CUDAAdvisor: LLVM-Based Runtime Profiling for Modern GPUs

Du Shen
College of William and Mary, USA
dshen@cs.wm.edu

Shuaiwen Leon Song[*]
Pacific Northwest National Laboratory, USA
shuaiwen.song@pnnl.gov

Ang Li
Pacific Northwest National Laboratory, USA
ang.li@pnnl.gov

Xu Liu
College of William and Mary, USA
xl10@cs.wm.edu

Abstract

General-purpose GPUs have been widely utilized to accelerate parallel applications. Given a relatively complex programming model and fast architecture evolution, producing efficient GPU code is nontrivial. A variety of simulation and profiling tools have been developed to aid GPU application optimization and architecture design. However, existing tools are either limited by insufficient insights or lacking in support across different GPU architectures, runtime and driver versions. This paper presents *CUDAAdvisor*, a profiling framework to guide code optimization in modern NVIDIA GPUs. CUDAAdvisor performs various fine-grained analyses based on the profiling results from GPU kernels, such as memory-level analysis (e.g., reuse distance and memory divergence), control flow analysis (e.g., branch divergence) and code-/data-centric debugging. Unlike prior tools, CUDAAdvisor supports GPU profiling across different CUDA versions and architectures, including CUDA 8.0 and Pascal architecture. We demonstrate several case studies that derive significant insights to guide GPU code optimization for performance improvement.

CCS Concepts • **General and reference** → **Measurement**; **Performance**; **Metrics**;

Keywords GPU, LLVM, Profiling, Optimization

ACM Reference Format:
Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. 2018. CUDAAdvisor: LLVM-Based Runtime Profiling for Modern GPUs . In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3168831

1 Introduction

General-purpose GPUs have been widely adopted in various computing domains, such as accelerating scientific computing applications, deep learning and graph workloads. From hardware perspective, 71 supercomputers in the top 500 list employ CPU+GPU heterogeneous architectures as of June 2017 [2]; among them, Piz Daint and Titan, both of which employ CPU-GPU architectures, rank third and fourth on the list, respectively. Moreover, due to the advancement of deep learning, NVIDIA released DGX-1 [19], a deep learning system consisting of eight Pascal GPUs. From software perspective, large packages, such as LAMMPS [41], TensorFlow [21] and Galois [40], have been leveraging modern GPUs to achieve superior performance.

Unlike CPUs, GPUs typically offer a relatively more complex programming and architectural scenario. For instance, they employ thousands of threads, which are divided into warps. With the *Single-Instruction Multiple-Threads* (SIMT) programming model, all the threads in one warp share the same program counter. Moreover, a warp is able to coalesce multiple memory requests to adjacent memory words into one single request, so threads can benefit from spatial locality. Caches on GPUs are often very limited in capacity and they are shared across threads. Typically, programmers have to offload kernels to GPU (e.g., NVIDIA GPUs) to benefit from its high parallelism.

Efficiently designing a GPU kernel is difficult, especially when using low-level programming models, such as CUDA [38] and OpenCL [48]. Thus, it is not uncommon to come across performance bottlenecks that prevent the code from achieving high performance on GPUs. There are multiple unique types of challenges for GPU performance optimization. First, as GPU uses the SIMT programming model, control flow divergence may hurt parallelism. Since threads in different branch paths have the same program counter, they need to serialize between each other. Second, due to memory divergence caused by irregular or strided memory access patterns, GPU performance can be significantly degraded. Third, given the limited cache size on GPU, a large number of threads can easily compete for cache resources

[*] Also with College of William and Mary, USA.

CGO'18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5617-6/18/02…$15.00
https://doi.org/10.1145/3168831

without efficient cache management strategies. Finally, optimizations at the level of intra- and inter-CTA [30] cannot be easily conducted without some clear guidance, especially when dynamic parallelism [9] is involved.

