

$$C_S = Max(C_{S^{IF}}, C_{S^{ELSE}}) \tag{8}$$

where S^{IF} and S^{ELSE} denotes the child nodes of S for IF and $ELSE$ branches, respectively. The *SWITCH* clause can be handled similarly.

For-loop type. The estimated number of cycles is shown in Eq. 9. S contains N_S child nodes, with each child node executing $\#Trips_S$ times.

$$C_S = \sum_{j=1}^{N_{S_i}} C_{S_j} + (\#Trips_S - 1) \times \max_{1 \le t \le N_S} (C_{S_t}) \tag{9}$$

The first part of Eq. 9 is the total number of cycles for all child nodes of S, and the second part is the elapsed cycles when executing the slowest child node for $\#Trips_S$ times, according to the pipelined execution model among the child nodes. From this estimation, it is important to optimize the slowest node within the *for* loop, especially when the number of iterations is high.

4.2 Estimation for a Basic Block

If k is a basic block, we estimate the cost of executing k to be the larger one of the computation and the memory cycles, considering that computation and memory instructions can overlap with each other. C_k represents the estimated number of cycles for executing basic block k, as shown in Eq. 10.

$$C_k = Max(Comp_k, Mem_k) \tag{10}$$

where $Comp_k$ and Mem_k represent the estimated number of cycles for executing the computation and memory instructions, respectively. In the following, we present their detailed estimations.

4.2.1 Estimating $Comp_k$

When estimating $Comp_k$, we need to consider whether there are sufficient active work items residing on the basic block k, $\#WI_k$ (the details of its estimation are presented later in subsection 4.2.3). Particularly, we check if the number of active work items in this basic block is larger than $\#CU \times \#Cycle_k$, where $\#CU$ (multiple CUs) means the kernel pipeline is replicated by $\#CU$ times and then the number of work items saturating kernel pipelines is also increased by $\#CU$ times. If so, we estimate $Comp_k$ to be $Comp_k^{min}$ (as shown in Eq. 11), where $\hat{k}$ indicates the LLVM node index for the basic block k. In this case, the pipeline of the basic block is fully utilized with sufficient work items. Then, data dependencies from one work item can be hidden due to massive thread parallelism within the OpenCL execution model and the corresponding performance actually depends on the throughput of the kernel pipelines. Otherwise, it is estimated to multiply $Comp_k^{min}$ by a delay factor p ($p = \frac{\#CU \times \#Cycle_k}{\#WI_k}$), because of insufficient work items existing on the basic block k.

$$Comp_k^{min} = \frac{\#WI \times \#Trip_{\hat{k}}}{\#CU} + \#Cycle_k - 1 \tag{11}$$

In summary, $Comp_k$ is estimated in Eq. 12.

$$Comp_k = Max(Comp_k^{min}, Comp_k^{min} \times \frac{\#CU \times \#Cycle_k}{\#WI_k}) \tag{12}$$

4.2.2 Estimating Mem_k

Mem_k is calculated to be the estimated number (Mem_trans_k) of global memory transactions divided by the estimated number (MTP_lsu_k) of memory transactions from the basic block k served per cycle, as shown in Eq. 13.

$$Mem_k = \frac{Mem\_trans_k}{MTP\_lsu_k} \tag{13}$$

We now present the details on the estimation of Mem_trans_k and MTP_lsu_k.

Evaluating Mem_trans_k. It is the estimated number of global memory transactions generated from the basic block k, where each work item will generate $\#Mem\_insts_k$ global memory operations, as shown in Eq. 14. However, the number of global memory transactions with sequential memory access pattern can be significantly reduced since they can be coalesced ($\#Mem\_burst_k > 1$), where $\hat{k}$ indicates the LLVM node index for the basic block k.

$$Mem\_trans_k = \frac{\#Mem\_insts_k \times \#WI \times \#Trip_{\hat{k}}}{\#Mem\_burst_k} \tag{14}$$

Multiple CUs ($\#CU > 1$) do not affect the number of global memory transactions for each basic block, since all CUs share the global memory bandwidth, as shown in Figure 3.

Evaluating MTP_lsu_k. Our estimation considers the number of work items on the memory subsystem ($\#WI^{mem}$). The estimation of $\#WI^{mem}$ is presented later in subsection 4.2.3. If there are sufficient work items existing on the memory subsystem. We calculate the peak number of simultaneous memory transactions to global memory, MTP_lsu_peak, to be the number $\#Banks$ of global memory banks, as shown in Eq. 15. That is, one global memory transaction can be served per cycle for each memory bank.

$$MTP\_lsu\_peak = \#Banks \tag{15}$$

Otherwise, if there are insufficient work items existing on the memory subsystem, $\#WI^{mem}$ is smaller than $\#DRAM\_lat \times MTP\_lsu\_peak$, the global memory bandwidth can be under-utilized. The actual peak number of memory transactions served per cycle MTP_lsu is estimated to be the smaller value of the two numbers (MTP_lsu_peak and $\frac{\#WI^{mem}}{\#DRAM\_lat}$), as shown in Eq. 16.

