CGPredict: Embedded GPU Performance Estimation from Single-Threaded Applications

SIQI WANG, National University of Singapore
GUANWEN ZHONG, National University of Singapore
TULIKA MITRA, National University of Singapore

Heterogeneous multiprocessor system-on-chip architectures are endowed with accelerators such as embedded GPUs and FPGAs capable of general-purpose computation. The application developers for such platforms need to carefully choose the accelerator with the maximum performance benefit. For a given application, usually, the reference code is specified in a high-level single-threaded programming language such as C. The performance of an application kernel on an accelerator is a complex interplay among the exposed parallelism, the compiler, and the accelerator architecture. Thus, determining the performance of a kernel requires its redevelopment into each accelerator-specific language, causing substantial wastage of time and effort. To aid the developer in this early design decision, we present an analytical framework *CGPredict* to predict the performance of a computational kernel on an embedded GPU architecture from un-optimized, single-threaded C code. The analytical approach provides insights on application characteristics which suggest further application-specific optimizations. The estimation error is as low as 2.66% (average 9%) compared to the performance of the same kernel written in native CUDA code running on NVIDIA Kepler embedded GPU. This low performance estimation error enables CGPredict to provide an early design recommendation of the accelerator starting from C code.

CCS Concepts: • **Computer systems organization** → **Parallel architectures**; **Heterogeneous (hybrid) systems**; **Embedded systems**; • **Computing methodologies** → *Model development and analysis*;

Additional Key Words and Phrases: Heterogenous platform, GPGPU, performance modeling, mobile platform, analytical model, cross-platform prediction

ACM Reference format:
Siqi Wang, Guanwen Zhong, and Tulika Mitra. 2017. CGPredict: Embedded GPU Performance Estimation from Single-Threaded Applications. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (October 2017), 22 pages.
DOI: 0000001.0000001

1 INTRODUCTION

The emergence of the heterogeneous system-on-chip platforms (e.g., Xilinx Zynq UltraScale+ MPSoC [24], Nvidia Jetson TK1 [20]) offers application developers diverse choice of accelerators including Graphics Processing Unit (GPU), Field-Programmable Gate Array (FPGA), Digital Signal

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue.
Authors' addresses: S. Wang, G. Zhong, T. Mitra, School of Computing, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117417. Authors' Email addresses: {wangsq, guanwen, tulika}@comp.nus.edu.sg.
 1539-9087/2017/10-ART39 $15.00
DOI: 0000001.0000001

Processor (DSP), etc. on the same chip. The developers now have the opportunity and the responsibility to take advantage of the unique characteristics of accelerators to improve the application performance. The appropriate choice of an accelerator that best matches an application kernel, however, is a challenging endeavor. The performance of an application on an accelerator is a complex interplay among the exposed parallelism, the compiler, and the accelerator architecture. The programmer needs to implement the kernel in different accelerator-specific languages (CUDA/OpenCL for GPU, RTL for FPGA) to measure the performance of each accelerator choice. Recent advances have somewhat alleviated this re-development effort. For example, High-Level Synthesis tools (e.g., Vivado HLS [23], LegUp [4]) can automatically generate RTL from C code for FPGAs, while [3] can perform C to CUDA transformation for GPU. There are also emerging frameworks, for example OpenCL [8] cross-platform parallel programming for heterogeneous systems, where the same program can run across diverse accelerators such as multi-core CPU, GPU, DSP, and FPGAs. Unfortunately, the generality of such approaches is also their shortcoming as accelerator-specific optimizations are imperative to unleash the true performance potential of a kernel on an accelerator.

Our goal is to guide the application developer in the early design choice of an accelerator without the tedious redevelopment effort and optimizations. Usually, the reference code for a kernel is specified in a high-level single-threaded programming language such as C. Starting with this sequential C code of a kernel, we aim to predict its relative performance on multiple accelerators such that the developer can make an informed choice. They can then concentrate their efforts on this selected accelerator with platform-specific languages and optimizations. The automated filtering of the unsuited accelerators saves tremendous effort that would have been otherwise completely wasted.

As one of the first steps towards achieving this goal of automated accelerator selection, we present *CGPredict* (C to GPU Prediction) — an analytical framework to accurately estimate the performance of a computational kernel on an embedded GPU architecture from unoptimized, single-threaded C code. The GPU is a highly multi-threaded architecture that thrives on concurrent execution of thousands of threads, which makes performance prediction from single-threaded code quite challenging. Moreover, modern GPUs feature complex memory hierarchy including caches that introduces considerable unpredictability in performance, making analytical memory performance modeling rather difficult. CGPredict builds the performance model from a dynamic execution trace of the sequential kernel. The trace is manipulated to expose the available thread-level parallelism that can be potentially exploited by the GPU. At the same time, the memory access trace is analyzed against a performance model of the memory hierarchy that captures the interaction between the cache, the DRAM memory, and the inherent memory latency hiding capability of the GPU through zero-cost context switching of the threads when necessary.

CGPredict can estimate the performance from C code with 9% estimation error compared to the performance of the corresponding native CUDA code on embedded NVIDIA Kepler GPU averaged across a number of kernels. As CGPredict is based on analytical modeling, it can provide insights regarding the characteristics of the kernel and the GPU that influence performance, including coalescing of memory accesses or shared memory usage. These insights offer opportunities for the programmers to understand the intrinsic strengths and weakness of the architecture in the context of a particular kernel that can facilitate further code optimizations. Also CGPredict in conjunction with an existing FPGA performance predictor from C code [26], achieves our objective of making the perfect choice of the accelerator (GPU or FPGA) given a kernel.

Performance estimation of general-purpose applications on GPU is a well-researched topic [1, 2, 7, 12, 22]. But CGPredict differs from the state-of-the-art in two important aspects. First, the earlier works primarily focused on performance estimation from CUDA [2, 7] or OpenCL

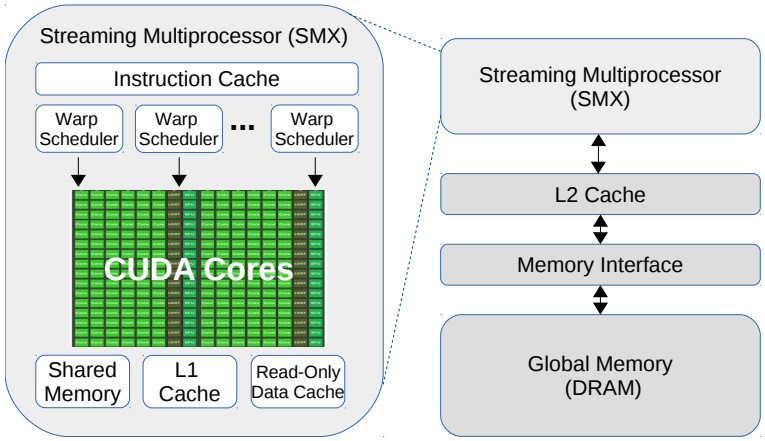

Fig. 1. Jetson TK1 Kepler GPU architecture

[22] where the thread-level parallelism has already been exposed. In contrast, CGPredict provides estimation from sequential single-threaded application.

Second, almost all existing techniques do not consider caches in the memory hierarchy and only model the software-controlled shared memory. As the content of the shared memory is under programmer control, the memory access latency is predictable. In contrast, state-of-art GPUs are usually endowed with multiple level of caches, including configurable L1 cache and L2 cache, which introduce unpredictable access latencies because the presence of a data element in a particular cache cannot be guaranteed. CGPredict models the cache behavior and the interplay between the computation latency and the memory access latency quite accurately.

2 BACKGROUND

In this section we present a brief background on the GPU architecture, the CUDA programming model as well as the concepts essential for performance modeling.

2.1 GPU Architecture

GPUs are prevalent in heterogeneous MPSoCs. We model embedded NVIDIA Kepler GPU architecture present in Tegra K1 SoC on Jetson TK1 development board [20] (see Figure 1). Kepler architecture is representative of the modern embedded GPUs in terms of power-performance characteristics. The GPU we model is equipped with one Streaming Multiprocessor (SMX) unit consisting of 192 CUDA cores, 32 special functional units, and 32 load/store units. It has 64KB on-chip memory that can be configured as shared memory or L1 cache, an on-chip 128KB L2 cache, and off-chip global DRAM memory shared with the on-chip CPU core.