Manually analyzing these performance bottlenecks is tedious, error-prone and sometimes impossible for large code bases. Thus, one usually uses performance profilers to guide code optimization. For example, many CPU profilers have been proposed, such as Intel VTune [22], Oracle Solaris Studio [39], HPCToolkit [3] and gprof [50]. However, these CPU performance tools cannot directly profile GPU kernels. Existing GPU profilers, such as NVProf [1], TAU [34], and G-HPCToolkit [7], perform coarse-grained analysis for GPU kernels with relatively fixed metrics. These tools leverage CUPTI [11] interface available in NVIDIA GPUs to obtain the callbacks upon kernel launches and returns. They enable hardware performance counters on GPUs at the kernel launch point, record the occurrence of performance events during the kernel execution, and associate the events with the kernel after the kernel execution. These performance events include cache misses, memory divergence, and branch divergence. However, these coarse-grained analyses associate performance metrics with GPU kernels, lacking insights into kernel's instructions, loops, or functions. Recent NVIDIA Maxwell and its later GPU generations support PC sampling [37], which samples instructions in a round-robin fashion and provides various stall reasons. However, PC sampling only provides sparse instruction-level insights.

To perform fine-grained analysis inside a GPU kernel, one needs to rely on GPU simulators (e.g. GPGPU-Sim [5]) or emulators (e.g., Ocelot [14]). However, simulation and emulation usually incur high overhead and are complex to develop. Moreover, they may suffer from compatibility issues related to the latest application and runtime features. Since they do not simulate or emulate every feature of the state-of-the-art GPU architectures, the optimization guidance generated may not apply to contemporary GPU hardware. Therefore, there is a high demand for fine-grained profilers that monitor kernel execution on real GPU architectures.

To support fine-grained profiling, NVIDIA released a research prototype named SASSI [47] —a tool that instruments GPU codes to support fine-grained analysis. However, as a close-source tool, SASSI has several limitations in practice, in terms of portability (i.e., not portable across CUDA runtime and architectures), expansibility (i.e., instrumentation engine is not open sourced), complexity (i.e., implementation level can be too low for common developers) and coverage (i.e., overlooks the interaction between CPU host and GPU). To address all these challenges, we present CUDAAdvisor, a fine-grained profiling framework that works on real GPU architectures across different CUDA versions. CUDAAdvisor leverages LLVM infrastructure to instrument a CUDA program for both its CPU and GPU code. Moreover, CUDAAdvisor collects performance data during CUDA program execution, associates the performance data with CPU and GPU behaviors and interactions, and derives useful metrics to guide various optimization techniques.

Contributions. In summary, we make the following contributions in CUDAAdvisor:

- CUDAAdvisor is the first fine-grained GPU profiler that works across generations of modern NVIDIA GPU architectures and CUDA versions, to the best of our knowledge.
- CUDAAdvisor combines the code- and data-centric profiling results from both CPU and GPU and associates performance bottlenecks with their root causes.
- We also demonstrate CUDAAdvisor is able to combine different analyses and derive useful metrics and insights to guide optimizations (e.g., cache bypassing).

We evaluate CUDAAdvisor on two GPU platforms to demonstrate portability: NVIDIA Tesla K40c (Kepler architecture) with CUDA runtime 7.0 and NVIDIA Tesla P100 (Pascal architecture) with CUDA 8.0. By applying CUDAAdvisor to a number of commonly-used GPU applications, we show that CUDAAdvisor can successfully associate performance bottlenecks with program source code and understand their provenience. To showcase the optimization scenarios, we also perform software-level horizontal cache bypassing under the guidance of CUDAAdvisor, which yields speedup as high as 2×.

2 Existing GPU Profilers and Limitations

In this section, we elaborate on the most related work—SASSI, the state-of-the-art fine-grained profiling framework—and distinguish our approach. SASSI is a research prototype from NVIDIA research group, which is implemented as a pass in NVIDIA's backend compiler ptxas. SASSI selectively inserts instrumentation code to monitor the execution of CUDA kernels. SASSI can instrument instructions and functions. Moreover, SASSI is able to read the values residing in memory location and registers. As demonstrated in the paper [47], SASSI supports to build effective fine-grained profilers. However, SASSI has several limitations in its portability, expansibility, complexity, and coverage.

- *Portability.* As SASSI is based on the CUDA backend compiler `ptxas`, it requires substantial efforts from NVIDIA to support SASSI with the rapid evolution of CUDA runtime and GPU architectures. SASSI currently does not work for CUDA runtime 8.0. In contrast, CUDAAdvisor is based on LLVM for code instrumentation, which can be generally applied to all modern NVIDIA GPUs and CUDA versions.
- *Expansibility.* As SASSI's instrumentation engine is close source, tool developers cannot add any new capability in SASSI. In contrast, CUDAAdvisor's instrumentation engine is based on LLVM, which is open source. Tool developers are able to extend CUDAAdvisor.