$$MTP\_lsu = \min\left(MTP\_lsu\_peak, \frac{\#WI^{mem}}{\#DRAM\_lat}\right) \tag{16}$$

Third, since all the basic blocks of the input OpenCL kernel share the global memory bandwidth, we estimate the number of global memory transactions from the basic block k served per cycle MTP_lsu_k, as shown in Eq. 17, where $\frac{Mem\_cycles\_total_k}{Mem\_cycles\_total}$ means the memory bandwidth occupation ratio of the basic block k to the entire OpenCL kernel, and the corresponding estimations of $Mem\_cycles\_total_k$ and Mem_cycles_total are presented later in subsection 4.2.3.

$$MTP\_lsu_k = MTP\_lsu \times \frac{Mem\_cycles\_total_k}{Mem\_cycles\_total} \tag{17}$$

4.2.3 Estimating $\#WI_k$ and $\#WI^{mem}$

In this part, we estimate the number of active work items $\#WI_k$ (or $\#WI^{mem}$) on the computation part of basic block k (or on the memory subsystem of the OpenCL kernel). The total number of work items for all the basic blocks is $\#WI$ at any time of the execution.

To simplify the estimation, we assume that the work items are distributed to the computation/memory parts of all basic blocks based on their total numbers of computation/memory cycles. We can obtain the total number ($Comp\_cycles\_total_k$) of computation cycles for the basic block k according to the LLVM hierarchy. In particular, we perform the calculation from the bottom up along the path from the basic block k to the root. Initially, $Comp\_cycles\_total_k$ is set to $\#Cycle_k$. Along the path, if we meet a loop-type node s, we multiply $Comp\_cycles\_total_k$ by a factor of $\#Trip_s$. The total number ($Mem\_cycles\_total_k$) of memory cycles for each basic block k can be calculated similar to $Comp\_cycles\_total_k$.

With the total cycles for each basic block, we sum them up and obtain $Comp\_cycles\_total$ and Mem_cycles_total to be the total number of cycles for computation and memory parts of the entire kernel, respectively. We calculate $Cycles\_total = Comp\_cycles\_total + Mem\_cycles\_total$ for the OpenCL kernel.

Therefore, we estimate $\#WI_k$ and $\#WI^{mem}$ in Eqs. 18– 19.

$$\#WI_k = \frac{Comp\_cycles\_total_k}{Cycles\_total} \times \#WI \tag{18}$$

$$\#WI^{mem} = \frac{Mem\_cycles\_total}{Cycles\_total} \times \#WI \tag{19}$$

5. PERFORMANCE ADVISOR

Recall that one of the key purposes of the proposed framework is to assist programmers to identify the performance bottleneck of their OpenCL programs, as in the evaluation phase of iterative and incremental software development. Therefore, we propose to quantify the performance potentials for further optimizations in programmer understandable metrics. With these metrics, programmers are able to understand the root cause of the performance bottleneck, and to choose to implement which type of optimizations next. If all the metrics are very small, the OpenCL program tends to work fairly well on FPGAs.

Specifically, we propose the following four metrics to quantify the performance potentials according to the architectural features of FPGA, including B_{comp}, B_{mem}, $B_{balance}$ and B_{itpp}. They represent the extent that the bottleneck comes from computation, memory, load balance among different basic blocks and inter-thread parallelism respectively. In practice, programmers can choose the one with the highest potential for the optimization of next step. In the following, we elaborate more details for each of the potential metrics, as well as the FPGA-centric optimizations to reduce those metrics.

B_{comp} represents the performance potential from more resource utilization on FPGA. We estimate B_{comp} according to the resource utilization in three major resource types of FPGA including logic blocks, memory blocks and DSP blocks. B_{comp} is calculated as the bottleneck resource type in Eq. 20, compared with the initial resource availability of each resource type. If B_{comp} is high, programmers may consider increasing resource utilization by using the optimizations CU, UL and SIMD.

$$B_{comp} = min\left(\frac{1-\#U_{logic}}{I_L}, \frac{1-\#U_{mem}}{I_M}, \frac{1-\#U_{dsp}}{I_D}\right) \tag{20}$$

B_{mem} represents the performance potential from more global memory bandwidth utilization. Since the FPGA does not have a cache hierarchy, each load/store instruction directly goes to the global memory. We estimate B_{mem} to be the available bandwidth in comparison with the peak global memory bandwidth, as shown in Eq. 21. $\sum_{i=1}^{\#N_B}(\#Mem\_bytes_i \cdot \#Mem\_insts_i \cdot \#Mem\_burst_i)$ represents the real memory bandwidth, and $(\sum_{i=1}^{\#N_B}\#Mem\_insts_i) \times \#Mem\_trans\_width$ calculates the total memory bandwidth by the kernel if all transactions uses the peak transaction width. If B_{mem} is high, programmer can apply MC, UL and SM to improve the memory bandwidth utilization.