2.2 Programming Model:

The Kelper GPU leverages CUDA [15] as its programming model. CUDA extends C by allowing programmers to define *kernels* that are executed in parallel by hundreds to thousands of CUDA threads with different data. A number of threads form a *thread block*, which is the unit for scheduling on SMX. Each thread is identified with two IDs: *blockID* and *threadID*. The threads within a block are further grouped into *warps* consisting of 32 threads each. Blocks are organized into one-, two-, or three-dimensional *grid* that represents the workload as shown in Figure 2.

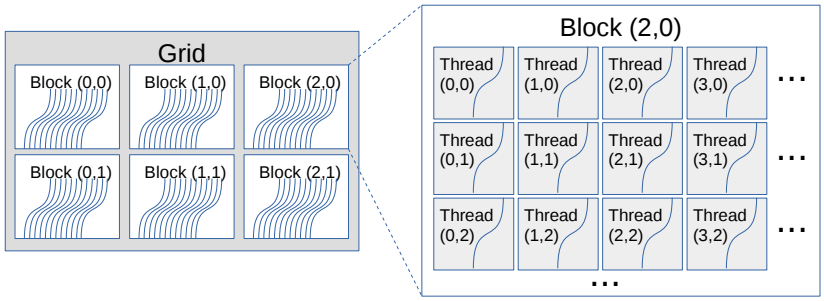

Fig. 2. CUDA threads organization

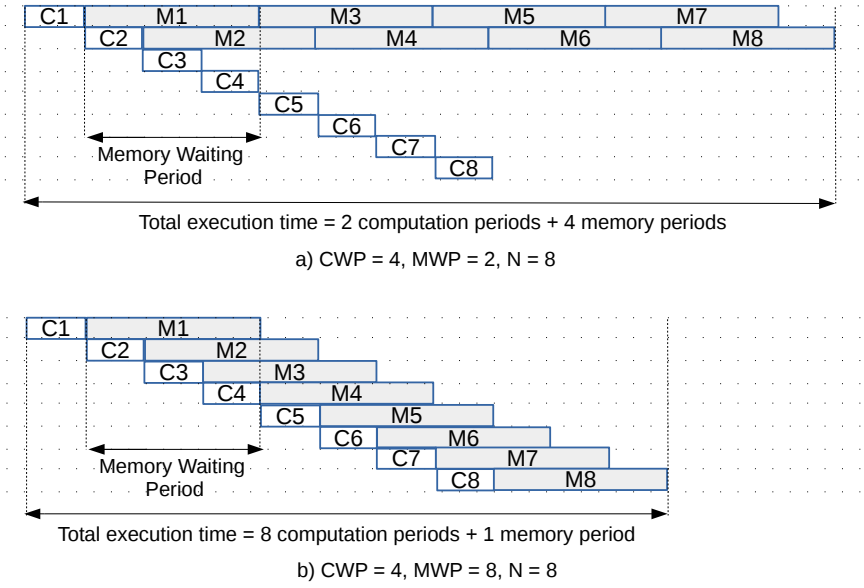

Fig. 3. Execution time visualization with Computation Warp Parallelism (CWP), Memory Warp Parallelism (MWP) and active number of warps (N).

2.3 Warp Scheduling

The unit of scheduling within SMX is warps. The SMX consists of four warp schedulers and each scheduler can issue two warp instructions each cycle. All the threads within a warp execute the same warp instruction in parallel on 32 CUDA cores in lock-step, but different warps can make independent progress. The warp scheduler issues the next available warp instruction when there are free CUDA cores available. Kepler GPUs employ aggressive latency hiding techniques when a memory access cannot be serviced immediately. The currently executing warp is context switched out and another available warp is scheduled instead.

Figure 3 shows a visualization of the latency hiding technique [7]. In Figure 3(a), let us assume that the architecture can service two memory warps concurrently and there are $N = 8$ warps waiting to execute their computation periods ($C1, \ldots, C8$) and memory periods ($M1, \ldots, M8$). A *computation period (comp_p)* is the execution of computation instructions in a warp before a memory access. A

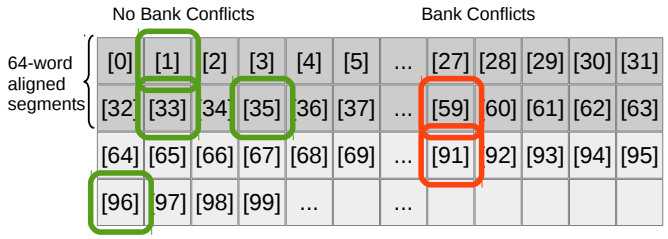

Fig. 4. Bank conflicts in Kepler shared memory

memory period (mem_p) is the execution of a memory access instruction. Instead of waiting for the warp to complete the memory access for the entire memory period, the next available computation period of a different warp is scheduled. Thus, the computations periods can be mostly hidden under the memory periods except for the first two warps and the total execution time comprises of only 2 computation periods and 4 memory periods.

This effect can be captured by the concept of memory warp parallelism (*MWP*) and computation warp parallelism (*CWP*) [7]. Memory periods from multiple warps can overlap depending on the memory bandwidth and memory bank parallelism; MWP represents the maximum number of warps per SMX that can access the memory concurrently during one memory period. Computation periods from consecutive warps do not overlap in the model. CWP represents the number of warps that the SMX can execute during one memory warp period plus one (the warp itself is waiting for memory). As MWP is 2 in this example, two computation periods are required before the memory bottleneck is reached, after which all the computation periods are hidden by memory periods.

In contrast, if the architecture can service more memory warps concurrently, memory accesses will no longer be the bottleneck as shown in Figure 3(b). In this example, $MWP = 8$, i.e., the memory can service 8 memory warps concurrently, while the CWP is still 4. Thus, the memory periods are mostly hidden except for the last warp, while the computation periods are all exposed. The total execution time therefore can be calculated as:

$$total\_cycle = \begin{cases} mem\_p \times \frac{N}{MWP} + comp\_p \times MWP, & \text{if } CWP \geq MWP \\ mem\_p + comp\_p \times N, & \text{if } CWP < MWP \end{cases}$$

2.4 Memory Access Patterns

Adjacent threads in a warp have high probability of accessing data from contiguous memory addresses; thus coalescing of memory accesses within a warp helps improve performance by reducing the number of transactions to fetch data from the memory. However, not all memory instructions within a warp can be coalesced. An analysis of the memory access pattern of the kernel is essential to predict the execution performance.

2.5 Shared Memory Bank Conflicts

In Kepler architecture, the on-chip shared memory has 32 banks that are each 8 bytes wide. Successive 4-byte words are mapped to successive banks. With certain access patterns, shared memory can have 256 bytes (32 banks × 8) bandwidth per cycle. Figure 4 shows an illustration of shared memory bank configuration and bank conflicts. Each box represents a word and the number represents its address. The 32 columns represent the 32 banks present in the architecture. Words in the same column (*[0][32][64][96]*) belong to the same bank. As each bank is 8 bytes wide, the words

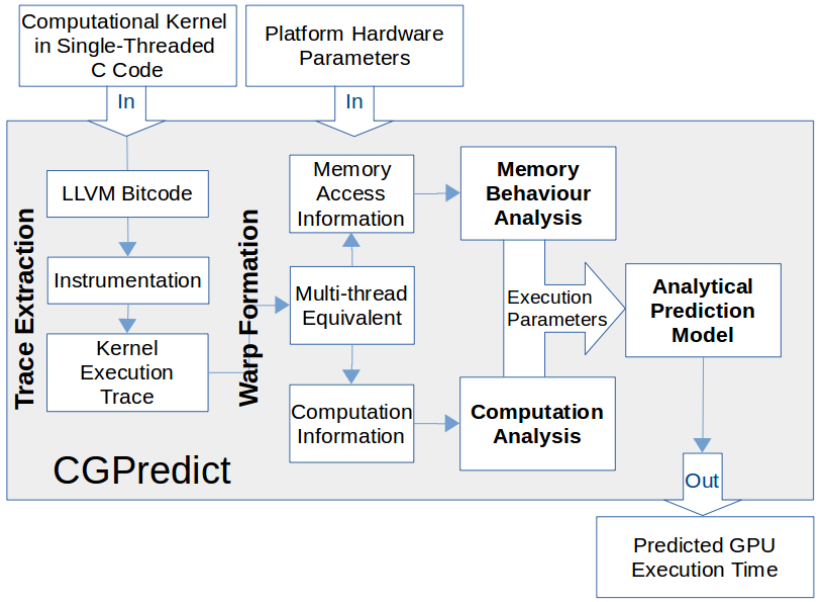

Fig. 5. CGPredict framework overview

within 64-word (32 banks × 2 words per 8-byte) aligned segments (*[0]* and *[32]*) can be accessed simultaneously even if they belong to same bank. In the default 4-byte access mode, bank conflicts occur if two or more threads access 4-byte words from the same bank that spans multiple 64-word aligned segments (*[59]* and *[91]*). For N threads in a warp conflict, called N-way bank conflict, the memory access instruction gets replayed $(N - 1)$ times. There is no bank conflict when accessing different banks (*[96]* and *[35]*), the same word (multiple accesses to *[1]*), or in the bank within one 64-word aligned segment (*[1]* and *[33]*).