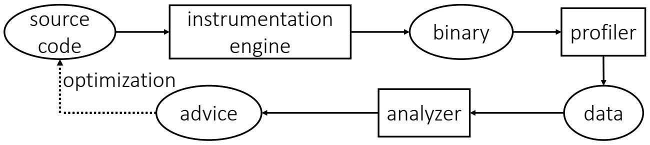


Figure 1. Workflow of CUDAAdvisor.

- *Complexity.* SASSI performs instrumentation during the translation from PTX code to a lower internal code representation for CUDA. Tool developers need to gain a deep knowledge of PTX to develop efficient profilers. To increase productivity, CUDAAdvisor instruments at bitcode level, which hides all the details in CUDA.
- *Coverage.* SASSI instruments GPU kernels only, which overlooks the interaction between CPU and GPU. Instead, CUDAAdvisor instruments both CPU and GPU code to analyze its interactions.

3 CUDAAdvisor Methodology

In this section, we introduce CUDAAdvisor's implementation methodology. Figure 1 shows the workflow of CUDAAdvisor, which consists of three components: instrumentation engine, profiler, and analyzer. CUDAAdvisor's instrumentation engine accepts the source code of CUDA program. It performs code transformation and leverages CUDA compiler to produce the binary code. CUDAAdvisor's profiler then collects the data that represent the behavior of the binary code during its execution on a real GPU hardware. Finally, CUDAAdvisor's analyzer analyzes the profiling data and generates optimization advice with source code attribution. Programmers can follow the advice to optimize the source code and begin another round of analysis if necessary.

In this section, we elaborate on the design and implementation of each CUDAAdvisor component. In the end, we point out some limitations of CUDAAdvisor. CUDAAdvisor is available at https://github.com/sderek/CUDAAdvisor.git.

3.1 CUDAAdvisor Instrumentation Engine

The task of the instrumentation engine is to add necessary instrumentation to CUDA code. We build CUDAAdvisor's instrumentation engine on top of LLVM framework [27] for three reasons. First, LLVM is a general compiler infrastructure that works on both CPU and GPU codes. Second, LLVM works across different GPU architectures and CUDA versions. Third, LLVM is robust, even for complex HPC programs. Figure 2 shows the positions of the instrumentation engine in the whole compilation workflow. As shown in the figure, LLVM frontend—Clang can translate the source code on both host (CPU) and device (GPU) into bitcode, the LLVM's intermediate representation. Clang also supports CUDA compiling (gpucc) [51]. Then, the instrumentation engine, implemented as an LLVM pass, works on both host and device bitcodes. After instrumentation in the device bitcode, LLVM uses specific backend [18] to translate the instrumented bitcode into the CUDA PTX intermediate code. The native CUDA assembler assembles the PTX code into a fat binary (i.e., a file with .fatbin extension), which is then inserted to the host-side CPU bitcode as a string literal. Finally, LLVM compiles the host bitcode into a binary executable.

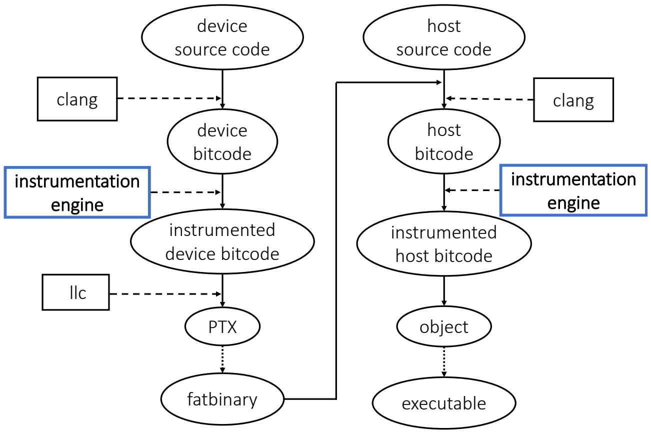


Figure 2. The workflow of the engine inserting instrumentation.

The engine inserts mandatory instrumentation and provides an interface to add optional instrumentation. We describe these two kinds of instrumentations as follows:

(I) Mandatory instrumentation CUDAAdvisor's engine inserts mandatory instrumentations since CUDAAdvisor always reconstructs the call path and data flow in the profiling component, which will be shown in Section 3.2. To collect necessary performance data for the profiling, CUDAAdvisor's engine mandatorily instruments calls and returns for CPU functions, as well as for GPU kernels. Moreover, it instruments functions that allocate memory in CPU code (e.g., malloc, calloc, realloc), in GPU code (e.g., cudaMalloc), and CPU-GPU data transfer functions (e.g., cudaMemcpy). At each instrumentation site, the engine inserts a function and passes the information as arguments of each function. For the memory allocations, the arguments include the starting addresses and the number of bytes allocated on CPU or GPU; for data transfers, the arguments include the starting addresses of memory ranges on both CPU and GPU, as well as the amount of bytes for the transfer.