$$B_{mem} = 1 - \frac{\sum_{i=1}^{\#N_B}\#Mem\_bytes_i \cdot \#Mem\_insts_i \cdot \#Mem\_burst_i}{\#Mem\_trans\_width \times (\sum_{i=1}^{\#N_B}\#Mem\_insts_i)} \tag{21}$$

$B_{balance}$ indicates the performance potential from more load balancing among the basic blocks. Due to the pipelined execution model of FPGA, the execution time of an OpenCL kernel is determined by the slowest basic block. To offer a simple and understandable hint, we compute $B_{balance}$ as the performance difference between the slowest basic block ($\hat{s}$) and second-slowest basic block ($\hat{t}$), as shown in Eq. 22, where $C_{\hat{s}}$ and $C_{\hat{t}}$ are the execution cycles of the basic blocks

Table 3: Application and datasets used in our experiments

Application	Dataset Size
Vector Add, (VecAdd)	32M integers for each vector size
Matrix Multiplication (MM)	2048*2048 matrices
K-means, K=128, 8 features (KM)	32M points
Word Count (WC)	1500MB text file
Similarity Scope (SS)	8000 files each with 8000 features

$\hat{s}$ and $\hat{t}$, estimated at subsection 4.2. If $B_{balance}$ is high, the programmer can unroll the most critical basic block with the optimization method UL (if the resource budget allows).

$$B_{balance} = \frac{C_{\hat{s}} - C_{\hat{t}}}{C_{\hat{s}}} \tag{22}$$

B_{itpp} represents the performance potential from more inter-thread pipeline parallelism. If there are insufficient alive work items to feed the OpenCL kernel pipeline, the performance degrades. We compute B_{itpp}, as shown in Eq. 23.

$$B_{itpp} = \max_{1 \leq t \leq \# N_B} (max(\frac{C_t - Comp_t^{min}}{C_t}, \frac{C_t - Mem_t^{min}}{C_t})) \tag{23}$$

We have defined $Comp_t^{min}$ in Eq. 11. For the basic block i, Mem_i^{min} is defined to be the estimated number of memory cycles when there are sufficient work items residing on the OpenCL kernel, as shown in Eq. 24.

$$Mem_i^{min} = \frac{Mem\_trans_i}{MTP\_lsu\_peak} \times \frac{Mem\_cycles\_total}{Mem\_cycles\_total_k} \tag{24}$$

If B_{itpp} is high, programmers need to increase the number of work items for better pipeline parallelism of the kernel execution.

6. EXPERIMENTAL EVALUATION

6.1 Experimental Setup

Our experiments were conducted on Terasic's DE5-Net board with Altera OpenCL SDK 14.0, which includes 2-bank 4GB DDR3 device memory (operating in roughly hundreds of MHz), and an Altera Stratix V GX FPGA (5SGXEA7N2F45C2). The FPGA includes 622K logic elements, 2560 M20K memory blocks (50Mbit) and 256 DSP blocks. The FPGA board is connected to the host via an X8 PCI-e 2.0 interface.

Applications. We have chosen some OpenCL programs with different application areas as use cases. We start with two relatively simple case studies: vector add (VecAdd) from Altera OpenCL SDK and a simple implementation of matrix multiplication (MM). We further choose more complicated examples: K-means (KM) from Rodinia [6] and MapReduce [9, 10]. The details on the applications and their data sets are summarized in Table 3.

Methodology. Our evaluations cover two major aspects of the proposed framework. Firstly, we evaluate the accuracy of our analytical model for predicting the performance of an OpenCL kernel with various FPGA-centric optimizations. Thus, we compare our performance analysis model in capturing the performance gain brought by individual optimizations and their combination. Secondly, we evaluate the effectiveness of the performance potential metrics in helping software development. As in iterative and incremental software development, we mimic the process by considering different FPGA-centric optimizations step by step. We demonstrate that the proposed framework can give hints to programmers which FPGA-centric optimization to choose and determine the suitable value for its parameter.

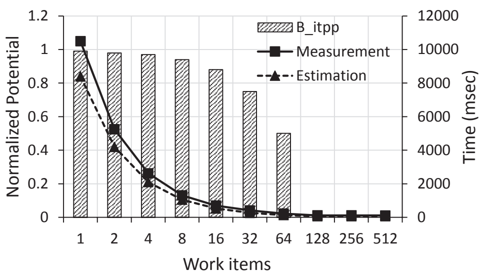


Figure 6: Performance and B_{itpp} with different numbers of work items

6.2 Micro Benchmark: VecAdd

We use a microbenchmark (VecAdd) to evaluate the impact of active work items (B_{itpp}) to the kernel performance, as shown in Figure 6. Our performance model is able to accurately predict the performance in real executions. When the number of active work items is small, the pipeline is severely under-utilized due to the shortage of work items and B_{itpp} is large. Thus, the programmer should provide sufficient work items (thread pipeline parallelism) to fully utilize the kernel pipeline. When the number of work items is larger than 128, the performance keeps stable. Our performance model can capture the performance trend with different numbers of active work items. Also, in our estimation, all other three potentials are low.