3 CGPREDICT FRAMEWORK

To aid the developer in the early design decision about the accelerator, we present an analytical framework *CGPredict* to predict the performance of a computational kernel on an embedded GPU architecture from un-optimized, single-threaded C code. The overview of CGPredict is shown in Figure 5.

CGPredict takes a computational kernel in the form of single-threaded C code as input and generates its execution trace through a **Trace Extraction** phase. In order to emulate the behavior of GPU, a **Warp Formation** phase is introduced to transform the single-threaded trace into its multi-threaded equivalent. CGPredict then extracts computation (in the form of compute instructions) and memory access information. Compute instructions are mapped to CUDA PTX ISA [16] to predict the number of GPU instructions, and thus *compute cycles* in **Computation Analysis** stage. To predict GPU memory cycles, CGPredict takes the memory access information and analyzes its access patterns and cache behavior in **Memory Behavior Analysis** stage. The results from the two analysis stages complete the execution characteristics we need from the kernel for performance prediction. Lastly, together with the architectural parameters obtained by micro-benchmarking [11, 21], an **Analytical Prediction Model** is engaged to predict the final execution performance using the computation and memory execution characteristics.

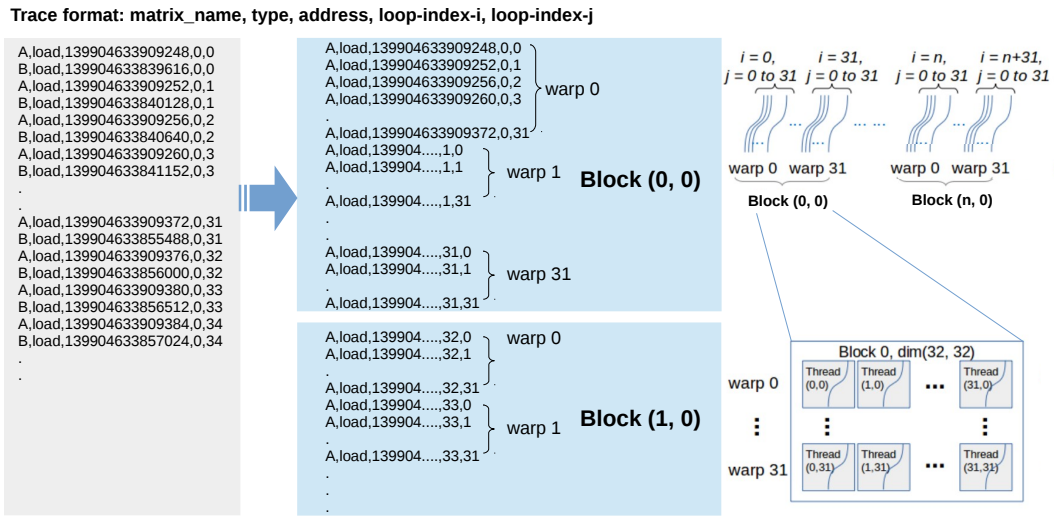

Fig. 6. Warp formation (trace transformation)

3.1 Trace Extraction

CGPredict leverages the Low-Level Virtual Machine (LLVM) [9, 25] for trace collection. It converts single-threaded C code into an LLVM intermediate representation (LLVM-IR). LLVM-IR is machine independent and is the core of LLVM. CGPredict then performs instrumentation by inserting a set of function calls in the generated LLVM-IR. These functions are used to record program characteristics such as runtime instances of static instructions, operation types, operands, load/store addresses and loop information (number of loops, iteration indices). The size of the trace is determined by the input sizes defined in C code. While a small size is preferred to reduce trace generation overhead, the trace generated must be large enough to fully exploit the parallelism present on the GPU platform to reveal the actual execution characteristics (see Sec. 3.2). The designer only needs to insert pragmas into the original C code to highlight the portion of the code that should be analyzed by CGPredict in the trace extraction stage. Designers do not need to have prescient knowledge regarding the suitability or parallelizability of the code fragment as CGPredict performs the analysis automatically and informs the designer of the potential performance improvement with GPU acceleration.

Given the LLVM-IR trace, we separate it into a Memory Trace and an Operation Trace. The Memory Trace contains memory load/store operations with their address and loop information (loop indices). This information is used in the Warp Formation phase for converting the single-threaded trace to its multi-threaded equivalent as shown in Figure 6. The Operation Trace includes the non-memory operations and is used to evaluate the computation cost in GPU execution time prediction (Sec. 3.4).

3.2 Warp Formation

When a code segment in an application is repeatedly executed in the form of a loop, the inherent parallelism makes it ideal candidate for acceleration through GPU. Consider a nested loop within an application with multiple loop levels. The outer-most loop indices can be directly mapped to

the multi-dimensional IDs of the GPU threads, and the loop body can be mapped to the thread execution.

The following code segment presents a simple matrix multiplication in serial C code. It performs multiplication of matrices A and B of size $SIZE$ ($= N * N$) and puts the result in matrix C.

```
1 //Matrix multiplication C implementation
2 void mm(TYPE A[SIZE], TYPE B[SIZE], TYPE C[SIZE]) {
3   int i, j;
4   for (i = 0; i < N; i++) {
5     for (j = 0; j < N; j++) {
6         C[i*N + j] = A[i*N +j] * B[i*N + j];
7         }
8     }
9   }
```

For a two-dimensional grid for GPU, the outer-most two loops (loop i and j) can directly map to the *threadID.x*, and *threadID.y*, as illustrated in Figure 6. Line 6 containing actual calculation therefore form the kernel code, which maps to a thread execution.

The kernel execution trace (memory trace, left of Figure 6) we obtain from the trace extraction phase is single-threaded where all the loops are unrolled because it is a dynamic execution trace. The loop-index i and loop-index j are the outer-most loop indices. We can consider an iteration in the innermost loop as a "*pseudo-thread*". In this case, a "*pseudo-thread*" trace is memory load of A, B and memory write to C, shown as the first 3 lines in the single threaded trace. We then fold the trace to have these "pseudo-threads" side-by-side. A group of 32 "pseudo-threads" form a "pseudo-warp". The transformed trace (right of Figure 6) shows multiple warps executing concurrently the first instruction of the "pseudo-threads".

As the mappings of threads on SMX are done in blocks, the block size setting affects the memory access pattern and is reflected in the warp formation. We assume that the warps progress in the order of WarpIDs within a block, and blocks are scheduled sequentially [18].

Furthermore, because of the hardware restrictions of certain platforms, including $thread\_per\_sm$, which denotes the maximum number of threads that can be concurrently executed on one SMX (2048), the "pseudo-threads" need also be arranged in batches of 2048. Continuing with the discussion of trace size in Section 3.1, trace size of at least $2 \times thread\_per\_sm$ are therefore advisable to ensure the fully occupancy of SMX and capture the interaction of the working sets between the two batches. For MM application, the trace should be generated for at least $N = 64$ to have $64^2 = 4096$ iterations that maps into two "pseudo-thread" batches.

We consider inter-thread collaboration and synchronization cost incurred due to the shared memory usage (Section 3.6). We assume that no inherent inter-thread communication is required when the original C implementation is parallelized into pseudo-threads in the process of warp formation. This assumption holds good for most kernels suitable for GPU acceleration. In the presence of inter-thread communication, CGPredict detects and informs the developer of the potential synchronization issues.

The formation of warps through trace transformation exposes the available thread-level parallelism that can be exploited by the GPU. We then extract the memory access information for memory analysis (Section 3.3) and computation information for computation analysis (Section 3.4). Note that the transformed trace may not be the exact replica of the actual GPU trace from equivalent CUDA code; but it is sufficiently close for performance estimation. Also, we do not need to generate functionally correct CUDA code from C. Instead, we focus on aiding the developer with

the high-level choice of accelerator, that is, whether the GPU is a good match for the application kernel. Therefore, CGPredict can tolerate certain discrepancies as long as the estimated performance is quite accurate.