(II) Optional instrumentation CUDAAdvisor's engine provides the capability of inserting optional instrumentations to support different analyses. Currently, the engine supports optional instrumentation in three categories.

- Memory operations. The engine can instrument every memory read and write and obtain the effective memory address accessed by this operation and its access width.
- Arithmetic operations. The engine can instrument every arithmetic computation and obtain the operator and the (symbolic) values of the operands.

- Control flow operations. The engine can instrument every control flow instruction, such as conditional or unconditional branches, and record their targets.

Common to all these mandatory and optional instrumentation functions, the engine is able to obtain the source code correlation of each instrumented operation, including the file name, line number and column number if available in the debugging information. This information is also passed to the instrumentation function as arguments.

In Section 3.2 and 4.2, we introduce the details of the profiling methods that are based on mandatory and optional instrumentation, respectively.

3.2 CUDAAdvisor Profiler

CUDAAdvisor's profiler divides its task into two stages: (1) the data collection during the CUDA kernel execution, and (2) the data attribution at the end of each CUDA kernel instance. Combining these two stages, CUDAAdvisor performs both code- and data-centric analyses on the fly, which provide the basic infrastructure for code optimization guidance.

3.2.1 Code-centric Profiling

CUDAAdvisor maintains a shadow stack to mirror the execution stack of each thread when the kernel runs on GPU. The profiler pushes the call site onto the shadow stack in the instrumented function at every call instruction, and pops the call site from the shadow stack in the instrumented function at every return instruction. By immediately querying the shadow stack, CUDAAdvisor is able to obtain the call path for each monitored instruction. For efficient analysis, CUDAAdvisor encodes each function with a unique ID number and a call stack is represented as an array of IDs. All GPU threads share the same encoding map from the number to function name and source code; the map resides in GPU global memory. To scale the analysis, each GPU thread maintains its own shadow stack; the shadow stacks are also in GPU's global memory. Upon kernel return, CUDAAdvisor copies all the data from GPU to CPU for further analysis.

On the CPU side, CUDAAdvisor maintains similar shadow stacks for CPU threads, which are used to determine the call stack for the invocation of each CUDA kernel. CUDAAdvisor concatenates this CPU call path with the ones collected inside the GPU kernel instance to give a complete path from the `main` function to each monitored CUDA instruction. We call this capability of CUDAAdvisor as code-centric profiling.

3.2.2 Data-centric Profiling

CUDAAdvisor's data-centric profiling reconstructs the data flow from CPU to GPU to help understand the access patterns of a data object across CPU and GPU, or even across different GPU kernels. Figure 3 highlights the behavior of CUDAAdvisor's profiler for data-centric analysis. The data flow of one data object starts at the beginning of its lifetime and ends in the memory accesses in GPU kernels that reference this data object. Typically, a data object is allocated on the CPU side dynamically or statically. The profiler interprets the malloc family functions (which are already instrumented by the engine) for dynamic allocation and reads the symbol table for static allocation. The profiler maintains a map that records the allocation call path for dynamic data objects and names for static data objects, and their allocated memory ranges. Similarly, CUDAAdvisor's profiler captures the data object allocation on the GPU side and keeps these data objects in another map. To correlate the two maps, CUDAAdvisor overloads the `memcpy` family functions and captures the two memory ranges involved in the memory copy.

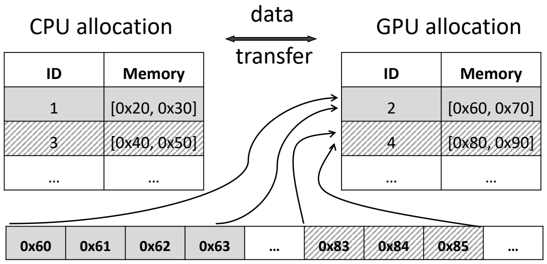


Figure 3. CUDAAdvisor's data-centric profiling.