In the following case studies, we have already set a large number of work items so that B_{itpp} is minimized. Therefore, we focus on other three performance potential metrics afterwards.

6.3 Case Study with Matrix Multiplication

We apply our performance model to the matrix multiplication (MM), which is classical application in the area of HPC. MM's baseline pseudo code is shown in Algorithm 2. The baseline represents a very preliminary and simple implementation.

Algorithm 2: MM ALGORITHM: C=A×B

```
1 for ($i \leftarrow 0$ to $A\_height - 1$) do
2     for ($j \leftarrow 0$ to $B\_width - 1$) do
3         $float\ sum = 0.0$;
4         for ($k \leftarrow 0$ to $A\_width - 1$) do
5             $sum += A[i][k] \times B[k][j]$;
6         end
7         $C[i][j] = sum$;
8     end
9 end
```

The data parallelism in MM is straightforward: all the elements of matrix C are calculated in parallel, represented as two nested loops in Lines 1–2. In this naive implementation, all matrices are stored in the row order. Each element $C[i][j]$ at Line 7 requires A_width floating point multiply-add operations and $A\_width \times 2$ global memory loads. Such repetitive memory accesses cause significant performance degradation.

6.3.1 *Optimizations for MM*

We study the following optimizations to MM one by one.

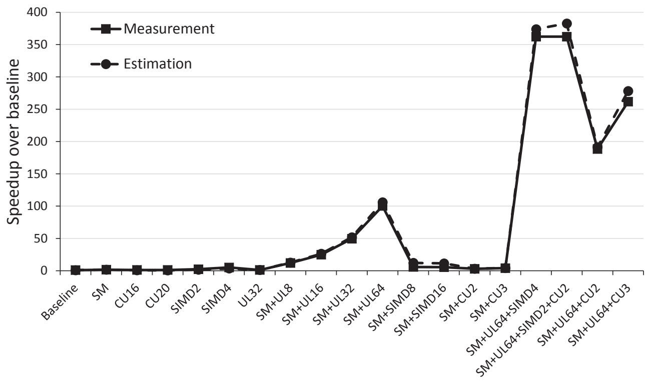


Figure 7: MM speedup with 18 optimization combinations

Local Memory (SM): We use the optimization SM to store two small blocks of matrices A and B to implement the tiling technique [31].

Loop Unrolling (UL):The innermost for-loop at Line 4 can be first unrolled to increase the throughput of this for-loop. It also reduces the number of global memory transactions. In particular, $A[i][k]$ at Line 5 can be coalesced, while $B[k][j]$ cannot, since the *for* loop at Line 4 iterates with the value of k. According to $\#Mem\_trans\_width$ (512-bit width) of DDR controller on FPGA, at most 16 32-bit memory operations of $A[i][k]$ can be coalesced. That reduces the number of total memory accesses, which may further improves the overall performance. We denote the optimization of the loop unrolling with a factor of g to be ULg.

Kernel Vectorization (SIMD): The floating-point multiplication addition operations at Lines 5 and 7 can be vectorized. We denote the optimization of combining v scalar arithmetic operations into a vector operation to be SIMDv.

Kernel pipeline replication (CU): Multiple compute units (CUs) are generated to execute MM algorithm together. We denote the optimization of c CUs to be CUc.

6.3.2 Performance Model Evaluation for MM

Figure 7 shows the speedup over the baseline of the actual execution and our prediction. Some part of the results have been reported in Figure 1 in Introduction. The x-axis show 18 optimizations (including individual optimizations and their combinations). The result shows that our prediction can capture the performance gain of different optimizations.

Another thing should be mentioned is that the optimization combination ($SM+UL64+SIMD4$) is 362 times faster than the baseline implementation. Thus, it is necessary to provide the FPGA programmer with the performance potentials to efficiently utilize the FPGA computing resource.

6.3.3 Optimization Steps for MM

We now demonstrate how the proposed framework can guide the code tuning, starting from the baseline of MM, in three steps, as shown in Figure 8. Overall, during the code tuning process of MM, our framework clearly pin-points the performance bottleneck for each step, and provides insights for the corresponding optimization.

Step 1: baseline $\rightarrow$ SM. For the baseline kernel, B_{mem} is very large, indicating that we should explore the optimization methods to reduce memory bottleneck. Therefore, the optimization method *SM* is applied, B_{mem} is reduced significantly since the pattern of global memory access becomes suitable for dynamic memory coalescing.