3.3 Memory Behavior Analysis

From the kernel execution trace, we extract memory access address trace for memory behavior analysis, including classification of memory access patterns and cache miss performances. The information is plugged into our memory access latency model. We consider embedded GPUs where CPU, GPU reside on the same chip and share off-chip DRAM. Thus, unlike discrete GPUs, we do not need to consider data transfer overhead between the host (CPU) and the device (GPU). The overhead of data transfer from DRAM to the on-chip memory (cache or shared memory) is modeled carefully.

3.3.1 Memory Configuration. The micro-architecture of the GPU platform determines the execution performance. The introduction of caches into the GPU architecture improves the memory access latency, while the unpredictability of cache access latency increases the complexity of the performance estimation.

Name	Shared Memory	L1 Cache	L2 Cache	DRAM
Size	48/32/16 KB	16/32/48 KB	128 KB	1892 MB
Cache Line (B)	-	128	64	-
Latency (cycles)	67	67	164	332

Table 1. Memory configuration of Jetson TK1 Kepler GPU

We first extract the cache specifications of the Jetson TK1 Kepler GPU. As no documentation is available for the detailed information about caches, the configurations shown in Table 1 are obtained by running micro-benchmarks. The results are cross-validated using two different tools [11, 21]. Moreover, performance estimation with our CGPredict framework using these cache configuration parameters produces low performance estimation error.

Given a variable, the programmer can specify the allocation of the variable in shared memory or read-only data cache through CUDA intrinsics. If unspecified, the data memory accesses by default goes to the L2 cache and to the Global Memory if it misses in the L2 cache. The L1 cache is reserved only for local memory accesses, such as register spills and stack data [17]. As the memory hierarchy model is easily extendable to multiple levels of caches, to ease the explanation, we assume that the memory hierarchy contains only the L2 cache and the DRAM (global memory) in our discussions for ease of explanation.

3.3.2 Classification of Memory Access Patterns. The memory access patterns observed in the kernels can be categorized as:

- Coalesced Access: The memory accesses within a warp are accessing adjacent memory addresses and therefore can be coalesced together as one (or a few) memory transactions.
- Uncoalesced Access: The memory accesses within a warp are accessing non-adjacent memory addresses and thus cannot be coalesced to few memory transactions. Generally, 32 (number of threads in a warp) memory transactions are required to complete the memory operation.
- Constant Access: All the 32 threads in a warp are accessing the same memory address and therefore only one memory transaction is required.

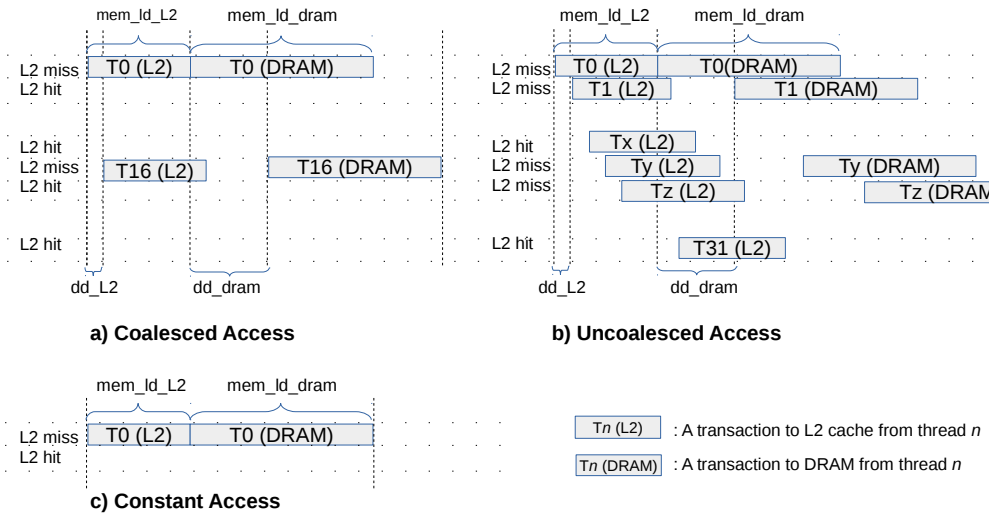

Fig. 7. Memory behavior of different access patterns

The different access types can be analyzed from the multi-threaded memory access information obtained from warp formation. This is achieved by calculating the memory access stride of the memory instructions within a "pseudo-warp". Memory warp instructions with maximum access stride of one data element (4 byte) are classified as coalesced access, maximum access stride of 0 are classified as constant access, while the remaining ones are classified as uncoalesced access.

3.3.3 Memory Access Latency Estimation. With memory access pattern information, we then analyze their effects on memory access latencies in the hierarchical memory architecture. As discussed in previous sections, we consider the memory hierarchy containing the L2 cache and the DRAM (global memory). The cache behavior of the three types of memory instructions will largely affect the execution performance, mainly in terms of memory access time (*mem_l*) and time delay between consecutive memory accesses (*depature_del*). The memory access behavior within a warp for different memory access patterns are shown in Figure 7. The parameters used in the discussion are summarized in Table 3.

As three types of memory access patterns exist in the applications, taking an average across all the different memory instructions will not lead to a good estimation. While memory instructions with different types may have very different access latencies, the memory instructions with the same access pattern (for example coalesced accesses) have roughly similar execution times. Therefore, for a more accurate estimation, we estimate the average behavior of the memory instructions with similar access patterns. Here we explain the detailed model of a warp memory instruction at thread level for the three different memory access patterns, as shown in Figure 7.

Coalesced and Un-coalesced Access. In coalesced accesses, the memory accesses are coalesced into one or more memory transactions to fetch the data from the cache in cache line size granularity. As shown in Figure 7(a), an L2 cache line contains 16 data elements. Therefore, two cache transactions are generated from one coalesced memory warp instruction with 32 memory operations. If any of these cache transactions results in a cache miss, then a memory transaction to the off-chip global DRAM memory will be initiated. For un-coalesced accesses, as the memory addresses are not

adjacent, each thread generates an independent memory transaction to the L2 cache and possibly to DRAM, as shown in Figure 7(b).

In both cases, the memory access time is determined by how many memory transactions are generated per warp for coalesced/uncoalesced access ($no\_(un)coal\_pw$) and how many DRAM transactions ($no\_dram\_trans\_(un)coal$) are generated per warp due to the cache misses. Therefore, the memory access time and the departure delay (the minimum time interval between the initiation of two memory transactions to the same memory) for coalesced and un-coalesced memory instructions per warp can be calculated as:

if $no\_dram\_trans\_(un)coal \leq 1$,

$$mem\_l\_(un)coal = mem\_ld\_L2 + (no\_(un)coal\_pw - 1) \times dd\_L2 \quad (1)$$

if $no\_dram\_trans\_(un)coal > 1$,

$$mem\_l\_(un)coal = mem\_ld\_L2 + mem\_ld\_dram + (no\_dram\_trans\_(un)coal - 1) \times dd\_dram \quad (2)$$

$$dep\_del\_(un)coal = max\{no\_(un)coal\_pw \times dd\_L2, no\_dram\_trans\_(un)coal \times dd\_dram\} \quad (3)$$

Constant Access. For constant access patterns, as shown in Figure 7 (c), only one memory address is accessed by all the threads in a warp. Thus only one memory transactions is generated. The number of DRAM transactions per warp for constant access pattern, denoted as $no\_dram\_trans\_const$, can therefore only have the value of 0 (cache hit) or 1 (cache miss).

$$mem\_l\_const = mem\_ld\_L2 + no\_dram\_trans\_const \times mem\_ld\_dram \quad (4)$$

$$dep\_del\_const = no\_const\_pw \times dd\_l2 + no\_dram\_trans\_const \times dd\_dram \quad (5)$$

With the detailed access time information of the three different memory access types, we can then have a more accurate estimation of the total memory access latency *mem_cycles*, the average memory access latency per memory warp instruction across all access types *mem_l*, and the average departure delay for a warp memory instruction across all access types *departure_delay*.

$$\begin{aligned} \textbf{\textit{mem\_cycles}} &= mem\_l\_coal \times no\_coal\_insts + mem\_l\_uncoal \times no\_uncoal\_insts \\ &\quad + mem\_l\_const \times no\_const\_insts \end{aligned} \quad (6)$$

$$\textbf{\textit{mem\_l}} = mem\_cycles \;/\; no\_mem\_insts \quad (7)$$

$$\begin{aligned} \textbf{\textit{depature\_delay}} &= (dep\_del\_coal \times no\_coal\_insts + dep\_del\_uncoal \times no\_uncoal\_insts \\ &\quad + dep\_del\_const \times no\_const\_insts) \;/\; no\_mem\_insts \end{aligned} \quad (8)$$

3.3.4 Cache Miss Estimation. We design a cache analyzer to estimate the number of off-chip DRAM transactions for different memory instruction types. The cache analyzer predicts the behavior of the L2 cache given the cache configuration and the memory traces using the reuse distance theory. It can thus estimate the L2 cache miss rate and generate the number of DRAM accesses per warp memory instruction, averaged across all memory instructions of same memory access types.