With the two maps ready, CUDAAdvisor's profiler associates the memory accesses in GPU kernels with the data allocation. As the memory instrumentation obtains the effective address accessed by each memory read or write, the profiler utilizes this effective address to associate the memory access with data object. Since all data allocations and transfers are invoked on the CPU side, the profiler performs data-centric attribution at the end of each kernel instance after the memory access traces are copied back to the host CPU. After the data-centric profiling, CUDAAdvisor is able to construct a complete data flow from every data allocation to its accesses, which helps observe interesting memory behaviors, such as the change of access patterns of the same data object across different GPU kernels.

3.2.3 Profiling Outputs

To minimize the overhead, CUDAAdvisor's profiler does not perform any analysis other than the code- and data-centric attribution. At the end of each kernel instance, the profiler copies all the performance data from GPU to CPU for further analysis. In Section 4, we elaborate on the implementations with case studies.

3.3 CUDAAdvisor Analyzer

CUDAAdvisor's analyzer has an online component that is invoked at the end of each kernel instance. It performs the analysis on the data collected by the profiler. The analyzer can be customized for different purposes. In Section 4, we show how we devise the analysis for reuse distance, memory

Table 1. GPU architectures for evaluation.

Architecture	GPU	CC.	CUDA	Driver	Host CPU
Kepler	Tesla K40c	3.5	7.0	361.93	Intel Xeon E5-2650
Pascal	Tesla P100	6.0	8.0	375.20	Intel Xeon E5-2698

divergence and branch divergence. Moreover, CUDAAdvisor's analyzer has an offline component that merges the analysis results of kernel instances in the same call path. It provides an aggregate statistical view, such as mean, min, max, and standard deviation across all these instances. Such statistical analysis demonstrates the performance variation across different instances of the same GPU kernel and provides intuitive guidance for performance optimization.

3.4 Limitations of CUDAAdvisor

CUDAAdvisor has a few limitations. First, similar to other runtime profilers such as SASSI, it incurs relatively high overhead (but much lower than simulators), which is caused by heavyweight fine-grained instrumentation to CPU and GPU instructions. This high overhead may disturb a program execution. However, the goal of CUDAAdvisor is not to capture the abnormal behavior of hardware-software interactions, but to formulate software metrics to identify software inefficiencies. Thus, CUDAAdvisor's overhead does not affect the accuracy of the framework. Moreover, it can capture detailed execution behavior, while no existing coarse-grained tools such as NVProf [1] and HPCToolkit [3] can. Second, CUDAAdvisor is based on LLVM, which requires the availability and recompilation of the source code of a monitored program. In the HPC community, the source code is often available and the recompilation time is much less than the execution time. Thus, the benefit of code optimization surpasses the extra work of recompilation. Third, the performance analysis is based on the code generation of LLVM, not other GPU compilers, such as nvcc. We will show that CUDAAdvisor's optimization guidance also works for CUDA codes with other compilers. Finally, since CUDAAdvisor is implemented at the bitcode level, it cannot profile register-related stats since NVIDIA has not released its assembly layer to the public.

4 Evaluation

In this section, we present use cases of CUDAAdvisor on detailed memory and control flow analysis, using a group of GPU applications. We also show two examples where CUDAAdvisor provides guidance for code optimization.

4.1 Evaluation Methodology

Evaluation Environment. We evaluate CUDAAdvisor on two different architectures, which are summarized in Table 1. Both host architectures are Intel Xeon CPUs with gcc 4.8.4 and installed with LLVM 5.0. For GPUs, we select an NVIDIA Kepler architecture because of its mainstream adaptation. Kepler's L1 cache shares the same on-chip storage with the shared memory (i.e., scratchpad memory) on each SM. Their sizes are configurable, 16/48 KB, 32/32 KB or 48/16 KB for Kepler. For demonstrating portability, we also evaluate CUDAAdvisor on the most recent NVIDIA Pascal architecture with CUDA 8.0. Pascal completely uses the entire on-chip storage for shared memory but it has a 24KB read-only L1/Texture unified cache. Note that although only Kepler and Pascal are tested in this paper, CUDAAdvisor can be generally applied to all the other modern NVIDIA GPUs such as Fermi and Maxwell.

Benchmarks. Shown in Table 2, we showcase CUDAAdvisor using ten representative GPU applications, from Rodinia [8] and Polybench [17]. Both are among the most commonly-used open source CUDA benchmarks. They contain a wide range of applications that fall into various research categories. The selected applications in Table 2 are also used in the previous studies [4, 10, 16, 23–26, 28, 30–33, 42–45, 53–55].