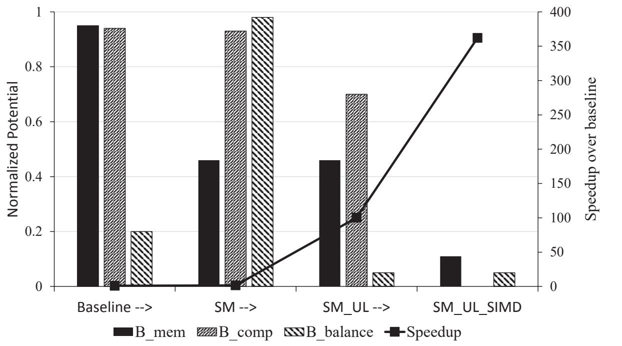


Figure 8: Potential metrics and optimization steps for MM

Step 2: SM $\rightarrow$ SM_UL. For the SM kernel, both B_{comp} and $B_{balance}$ are high, indicating that we should explore the optimization methods to reduce unbalance and computation bottlenecks. Therefore, the optimization method *UL* is applied to the SM kernel since it can reduce both $B_{balance}$ and B_{comp}. As a consequence, the kernel performance is significantly improved (63X speedup over the SM kernel).

Step 3: SM_UL $\rightarrow$ SM_UL_SIMD. For SM_UL kernel, B_{comp} is still high, indicating that we should explore the optimization methods to reduce computation bottleneck. At the same time, B_{comp} is relatively high. Therefore, the optimization method *SIMD* is applied, and then both B_{comp} and B_{mem} are reduced. We may stop the optimization progress here, since B_{comp} has become zero (e.g. 100% utilization of DSP blocks in the FPGA). The SM_UL_SIMD kernel achieves the optimum performance for MM on our FPGA platform.

6.4 Case Study with K-means

KM is a clustering algorithm used extensively in data-mining. The current OpenCL implementation (baseline) is from Rodina [6] and it is originally designed for GPU. So, its performance is far from optimum on FPGAs.

6.4.1 Optimizations for KM

We examine the impact of the following optimizations: *SM*, *UL*, *MC* and *CU*. Since optimizations are quite similar to those in MM, we briefly describe some details about them. *SM* is to load the data (cluster center) into the local memory of FPGA, since they are reused for each data object. When applying *UL*, the optimization is to unroll the destination computation process, since there is significant load imbalance in the kernel. There are two candidate loops to apply this optimization method, including inner loop and outer loop in the main body of the source code (we omit the source code, and refer readers to the Rodinia benchmark [6]). We denote *ULx_ULy* for the case where the inner loop's unroll factor is x and the outer loop's unroll factor is y. *MC* is to refine the global memory access pattern so that the address of memory instruction increases with the number of work items.

6.4.2 Performance Model Evaluation for KM

Figure 9 shows the speedup over the baseline kernel of

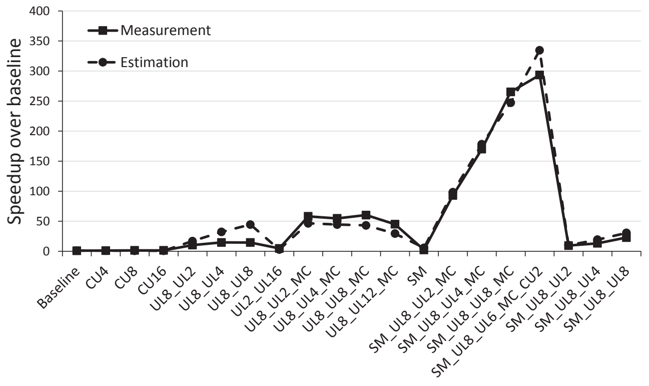


Figure 9: KM speedup with 19 optimization combinations

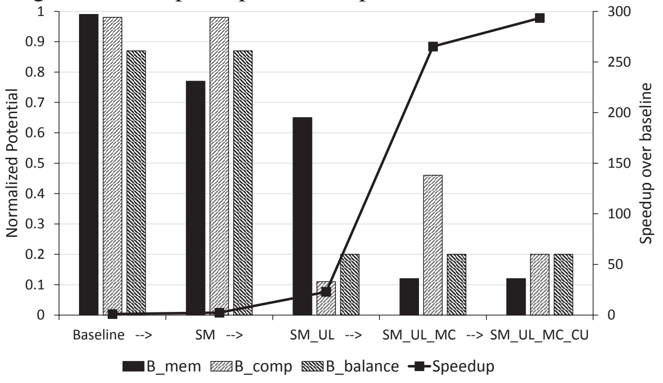


Figure 10: Potential metrics and optimization steps for KM

the actual execution. The result shows that our prediction can capture the trend of different optimization combinations. When the optimization method *MC* is applied, the performance improves significantly since the global memory access pattern has good locality on FPGAs. It indicates that the performance is very sensitive to the memory access pattern. Interestingly, with *MC*, the computation turns out to be the bottleneck, and then *UL* and *CU* can boost the performance. Without *MC*, the bottleneck of the kernel implementation is the under-utilized global memory bandwidth (indicated by large B_{mem} in our estimation).