There are three major parameters for a cache configuration: *cache block size B*, *number of sets K* and *associativity A*. The cache size can be calculated as $(K \times A \times B)$. Let $\mathcal{T}$ be the input memory address trace in Section 3.2. The accesses to memory are in granularity of blocks with size B. Thus $\mathcal{T}$ is first converted into a block address trace T by eliminating the least significant $(log_2 B)$ bits. A memory block address m is mapped into the i^{th} cache set C_i, where $i = m \; modulo \; K$ and $i \in [0, K)$. We use M_i to denote the set of all memory blocks that are mapped to C_i. There is no interference

between different cache sets. Therefore the cache miss behavior can be analyzed independently for each set. Trace T can be partitioned into K traces: T_1, T_2, ... ,T_K, one for each cache set. For a given memory block address $m \in M_i$, we define $m[j]$ to be the j^{th} reference and N_m to be the total number of references of m in the sub-trace T_i.

We borrow the concept of Temporal Conflict Set (TCS) from [10]: Given a memory block reference $m[j]$ in the subtrace T_i, where $j > 1$, $i \in [0, K)$ and $m \in M_i$, the temporal conflict set $TCS_{m[j]}$ is defined as the set of unique memory blocks referenced between $m[j-1]$ and $m[j]$ in T_i. $TCS_{m[j]} = \emptyset$ indicates no such references.

Clearly, if $|TCS_{m[j]}| \geq A$, reference $m[j]$ will be a cache miss; if $|TCS_{m[j]}| < A$, reference $m[j]$ will be a cache hit. The analysis of $TCS_{m[j]}$ is performed for all N_m references of m in T_i.

$$hit(m[j]) = \begin{cases} 1, & \text{if } |TCS_{m[j]}| < A \text{ and } j > 1 \\ 0, & \text{otherwise} \end{cases}$$

$$num\_hit(m) = \sum_{j=1}^{N_m} hit(m[j]) \tag{9}$$

The total number of cache hits for a cache set C_i and the entire memory trace are therefore:

$$num\_hit(T_i) = \sum_{m \in M_i} num\_hit(m) \tag{10}$$

$$num\_hit(T) = \sum_{i=1}^{K} num\_hit(T_i) \tag{11}$$

We compare the performance of the cache analyzer against a commonly used cache simulator Dinero [5] and verify that our cache analyzer has 99% prediction accuracy.

3.4 Computation Analysis

Regardless of the execution platform, the computations and memory operations performed by CPU and GPU are quite similar. We obtain the CPU computation instructions (LLVM-IR) information from the trace and predict the GPU computation instructions performance.

Parallel Thread Execution (PTX) is a pseudo-assembly language used in NVIDIA's CUDA programming environment [16]. The binary code to be run on the GPU processing cores are translated from PTX code by a compiler in the graphics driver. Although PTX code is not a direct representation of the actual machine code, it is an accurate enough representation of the native CUDA code that captures more GPU characteristics. After careful consideration, we find the mapping from LLVM-IR to CUDA PTX instructions as shown in Table 2. From the instruction counts, *comp_cycles* can therefore be calculated as shown in the following equation:

$$\boldsymbol{comp\_cycles} = inst\_cycle \times no\_total\_insts \tag{12}$$

3.5 Putting it All Together

So far, we have discussed how we can extract the application features into execution parameters through analytical methods taking into consideration the platform-specific hardware parameters. We have performed this analysis separately for memory operations and computation operations. Finally, CGPredict engages an analytical model to estimate the overall program execution time.

Table 3 summarizes all the parameters required in the model. The first part includes platform-dependent parameters that are obtained by carefully examining the hardware platform, running

LLVM-IR Instruction	PTX Instruction	GPU Instruction
load, store	ld, st	memory instruction
add, mul, shl, br	add, mul, shl, br	compute instruction
fmul + fadd	fma	compute instruction
loop	1 add and 1 branch	compute instruction

Table 2. Instruction mapping between LLVM-IR and PTX

devicequery and micro-benchmarks [11][21] on the platform . The second part summarizes the kernel execution parameters that are obtained from the trace analysis as described in the previous sections.

We adopt the analytical model presented in [7] with the concept of MWP (memory warp parallelism) and CWP (computation warp parallelism) as discussed in Section 2. The idea is to model the effect of latency hiding of either the computation operations or the memory operations depending on the availability of memory-level parallelism versus computational parallelism. The model is given below where mem_l and $departure\_delay$ are calculated from Section 3.3.3 through cache and DRAM analysis.

$$MWP = \frac{mem\_l}{departure\_delay} \tag{13}$$

$$CWP = \frac{mem\_cycles + comp\_cycles}{comp\_cycles} \tag{14}$$

In addition, the value of MWP and CWP are bounded by N, the number of active running warps existing on one SMX. The number of active running blocks (B) and N can be estimated with application kernel settings (problem size, block size, shared memory usage) and architecture support (maximum number of threads per block, available shared memory size), as suggested in CUDA occupancy calculator [14]. The number of batches of thread execution ($batch$) can therefore be calculated by Eqn (15) with the total number of blocks for the kernel (no_blocks) and B. We can then calculate the execution cycles from MWP and CWP by Eqn. (16, 17) and finally the execution time in seconds with platform frequency information.

$$batch = no\_blocks/B \tag{15}$$

if $CWP \geq MWP$

$$exec\_cycles = mem\_cycles \times \frac{N}{MWP} + \frac{comp\_cycles}{no\_mem\_insts} \times MWP \times batch \tag{16}$$

if $CWP < MWP$

$$exec\_cycles = mem\_l + comp\_cycles \times N \times batch \tag{17}$$

$$exec\_time = exec\_cycles \; / \; freq \tag{18}$$

3.6 Shared Memory Consideration

Although the cache hierarchy brings the data closer to the GPU, the limited size of the caches as well as the memory access patterns of certain kernels may still result in minimal benefit from caching. For applications that can be tiled, the utilization of the shared memory can largely reduce the data access latencies. Programming efforts are required in determining the portion of data to be put into shared memory and the tile size. The algorithm may also need to be modified to be tiled

Parameter Name	Definition	From/Value
freq	clock frequency of GPU	852 MHz
inst_cycle	average number of cycles to execute one instruction	0.5
mem_ld_L2	access latency of L2 cache	164
mem_ld_dram	access latency of DRAM	332
mem_ld_smem	access latency of shared memory (which shares the same physical on-chip storage as L1 cache)	67
smem_load_const	latency for loading from main memory to shared memory	506
dd_L2	departure delay, the delay between two memory transactions to L2 cache	2
dd_dram	departure delay, the delay between two memory transactions to DRAM	10
X_uncoal, X_coal, X_const	X-way bank conflict caused by uncoalesced, coalesced or constant memory accesses	16 / 1 / 1
no_uncoal_pw, no_coal_pw, no_const_pw	number of L2 transactions generated for a warp memory access instruction of uncoalesced, coalesced or constant memory access pattern	32 / 2 / 1
no_dram_trans_uncoal, no_dram_trans_coal, no_dram_trans_const	number of DRAM transactions generated from a warp memory instruction of uncoalesced, coalesced or constant memory access pattern	Sec. 3.3
dep_del_uncoal, dep_del_coal, dep_del_const	depature delay, the delay between two warp memory instruction dispatches of uncoalesced, coalesced or constant memory access pattern	Sec. 3.3
mem_l_uncoal, mem_l_coal, mem_l_const	memory access latency for a warp memory instruction of uncoalesced, coalesced or constant memory access pattern	Sec. 3.3
no_mem_insts	number of total memory instructions	Sec. 3.4
no_uncoal_insts, no_coal_insts, no_const_insts	number of memory instructions of uncoalesced, coalesced or constant memory access pattern	Sec. 3.4
no_comp_insts	number of total compute instructions	Sec. 3.4
no_smem_insts	number of total share memory access instructions	Sec. 3.6
no_sync_insts	number of total synchronization instructions	Sec. 3.6
no_total_insts	number of all instructions (mem, comp)	Sec. 3.4
B, N	B: no of active running blocks per SMX, N: no of active running warps per SMX	Sec. 3.5, [14]

Table 3. Summary of model parameters

in some cases. The decisions are to be made based on the data usage of the application and the share memory size available for the architecture. The loop tiling is performed on the sequential code. Given these hints by the programmer (regarding data elements that should be brought into shared memory and the tiling information), the accesses to the shared memory can be extracted out from the sequential memory access trace.