4.2 Case Studies

In this subsection, we present the use cases of CUDAAdvisor on detailed memory system analysis and control flow analysis, both of which are critical performance bottlenecking factors for modern GPUs. To further showcase the application of our tool, we demonstrate how these insights extracted from CUDAAdvisor can be used to aid program debugging, steer performance optimizations and facilitate compiler/architecture research. Additionally, we provide instrumentation scenarios for these case studies to assist users.

(A) GPU Reuse Distance

Significance. Recent research [4, 10, 16, 23–26, 28, 30–33, 42–45, 54, 55] has focused on on-chip cache locality optimization for overall performance and energy consumption since modern GPUs heavily rely on their on-chip memories (e.g., L1 and texture caches) to reduce the off-chip memory wall effects. One of the most widely applied approaches to analyze cache behavior is *reuse distance analysis* [15, 36]. At the surface level, reuse distance can generally reflect how good cache locality an application has under certain input. Combined with other detailed information such as memory divergence degree and MSHR (*miss-status holding-registers*) status, it can be used to help architects predict optimal cache design such as size and associativity. However, conducting this analysis can be quite complex due to GPU's parallel execution model and fine-grained massive multi-threading. Conventional approaches resort to either a GPU simulator such as GPGPU-Sim, which often uses trimmed down input sizes due to time-consuming execution, or a GPU emulator such as Ocelot [14] which only provides unordered lists of memory accesses rather than ordered traces that can be obtained from simulators (e.g., [36] has to enable its own assumed warp and CTA scheduler on top of Ocelot to produce usable traces). On the contrary, our CUDAAdvisor enables much faster and convenient runtime profiling of reuse distance across generations of NVIDIA GPU architectures.

Table 2. Benchmarks for showcasing CUDAAdvisor.

Application	Description	warps/CTA	Input dataset	Source
backprop	Back Propagation	8	65536	Rodinia[8]
bfs	Breadth First Search	16	graph1MW_6.txt	Rodinia[8]
hotspot	Temperature Simulation	8	temp_512 power_512	Rodinia[8]
lavaMD	Molecular Dynamics	4	-boxes1d 10	Rodinia[8]
nn	Nearest Neighbor	8	filelist_4 -r 5 -lat 30 -lng 90	Rodinia[8]
nw	Needleman-Wunsch	1	2048-10	Rodinia[8]
srad_v2	Speckle Reducing Anisotropic Diffusion	8	2048-2048-0-127-0-127-0.5-2	Rodinia[8]
bicg	BiCGStab Linear Solver	8	1024*1024	Polybench[17]
syrk	Symmetric Rank-K Operations	8	default	Polybench[17]
syr2k	Symmetric Rank-2K Operations	8	default	Polybench[17]

```
1 virtual bool runOnBasicBlock(Function::iterator &B) {
2   for(BasicBlock::iterator BI = B->begin(), BE = B->end
        (); BI != BE; ++BI) {
3     if ( auto *op = dyn_cast<LoadInst>(&(*BI))) {
4       /* get loc info */
5       const DebugLoc &loc = BI->getDebugLoc();
6       int line =  loc.getLine();
7       int col =  loc.getCol();
8
9       /* get effective address */
10      Value* addr = op->getPointerOperand();
11      Value* ptr = builder.CreatePointerCast(addr, Type
             ::getInt8PtrTy(C) );
12
13      /* get number of bits */
14      Type* tp = CI->getType();
15      int sizebits=(int)tp->getPrimitiveSizeInBits();
16
17      /* insert a function call */
18      IRBuilder<> builder(op);
19      builder.CreateCall(hook, {ptr, builder.getInt32(
             sizebits), builder.getInt32(line), builder.
             getInt32(col), 1});
20 }}}
```

Listing 1. The LLVM pass to instrument memory accesses.

CUDAAdvisor Instrumentation. We instruct CUDAAdvisor to instrument at all the global memory operations. At each instrumentation site, CUDAAdvisor collects some specific information and passes them as arguments to the *analysis function*, such as effective address, number of bits accessed, source code location, etc. As discussed previously, CUDAAdvisor leverages a LLVM pass to conduct instrumentation at the bitcode level. Shown in Listing 1, LLVM parses the bitcode (Line 1 and 3) and instruments every global load instruction (Line 5). It then extracts the source code line and column from the debugging information (Line 7-9), obtains the effective address (Line 11-12) and the number of bits (Line 14-15). Finally, the pass creates a function call to a predefined analysis function and passes all these arguments (Line 18). Note that global stores and shared/constant/texture/read-only accesses can be profiled in a similar fashion.