About the optimization *UL*, the inner loop is more critical than the outer loop. Guided by the performance model, we should first put the FPGA resource to the inner loop. When it is fully unrolled, the outer loop can be further unrolled if there is still available resource in FPGA. This code tuning strategy has very sensitive performance gain. Our tool can assist this decision making. For example, the kernel with *UL2_UL16* (137s) is much slower than *UL8_UL2* (64.5s).

6.4.3 Optimization Steps for KM

We now demonstrate how the proposed framework can guide the code tuning of KM in four steps, as shown in Figure 10.

Step 1: baseline → SM. For the baseline kernel, B_{mem} is very large, indicating that the memory performance is the main bottleneck. Therefore, the optimization method *SM* is applied, B_{mem} is reduced since the pattern of global memory access becomes suitable for dynamic memory coalescing.

Step 2: SM → SM_UL. For the SM kernel, both $B_{balance}$ and B_{comp} are relatively high, indicating that we should explore the optimization methods to reduce unbalance and computation bottlenecks. Therefore, the optimization method *UL* is applied since it can reduce both $B_{balance}$ and B_{comp}.

Step 3: SM_UL → SM_UL_MC. For the SM_UL kernel, B_{mem} is still high, indicating that we should explore the optimization methods to reduce memory bottleneck. Therefore, when the optimization method *MC* is applied, B_{mem} is significantly reduced since the optimized memory access pattern is suitable for the optimization method *UL*. And the performance is improved significantly (11.6X speedup over SM_UL kernel). Interestingly, B_{comp} becomes larger since the *MC* can reduce the number of global memory access instructions, compared with the SM_UL kernel.

Step 4: SM_UL_MC → SM_UL_MC_CU. Since there are FPGA resources available for the further optimization, the optimization *CU* is applied, then B_{comp} is reduced. We may stop the optimization progress here, since three metrics are relatively small.

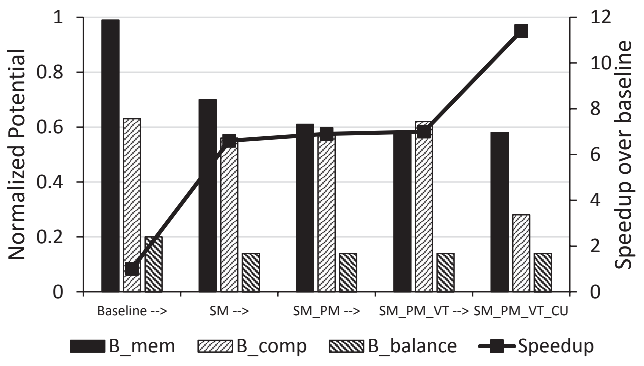


Figure 11: Potential metrics and optimization steps for WC

6.5 Case Study with MapReduce

MapReduce has been widely studied on parallel architectures such as GPUs [16, 9, 10]. According to previous papers [9, 10], MapReduce applications mainly have two kinds: reduction-intensive and map-computation-intensive. We choose Word Count (WC) and Similarity Scope (SS) as the representatives for reduction-intensive and map-computation-intensive applications, respectively. WC is to count the number of occurrences of words in a set of text files, and SS is to calculate the similarity score among documents, which is widely used as basic routines in web search.

6.5.1 Optimizations for MapReduce

In MapReduce applications, we consider *SM*, *UL* and *CU*, which are similar to the above applications. Additionally, we discuss more details on the following two optimizations.

Private Memory (PM). The private memory is implemented using completely-parallel registers (logics), which are plentiful resources in FPGAs. Then, the private memory is useful for storing temporary variables or small arrays in the OpenCL kernel [2]. The kernel can access private memory completely in parallel, and no arbitration is required for access permission. Since MapReduce applications contain a lot of global memory accesses, we should use private memory, instead of global memory, whenever possible.

Vector Type (VT). The programmer should employ built-in vector type, e.g.*uint4*, to combine several small-sized global memory accesses, e.g. *uint*. Then, the number of

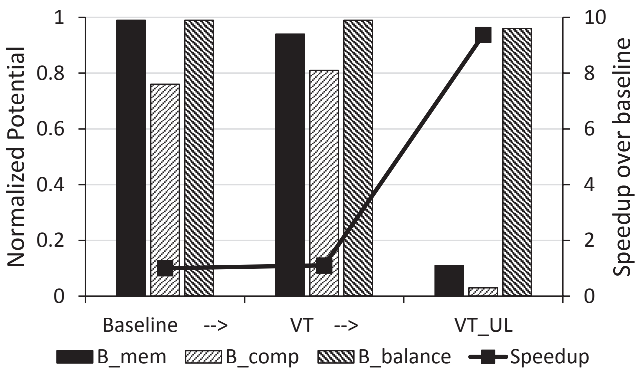


Figure 12: Potential metrics and optimization steps for SS

global memory transactions is reduced.