In a shared memory implementation, each thread in a block brings in one (or more) data element from main memory. Together all the threads in a block bring in all the data elements required for execution in this block. The latency of these memory accesses is predictable and can be estimated with a fixed load latency (Eqn. 19). A thread barrier is inserted to ensure all the data elements are loaded before execution.

Secondly, during the execution, the latency of a shared memory access depends on bank conflicts. A X-way bank conflict will result in X times longer latency than zero bank conflict case. To predict bank conflicts, an access pattern analysis similar to the discussion in Section 3.3.2 is performed for the memory access trace. This analysis determines the number of warp memory instructions with X-way bank conflict where X can vary from 1 to 32. The access latency can then be estimated with (Eqn. 20). The rest of the analysis then follows the same way as discussed in the previous sections.

In addition, as synchronization barriers are required in shared memory implementation, additional synchronization cost (*sync_cost*) is added to the final execution time. The synchronization cost is calculated as the departure delay of memory instructions times the number of warps that can access the memory concurrently. This is essentially the waiting time of warps that have finished the current memory period but cannot schedule the next memory period. This value is further multiplied by the number of synchronization instructions and the number of active running blocks [14].

$$smem\_load\_cycles = no\_smem\_load\_inst \times smem\_load\_const \tag{19}$$

$$\begin{aligned} smem\_cycles = {} & mem\_ld\_smem \times X\_uncoal \times no\_uncoal\_insts \\ & + mem\_ld\_smem \times X\_coal \times no\_coal\_insts \\ & + mem\_ld\_smem \times X\_const \times no\_const\_insts \end{aligned} \tag{20}$$

$$\boldsymbol{mem\_cycles} = mem\_cycles + smem\_load\_cycles \tag{21}$$

$$\boldsymbol{comp\_cycles} = comp\_cycles + smem\_cycles \tag{22}$$

$$sync\_cost = departure\_delay \times (MWP - 1) \times no\_synch\_insts \times B \times batch \tag{23}$$

3.7 Limitations

CGPredict, similar to any dynamic analysis tools based on profiling, may not achieve accurate performance estimation if the behavior of the application varies significantly across different inputs. In such cases, it is imperative to carefully select representative program inputs for trace generation. Fortunately, application kernels that can potentially benefit from GPU acceleration present relatively stable behavior across different inputs. Moreover, as explained in Section 3.2, CGPredict ensures that the input trace is of sufficient size to capture the interaction among the threads and their memory behavior. Note that CGPredict can accurately estimate performance for different input sizes irrespective of the profiling input size. In addition, CGPredict targets the NVIDIA GPU architecture and can be easily re-targeted to any NVIDIA GPU architecture by simply changing the hardware-specific parameters in the first part of Table 3. These parameters can be easily obtained through standard benchmarking and/or from architectural specifications. However, the architecture of non-NVIDIA GPUs, for example, ARM Mali GPU, can be vastly different requiring substantial changes to our framework. Furthermore, CGPredict works well for applications ideally suited for GPUs with inherent data parallelism and little inter-thread dependencies. For applications requiring data sharing among pseudo-threads after warp formation in Section 3.2, CGPredict reports this dependency but currently cannot insert the synchronization primitives automatically. The

developer needs to manually insert the synchronizations to accurately evaluate the feasibility of GPU acceleration for the application.

4 EXPERIMENTAL EVALUATION

We now evaluate CGPredict framework on an embedded GPU.

4.1 GPU Performance Prediction Quality

To evaluate the estimation accuracy of CGPredict, we use the NVIDIA embedded Kelper GPU on Jestson TK1 development board [20]. For benchmark applications, we select Polybench benchmark suite [6] because each application is available in both sequential C version and the corresponding CUDA code. Different implementations of the same algorithm on different platforms ensure the fairness when comparing the predicted performance obtained through CGPredict analyzing the single-threaded C code against the real execution time of the threaded CUDA code on Jetson TK1.

Table 4 shows the characteristics of the benchmarks as well as the estimation accuracy. The column *Work Size* is the size of workload in a single dimension. The *Work Size* of 4096 in a two-dimensional grid means that the total work size is 4096×4096, while in a one-dimensional grid means 4096×1. The block size is set to be 32×32 and 256×1 for 1D and 2D grid, respectively. Note that CGPredict estimates the execution time based on a trace generated by a small portion of the workload (and not the entire workload) as mentioned in Section 3.2. The average estimation error for CGPredict is 9.00% across all the 15 kernels, demonstrating the high accuracy of CGPredict.

The analysis time of CGPredict includes the generation and analysis of traces, including trace transformation and cache analysis. The trace generation from C code usually takes seconds to minutes depending on the trace size, shown in Table 4. Though the whole trace is generated for the application, CGPredict only extracts part of the trace for warp formation and cache analysis, resulting in short analysis time. CGPredict trace generation plus analysis time ranges from 1 to 5 minutes for all the benchmarks.

Looking into the details of the evaluation results, we can make some interesting observations. For example, SYRK and GEMM have very similar algorithms in C implementation. However, their GPU performances are quite different. From the memory behavior analysis of CGPredict, we can infer that half of the memory instructions in GEMM are coalesced access type, while the other half are constant access type. In contrast for SYRK, half of the memory instructions are constant access with the other half being uncoalesced access. With this coalescing information, CGPredict predicts MWP of GEMM to be 54.45, which is higher than the MWP of SYRK (5.97), and leads to a much shorter execution time. This can be further justified by the profiling information of the CUDA version of the two benchmarks from *nvprof* [14]. While the same instruction counts are observed, SYRK generates more global memory transactions compared to GEMM due to extensive uncoalesced memory accesses. Thus, SYRK has much worse performance. This suggests possible coalescing of such memory accesses to achieve better performance.

Moreover, to test the sensitivity of CGPredict to input workload size, we evaluate the estimation accuracy of CGPredict by changing the input workload size for 10 benchmarks, as shown in Figure 8. The *Input Size* bar stands for the estimation error as reported in Table 4. The other two bars are with workload size that are a half and a quarter of the workload size reported in Table 4. The estimation error remains low with varying input workload size, demonstrating that the prediction accuracy of CGPredict is stable across different input size.

Benchmark Name	Work Size	Grid Dim.	Trace Size	Actual Time (ms)	Estimated Time (ms)	Estimation Error (%)
2DCONV	4096	2	512	29.52	28.01	5.13
2MM	4096	2	128	16294.07	15518.11	4.76
3DCONV	4096	2	64	84.54	68.30	19.20
3MM	2048	2	128	5990.76	5819.73	2.86
ATAX	4096	1	1024	201.70	193.04	4.29
BICG	4096	1	1024	237.69	199.22	16.19
CORR	1024	1	128	3071.66	2678.05	12.81
COVAR	1024	1	128	3073.58	3465.71	12.76
FDTD-2D	4096	2	64	1492.23	1243.21	16.69
GEMM	1024	2	128	249.16	242.53	2.66
GESUMMV	4096	1	1024	680.85	769.69	13.05
GRAMSCHM	8192	1	512	43.79	45.58	4.09
MVT	4096	1	1024	215.96	193.04	10.61
SYR2K	1024	2	128	5430.54	5204.73	4.16
SYRK	1024	2	128	2762.50	2605.45	5.69
					Average Estimation Error	9.00