Listing 2 shows a snippet of instrumented bitcode. Line 1 is the original code, which loads a float from address *%a*. Line 2 and 3 are inserted by CUDAAdvisor. Line 2 converts the pointer from float (float*) into a general pointer (i8*), while Line 3 calls analysis function *Record()*, and passes arguments as the effective address, number of bits accessed, the source code line and column, and operation type. *Record()* packs all the arguments along with CTA ID and thread ID into one entry. Entries from all memory accesses form a trace. CUDAAdvisor stores this trace in a buffer located in GPU's global memory. This analysis function is a __device__ function so that it is callable by device. It is written in a separate CUDA source file and compiled into a separate bitcode file before being merged with the kernel bitcode by llvm-link. For data marshaling, CUDAAdvisor initiates data transfer using cudaMemcpy and copies collected trace from device to host (e.g., can be accomplished through Unified Memory supported by CUDA 6 and beyond). To calculate reuse distance for each CTA, the trace is first regrouped into multiple traces based on their associated CTA IDs, which is then used by CUDAAdvisor. CUDAAdvisor offers two reuse distance model: memory element based and cache line based. In addition to reuse, CUDAAdvisor also records the number of streaming accesses (i.e., accesses to memory elements that are never reused by the same CTA).

```
1 %3 = load float, float* %a, align 4,  !dbg !1127
2 %4 = bitcast float* %a to i8*, !dbg !1127
3 call void @Record(i8* %4, i32 32, i32 20, i32 13, i32 1)
     , !dbg !1127
```

Listing 2. Memory access instrumented bitcode.

Results and Analysis. Throughout the paper, we refer to reuse distance as the traditional definition of data reuse distance. From the view of memory and cache, a sequential execution of a program is a sequence of data access [15]. Given such a sequence, we define reuse distance as the number of distinctive data elements accessed between two consecutive uses of the same element. Under such definition, reuse distance directly reflects data temporal reuse. For instance, *ABCCDEFAAAB* is a data access sequence. The reuse distance of *B* is 5. For better facilitating the later discussion on GPU L1 cache-level optimization, we slightly tweak its definition: once an address *A* is written, we will restart its reuse distance counting as another address '*A'*' because GPU L1 cache follows write-no-allocate write-evict policy [32, 55]. Since reuse distance is an inherent program property and independent of cache parameters or underlying machines, we perform reuse distance analysis with CUDAAdvisor on Kepler architecture only. We evaluated ten applications from Table 2. Seven of them are shown in Figure 4. BFS and NN are excluded because they exhibit very low reuse (more than 99% of the accesses), while Syr2k is excluded since it resembles Syrk. The x-axis represents the range of reuse distance.

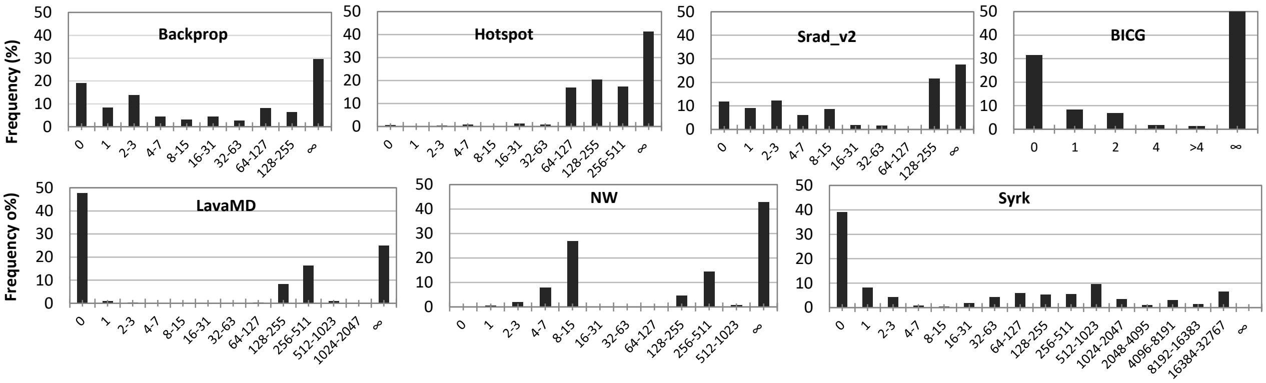


Figure 4. Reuse distance analysis through CUDAAdvisor. ∞ is defined as data is never reused again during the program execution or before the next write to the address (e.g., write-evict L1 on NVIDIA GPUs).