6.5.2 *Optimization Steps for MapReduce*

Figures 11 and 12 show how the proposed framework can effectively guide the code tunings for both WC and SS. The optimizations are applied to the user-defined *map* and *reduce* functions in those two applications. The optimum implementation is over 11 (or 9) times faster than the baseline of WC (or SS).

WC has four tuning steps by applying *SM*, *PM*, *VT* and *CU* accordingly. SS has two tuning steps by applying *VT* and *UL* accordingly. Due to the space limitation, we omit the discussion on code tuning, and point out the different findings compared with the above experiments on MM and KM. After tuning, WC and SS still have some potential metric with a high value. For WC, B_{mem} is always high for each step. This is due to the inherent random memory accesses due to the random insertions into the hash table for the intermediate data. For SS, $B_{balance}$ is high for the VT_UL kernel. The proposed framework cannot suggest any further optimization due to the resource constraint of FPGA. That also explains the relatively low speedup on MapReduce than on MM and KM. For further performance improvement, programmers need to develop more efficient algorithms or employ more powerful FPGAs (rather than pure FPGA-centric optimizations studied in this experiment).

Table 4: Speedup of our framework and "-O3 approach" in Altera OpenCL SDK over baseline

	MM	KM	WC	SS
-O3 approach	0.7	4.4	0.93	1
Our framework	362.4	293.4	11.4	9.4

6.6 Comparison with Altera SDK

We compare the performance of the OpenCL program 1) with the optimization guided by our framework, and 2) with the Altera SDK *resource driven optimizations* ("-O3 approach", "aoc -O3" command, with default *maximum_logic_utilization=85%*) [2]. In particular, we apply these two optimization frameworks to the four baseline implementations (MM, KM, WC and SS). The experiment result shows that the implementations optimized by our framework over the baselines can achieve significantly higher speedups than the Altera SDK resource driven optimizations, as shown in Table 4. Compared with the baseline, the "-O3 approach" even slows down the performance for some cases such as MM and WC. We further analyze "-O3 approach" on MM. The "-O3 approach" suggests (#SIMD=2 and #CU=4), which requires twice on the number of RAM blocks and four times on the number of DSP blocks of the baseline. However, its performance is even slower than the baseline approach. The major reason is that setting #CU=4 does not reduce the number of global memory transactions (major bottleneck), while the frequency is significantly dropped due to more resource requirement. In fact, our performance model can capture this effect.

7. RELATED WORK

OpenCL has become a popular programming framework for heterogeneous computing, including high performance computing [30] and databases [17, 18]. Most of those studies are performed on CPU/GPU. Evaluations with OpenCL implementations for computationally intensive applications have been conducted on CPUs, GPUs and FPGAs [7, 13]. Generally, FPGAs achieve much better energy efficiency than CPUs/GPUs. With the introduction of OpenCL SDK on FPGAs, the programmability of FPGAs has been eased, and more applications can be deployed and ran on FPGAs.

There have been a number of performance models for FPGAs (e.g., [12, 14, 15, 24, 32]). Curreri et al. [12]'s model is mainly for Impulse C. Silva et al. [14] developed a Roofline model for high-level synthesis tools. Park et al. [25] developed a cost model targeting at loop transformations. Papadimitriou et al. [24] developed the cost model to assess partial reconfiguration of FPGAs. Quite different from the previous studies with a focus on individual applications or on HDL, this study focuses on the emerging OpenCL SDK. We develop performance analysis and diagnosis tool for OpenCL programs on FPGAs.

Performance model and analysis is a hot research topic on CPUs/GPUs. Due to the architectural differences between FPGAs and CPUs/GPUs, previous studies on CPUs/GPUs (e.g., [3, 20, 27, 29, 33]) cannot be directly to FPGAs. Since GPUs are emerging in recent years, we briefly review a number of studies on GPUs. Ryoo et al. [27] summarized a number of optimization principles for CUDA programs. Baghsorkhi et al. [3] presented a cycle-accurate analytical model on GPU to predict the performance of GPU applications. Zhang et al. [33] developed a throughput model by modeling three major components, including instruction pipeline, shared memory access, and global memory access. Kim et al. [20] proposed the warp-aware performance analysis model to estimate the performance of GPGPU application. Further, Sim et al. [29] developed a novel performance analysis model to exactly identify the main cause of performance bottlenecks of GPGPU applications. Sharing the same spirit as the framework and tools on GPUs (e.g., [29, 33]), our study develops a performance model and analysis framework for OpenCL programs on FPGAs. By capturing the performance behavior on OpenCL-based FPGAs, the proposed performance model is significantly different from those in the previous studies for CPUs/GPUs.