Table 4. CGPredict GPU performance estimation accuracy

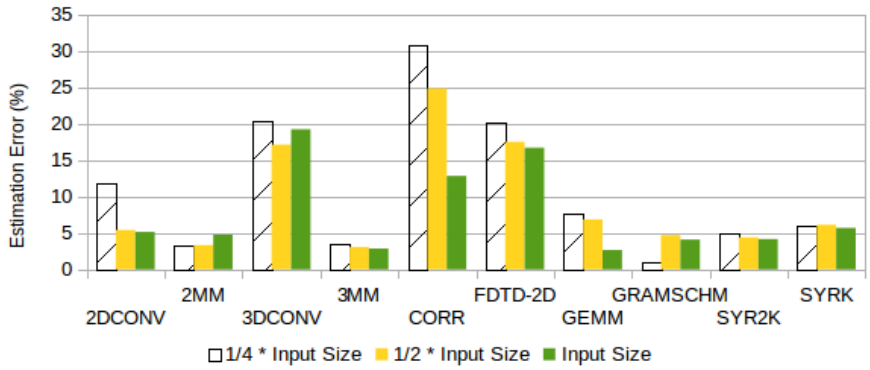

Fig. 8. Sensitivity of CGPredict estimation accuracy to input workload size

4.2 Cache Modeling

One of the important contributions of CGPredict is to analyze the cache behavior of the architecture. In order to evaluate the accuracy of the cache model of CGPredict, we compare the estimation accuracy of CGPreidct (with cache modeling) against a baseline estimation method with simplistic cache modeling. The baseline estimation methods have the same architectural parameters and same application parameter inputs as CGPredict. The analytical model used in the baseline estimation approach is similar to [7], which is also used by CGPredict in the final stage. Instead of the detailed cache and DRAM modeling of CGPredict, a simple cache miss rate value obtained by cache simulator Dinero [5] is used in the baseline model. The memory access latencies and departure delay values are calculated as a simple weighted average of the respective values of the L2 cache and the main memory, as show in Eqn. (24,25), where M stands for the different memory access patterns (*uncoal*,

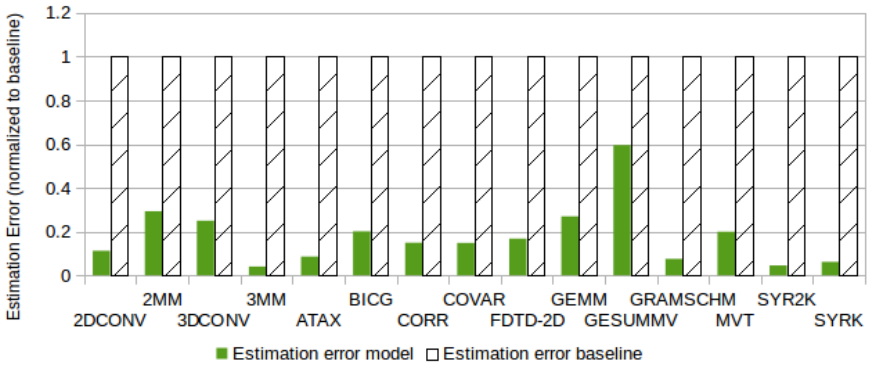

Fig. 9. Esimation error comparison of CGPredict and baseline (simple cache model)

Benchmark Name	Work Size	Tile Size	Smem Size (B)	Actual Time (ms)	Estimated Time (ms)	Estimation Error (%)
2MM	4096	32	8192	7019.37	6748.05	3.87
3MM	2048	32	8192	2538.99	2530.52	0.33
GEMM	1024	32	8192	112.44	105.75	5.96
SYR2K	1024	32	16384	1468.68	1445.48	1.58
SYRK	1024	32	8192	721.89	713.42	1.17

Table 5. CGPredict estimation accuracy with shared memory

coal, *const*, respectively) as discussed in Section 3.3. Figure 9 shows that CGPredict reduces the estimation error significantly compared to the baseline model.

$$mem\_l\_M = mem\_ld\_L2 \times (1 - L2\_miss) + mem\_ld\_dram \times L2\_miss \quad (24)$$

$$depature\_delay\_M = dd\_L2 \times (1 - L2\_miss) + dd\_dram \times L2\_miss \quad (25)$$

4.3 Shared Memory Modeling

To evaluate the accuracy of CGPredict in the presence of shared memory, we select few two-dimensional benchmarks from the Polybench benchmark suite. In order for CGPredict to work with the shared memory, the C implementation of each benchmark is manually modified to be tiled with tile size (32×32). The CUDA version of the benchmarks are also manually transformed for shared memory usage. Table 5 shows the estimation accuracy.

In general, usage of shared memory results in 2X to 4X performance improvement for the applications. But more importantly, CGPredict is able to predict the performance of the shared memory implementation from the tiled C code with high accuracy. For SYRK benchmark, though the shared memory version eliminates the uncoalesced accesses to the cache and the global memory, the inherent uncoalesced data access pattern causes significant bank conflicts in shared memory accesses. Thus, the performance of SYRK is worse compared to GEMM even with shared memory version.

Name	**SYRK**	**GEMM**
Execution Time	2762.50 ms	249.16 ms
Optimization	Memory access coalescing	Shared memory
Estimated Optimized Time	237.53 ms	105.75 ms
Actual Optimized Time	250.37 ms	112.44 ms
Estimation Error	5.13 %	5.96 %
Performance Improvement	~11X	~2X

Table 6. Results for application-specific optimizations

4.4 Suggestions for Optimizations

CGPredict can not only generate the performance prediction for a kernel on a GPU platform accurately with short analysis time, it can further provide insights for users to develop application-specific optimizations. CGPredict analyzes the memory access pattern of the application, provides information about memory coalescing, and suggests possible bottlenecks. With these information, the programmer can further develop optimizations including shared memory and coalescing of memory accesses. Table 6 shows two examples of such optimizations to achieve better performance.

4.4.1 Coalescing of Memory Accesses. Continuing the discussion in Section 4.1, although SYRK and GEMM have similar algorithmic structures, their execution times are very different. CGPredict evaluates the memory access patterns of all the memory instructions, and points out the bottleneck of the execution through MWP and CWP values. For SYRK, there are in total 2048 memory instructions (per thread), of which 2014 instructions are uncoalesced accesses. These uncoalesced memory accesses result in a very low MWP value (5.97) compared to CWP value (64).

To improve the performance of SYRK, we can coalesce the memory accesses by manipulating the memory access patterns. We observe that the threads within a warp in SYRK are accessing a matrix in a column-wise direction, resulting in uncoalesced accesses. To coalesce such accesses, we can pre-transpose the matrix before the actual kernel execution to have row-wise coalesced access pattern within a warp. We modify the original C code to have the matrix transposed and estimate the performance again using CGPredict. The estimated execution time reduces to 237.53 ms, from the original estimated execution time of 2605.45 ms. To verify the effectiveness of coalescing of memory accesses as well as the estimation accuracy, we also manually modify the CUDA implementation. The actual execution time of the optimized application is shown in table 6. The performance of SYRK is improved by 11X through the coalescing of the previously uncoalesced memory accesses.

4.4.2 Usage of Shared Memory. For benchmarks like GEMM, we observe from CGPredict that the memory accesses are already coalesced. As GEMM can be tiled, additional optimizations can be performed using the shared memory as shown in Table 6.

4.5 Choice of Accelerator

With the emergence of heterogeneous architectures (e.g., XILINX ZYNQ UltraSCALE+ [24]) consisting of CPU, GPU and FPGAs, assisting the designers in selecting the appropriate accelerator (GPU or FPGA) for a given application is of great importance. We now evaluate the potential usage of CGPredict in conjunction with an FPGA performance predictor [26]. The performance predictor can accurately estimate FPGA performance in the early design stage starting with single-threaded C code. We use five benchmarks from [26] in this set of experiments. Equivalent CUDA code are

Benchmark Name	Input Size	Estimated Time (ms)		Actual Time (ms)		Choice of Platform
		GPU	FPGA	GPU	FPGA	
MM	1024	**242.51**	1180	**250.27**	1450	GPU
MVT	2048	48.31	**9.09**	42.371	**10.41**	FPGA
GEMVER1	2048	**2.61**	16.55	**4.57**	19.81	GPU
DERICHE1	1024	**0.95**	2.99	**1.53**	3.37	GPU
DCT1D	1024	2697.75	**636.47**	2685.362	**650.8**	FPGA

Table 7. Accelerator choice between GPU and FPGA

implemented manually and executed on the Jetson TK1 for verification. We use an embedded FPGA, Xilinx ZC702 [24] with 100MHz frequency.

Table 7 shows that CGPredict along with the FPGA performance predictor can suggest the correct accelerator (GPU or FPGA) for each application. For MM, GEMVER1 and DERICHE1, GPU is better choice than FPGA because (a) GPU in TK1 has much higher frequency (852MHz) compared to FPGA (100MHz); (b) TK1 has much higher memory bandwidth (17GB/s) compared to FPGA in ZC702 (4GB/s); and (c) the coalesced memory access pattern of MM, GEMVER1 and DERICHE1 can significantly reduce memory transactions of GPU implementations and improve performance.

For MVT and DCT1D, the FPGA is better compared to the GPU. Both MVT and DCT1D have uncoalesced memory access patterns and GPU suffers from extensive memory transactions. Different from GPU implementations, FPGA accelerator first loads input data of several tiles into its local memory and start computation. Memory access patterns do not have large impact on FPGA performance, as access latency of FPGA local memory is quite small. It should be noted that GPU performance could be improved by several optimizations such as data layout transformation, loop tiling with shared memory and vectorization. However, the reference CUDA code that we are comparing against do not include such optimizations and hence we refrain from using them.