8. CONCLUSIONS AND FUTURE WORK

This paper proposes a performance model that captures the key architectural features of the new FPGA abstraction under OpenCL, predicts the performance with different optimization combination and identifies the bottlenecks of OpenCL kernel code. We demonstrate the efficiency of our model with several user cases, by considering dozens of optimization combinations. The results show that our proposed model is highly accurate in predicting the performance speedup of individual optimization and their combinations, and that careful tuning can achieve up to two orders of magnitude speedup over the baseline implementation. Also, our tools can offer programmer understandable metrics, and guide the code tuning on resolving performance bottlenecks step by step. As for future work, we plan to automate the performance analysis framework for FPGAs using OpenCL.

9. ACKNOWLEDGEMENT

We thank Altera University Program which has denoted Terasic's DE5-Net FPGA board for our research. Also, we thank John Freeman at Altera for providing his constructive feedback about Altera OpenCL SDK. This work is in part supported by MoE AcRF Tier 2 grants (MOE2012-T2-1-126 and MOE2012-T2-2-067) in Singapore.

10. REFERENCES

[1] Altera. Implementing fpga design with the opencl standard. *Altera Whitepaper*, 2011.

[2] Altera. Altera sdk for opencl optimization guide. 2014.

[3] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2009.

[4] K. K. Bor-Yiing Su. clspmv: A cross-platform opencl spmv framework on gpus. In *ACM international conference on Supercomputing (ICS)*, Jun. 2012.

[5] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, Mar. 2011.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, Oct. 2009.

[7] D. Chen and D. Singh. Using opencl to evaluate the efficiency of cpus, gpus and fpgas for information filtering. In *International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2012.

[8] D. Chen and D. Singh. Fractal video compression in opencl: An evaluation of cpus, gpus, and fpgas as acceleration platforms. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2013.

[9] L. Chen and G. Agrawal. Optimizing mapreduce for gpus with effective shared memory usage. In *international symposium on High-Performance Parallel and Distributed Computing (HPDC)*, Jun. 2012.

[10] L. Chen, X. Huo, and G. Agrawal. Accelerating mapreduce on a coupled cpu-gpu architecture. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2012.

[11] C.Lattner and V.Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization (CGO)*, Mar. 2004.

[12] J. Curreri, S. Koehler, A. D. George, B. Holland, and R. Garcia. Performance analysis framework for high-level language applications in reconfigurable computing. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)*, Jan. 2010.

[13] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh. From opencl to high-performance hardware on fpgas. In *International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2012.

[14] B. da Silva, A. Braeken, E. H. D'Hollander, and A. Touhafi. Performance modeling for fpgas: Extending the roofline model with high-level synthesis tools. *International Journal of Reconfigurable Computing (IJRC)*, 2013.

[15] L. Deng, K. Sobti, Y. Zhang, and C. Chakrabarti. Accurate area, time and power models for fpga-based implementations. *Signal Processing Systems*, 2011.

[16] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[17] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proc. VLDB Endow.*, 6(10):889–900, Aug. 2013.

[18] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled cpu-gpu architectures. *Proc. VLDB Endow.*, 8(4), Dec. 2014.

[19] M. C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello. Achieving high performance with fpga-based computing. In *Computer (Long Beach Calif)*, March, 2007.

[20] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2009.

[21] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, May 1996.

[22] C. Larman and V. Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, June 2003.

[23] R. Mueller and J. Teubner. Fpga: What's in it for a database? In *International Conference on Management of data (SIGMOD)*, Jun. 2009.

[24] K. Papadimitriou, A. Dollas, and S. Hauck. Performance of partial reconfiguration in fpga systems: A survey and a cost model. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)*, Dec. 2011.

[25] J. Park, P. Diniz, and K. Shesha Shayee. Performance and area modeling of complete fpga designs in the presence of loop transformations. *IEEE Transactions on Computers (TC)*, Nov. 2004.

[26] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2014.

[27] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Principles and Practice of Parallel Programming(PPoPP)*, Mar. 2008.

[28] S. Sarkar, T. Majumder, A. Kalyanaraman, and P. Pande. Hardware accelerators for biocomputing: A survey. In *International Symposium on Circuits and Systems (ISCAS)*, May 2010.

[29] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. In *Principles and Practice of Parallel Programming(PPoPP)*, Mar. 2012.

[30] B.-Y. Su and K. Keutzer. clspmv: A cross-platform opencl spmv framework on gpus. In *ACM international conference on Supercomputing (ICS)*, Jun. 2012.

[31] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2008.

[32] Z. Wang, B. He, and W. Zhang. A study of data partitioning on opencl-based fpgas. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–8, Sept 2015.

[33] Y. Zhang and J.D.Owens. A quantitative performance analysis model for gpu architectures. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, Feb. 2011.