In addition, for MM, MVT and DCT1D in Table 7, the estimation errors of GPU performance prediction are quite low. For GEMVER1 and DERICHE1, the error is relatively high. These two benchmarks have quite small runtime compared to the others, and are thus highly sensitive to small differences in actual runtime due to external factors. But both are still reasonable estimations.

5 RELATED WORKS

Lots of research efforts have been put into the performance estimation on GPU platforms [2, 7, 18, 22]. Hong in [7] proposed an analytical model for GPU architecture to predict execution performance from CUDA codes. The model approximates the execution of GPU kernels as computation phases of equal length with memory accesses in between. The concept of memory warp parallelism (MWP) and computation warp parallelism (CWP) works well in evaluating the workload bottleneck and modelling the effect of latency hiding. The computational part of the kernel is estimated by a simple mapping from PTX code. The shared memory accesses are assumed to have no bank conflicts and as fast as accessing register file. This model is created for a early GPU architectures for which the access latencies of memory instructions do not vary. State-of-art GPUs are usually endowed with multiple level of caches, which introduces randomness in access latencies. Each memory access may go to different hierarchy of memory, resulting in different access latencies. Thus, such model is not applicable to state-of-art GPU architectures. GPU Cache behaviour can be anaylsed and modelled based on reuse distance theory [10, 13, 19] to predict cache misses and thus performance. Another work [18] builds on a simplified model of [7], and model cache behaviour from the memory

request queue maintained at every memory hierarchy level. Since the predictions are from CUDA code where the thread-level parallelism has been exposed, the memory trace generated are highly accurate compared to the actual GPU memory access trace. Furthermore, the usage of memory request queue limits the portability since such real-time information is not made known in other architectures. In comparison, CGPredicts works with sequential C code and the cache behaviour is modelled accurately with sequential traces and cache configurations of the hardware.

Cross-platform performance prediction has been explored in several works [1, 12]. GROPHECY [12], based on [7] in GPU modeling from CUDA, proposed a GPU performance projection framework from skeleton CPU code for various optimizations including staging, folding, shared memory and loop unrolling. However, the generation of code skeletons requires manual development of a parallel version, which, in turn demands good understanding to implement CUDA equivalent of a given piece of CPU code. XAPP [1] proposed a machine-learning (ML) based framework to predict GPU performance from single-threaded CPU implementation. The framework formulates program properties as variables and GPU hardware characteristics as coefficients into an established ML technique. However, machine learning approaches cannot provide much insights about the application characteristics. As an anlytical approach, CGPredict not only can accurately predict the performance, but also provide performance bottlenecks of the application which can suggest further hardware specific optimizations.

6 CONCLUSION

With the emergence of heterogeneous system-on-chip platforms, developers are now able to achieve better performance by porting part of the execution onto accelerators. In order to facilitate this process, we present CGPredict, a C-to-GPU performance estimation framework based on an analytical approach to aid the application developers in making early design decisions regarding the choice of accelerators, which will save tremendous time and effort spent to redevelop the application into platform-specific programming languages. CGPreict can estimate in seconds to minutes the performance of applications on GPU platforms starting with single-threaded C code. Experimental results show that CGPredict can accurately estimate GPU performance with an average 9% estimation error across a range of kernels. In addition, CGPredicts performs detailed memory access pattern and cache behaviour analysis which provides developers with insights for further optimizations. Furthermore, CGPredict in conjunction with an existing FPGA estimator is able to guide application developers in choosing the right accelerator platforms (GPU or FPGA).

ACKNOWLEDGMENTS

This work was partially funded by the Singapore Ministry of Education Academic Research Fund Tier 2 MOE2015-T2-2-088.

REFERENCES

[1] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. 2015. Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '15)*. IEEE, 725–737. https://doi.org/10.1145/2830772.2830780

[2] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. 2010. An Adaptive Performance Modeling Tool for GPU Architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. ACM, New York, NY, USA, 105–114. https://doi.org/10.1145/1693453.1693470

[3] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. 2010. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction (CC'10/ETAPS'10)*. Springer-Verlag, Berlin, Heidelberg, 244–263. https://doi.org/10.1007/978-3-642-11970-5_14

[4] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*. ACM, New York, NY, USA, 33–36. https://doi.org/10.1145/1950413.1950423

[5] Jan Edler. 1998. Dinero IV trace-driven uniprocessor cache simulator. *urlhttp://www. cs. wisc. edu/~ markhill/DineroIV/* (1998).

[6] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar '12)*. IEEE, 1–10. https://doi.org/10.1109/InPar.2012.6339595

[7] Sunpyo Hong and Hyesoon Kim. 2009. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 152–163. https://doi.org/10.1145/1555754.1555775

[8] Khronos. 2017. OpenCL: The open standard for parallel programming of heterogeneous systems. (2017). https://www.khronos.org/opencl/.

[9] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[10] Yun Liang and Tulika Mitra. 2010. Instruction Cache Locking Using Temporal Reuse Profile. In *Proceedings of the 47th Design Automation Conference (DAC '10)*. ACM, New York, NY, USA, 344–349. https://doi.org/10.1145/1837274.1837362

[11] Xinxin Mei and Xiaowen Chu. 2017. Dissecting GPU Memory Hierarchy through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (Jan 2017), 72–86. https://doi.org/10.1109/TPDS.2016.2549523

[12] Jiayuan Meng, Vitali A. Morozov, Kalyan Kumaran, Venkatram Vishwanath, and Thomas D. Uram. 2011. GROPHECY: GPU Performance Projection from CPU Code Skeletons. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. ACM, New York, NY, USA, Article 14, 11 pages. https://doi.org/10.1145/2063384.2063402

[13] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri Bal. 2014. A detailed GPU cache model based on reuse distance theory. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA '14)*. IEEE, 37–48. https://doi.org/10.1109/HPCA.2014.6835955

[14] Nvidia. 2017. CUDA Toolkit Documentation. (2017). http://docs.nvidia.com/cuda/index.html.

[15] Nvidia. 2017. NVIDIA. CUDA C Programming Guide v8.0 2017. (2017). https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[16] Nvidia. 2017. Parallel Thread Execution ISA Version 5.0. (2017). http://docs.nvidia.com/cuda/parallel-thread-execution

[17] Nvidia. 2017. Tuning CUDA Applications for Kepler. (2017). http://docs.nvidia.com/cuda/kepler-tuning-guide/.

[18] Arun Kumar Parakh, M Balakrishnan, and Kolin Paul. 2012. Performance Estimation of GPUs with Cache. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*. 2384–2393. https://doi.org/10.1109/IPDPSW.2012.328

[19] Tao Tang, Xuejun Yang, and Yisong Lin. 2011. Cache Miss Analysis for GPU Programs Based on Stack Distance Profile. In *2011 31st International Conference on Distributed Computing Systems*. IEEE, 623–634. https://doi.org/10.1109/ICDCS.2011.16

[20] NVIDIA Tegra. 2014. K1: A New Era in Mobile Computing. *Nvidia, Corp., White Paper* (2014).

[21] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU Microarchitecture through Microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS '10)*. IEEE, 235–246. https://doi.org/10.1109/ISPASS.2010.5452013

[22] Gene Wu, Joseph L Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. 2015. GPGPU performance and power estimation using machine learning. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA '15)*. IEEE, 564–576. https://doi.org/10.1109/HPCA.2015.7056063

[23] Xilinx. 2017. Vivado design suite. (2017). https://www.xilinx.com/products/design-tools/vivado.html

[24] Xilinx. 2017. XILINX inc. (2017). http://www.xilinx.com.

[25] Guanwen Zhong, Alok Prakash, Yun Liang, Tulika Mitra, and Smail Niar. 2016. Lin-analyzer: A High-level Performance Analysis Tool for FPGA-based Accelerators. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, New York, NY, USA, Article 136, 6 pages. https://doi.org/10.1145/2897937.2898040

[26] Guanwen Zhong, Alok Prakash, Siqi Wang, Yun Liang, Tulika Mitra, and Smail Niar. 2017. Design Space exploration of FPGA-based accelerators with multi-level parallelism. In *Design, Automation Test in Europe Conference Exhibition, 2017 (DATE '17)*. IEEE, 1141–1146. https://doi.org/10.23919/DATE.2017.7927161

Received April 2017; revised June 2017; accepted July 2017