∞ represents no-reuse, indicating that data is never reused in the program or before the next write to it. We have the following interesting observations: (1) BFS and NN exhibit very low reuse, both suffering from branch-heavy codes with little loads during execution. For example, *Kernel* and *Kernel*2 with few loads to *some_array* in BFS, and *euclid* with few loads to *d_locations* in NN. (2) Eight out of ten applications suffer from high no-reuse accesses (except for Syrk and Syr2k), which wastes the precious resources such as cache and MSHRs. Among them, Hotspot exhibits both long reuse distance and very high no-reuse, making it insensitive to L1 cache level optimizations, such as capacity increment and bypassing. The other seven cases incur accesses with both decent frequency of short reuse distance and high no-reuse, e.g., BICG and LavaMD. They present some level of sensitivity to L1 level optimization schemes. (3) Syrk and Syr2k both exhibit low no-reuse and higher frequency of short reuse distance (e.g., reuse distance 0's frequency is close to 40%). However, they also exhibit accesses with very long reuse distance (e.g., around 25% beyond 512), indicating that cache capacity likely affects the effectiveness of L1 level optimization schemes. In summary, CUDAAdvisor's reuse distance analysis paints a general picture of an application's cache locality and provides important insights for selecting potential optimizations for different software/architectures. To better steer optimizations, reuse distance analysis can be combined with other information, such as memory divergence degree, register pressure and shared memory usage.

(B) Memory Divergence Frequency and Degree

Significance. GPU memory divergence can significantly bottleneck performance, thus becomes a popular research topic in recent years [33, 35, 44, 49, 52]. It is also an important indicator on whether a program is well optimized for memory access. Because of GPU's SIMT execution model, warp instructions execute in a lock-step. All the memory requests for a given instruction must be received before this warp can proceed. At architecture level, a coalescing unit has been added into the data path before reaching to L1 cache as a "best effort" approach for combining accesses to the same cache line into a single request for reducing off-chip memory access. However, memory divergence still occurs quite often in irregular kernels that touch many unique cache lines, causing performance degradation. Effectively profiling memory divergence frequency and degree for applications is essential for optimizations. Unlike production tools (e.g., NVprof [1]) that provides coarse-grained estimation and is limited to early generations of hardware, CUDAAdvisor provides programmers with fine-grained profiling for memory divergence at instruction level. It also flexibly reports summarized results, such as memory divergence degree.

CUDAAdvisor Instrumentation. CUDAAdvisor relies on memory traces to analyze GPU kernel's memory divergence characteristics. We instruct CUDAAdvisor to instrument at all the global memory *read* and *write* operations. Similar to reuse analysis, CUDAAdvisor collects effective address, number of bits accessed and source code location at instrumentation site, and then passes them to the analysis function. CUDAAdvisor leverages the same LLVM pass of Reuse Distance in Listing 1 for memory divergence profiling. Runtime traces are stored on GPU's global memory and copied to host for data marshaling. Analysis metrics such as memory divergence degree is modularized in post-analysis.

Results and Analysis. Figure 5 shows the profiled memory divergence distribution for an entire application on Kepler (128 Byte cache line) and Pascal (32 Byte cache line) architectures in Table 1, which is calculated as the number of unique cache lines touched by each instruction. The selected applications are from Table 2 and the maximum range of x-axis is 32 for NVIDIA GPUs due to 32 threads/warp. Note that since the distribution figures for three applications including BICG, Syrk and Syr2k have mostly 1 and 32 cache lines touched, we do not show them in the figures to save space but report their numbers here: for Kepler architecture, BICG (1 ⇒ 75%, 32 ⇒ 25%), Syrk (1 ⇒ 50.02%, 32 ⇒ 49.98%) and Syr2k (1 ⇒ 50%, 32 ⇒ 50%); for Pascal, BICG (1 ⇒ 50%, 4 ⇒ 25%, 32 ⇒ 25%), Syrk (1 ⇒ 49.98%, 32 ⇒ 49.98%) and Syr2k (1 ⇒ 49.99%, 32 ⇒ 49.99%). Such distribution of unique cache lines touched reflects the general optimization

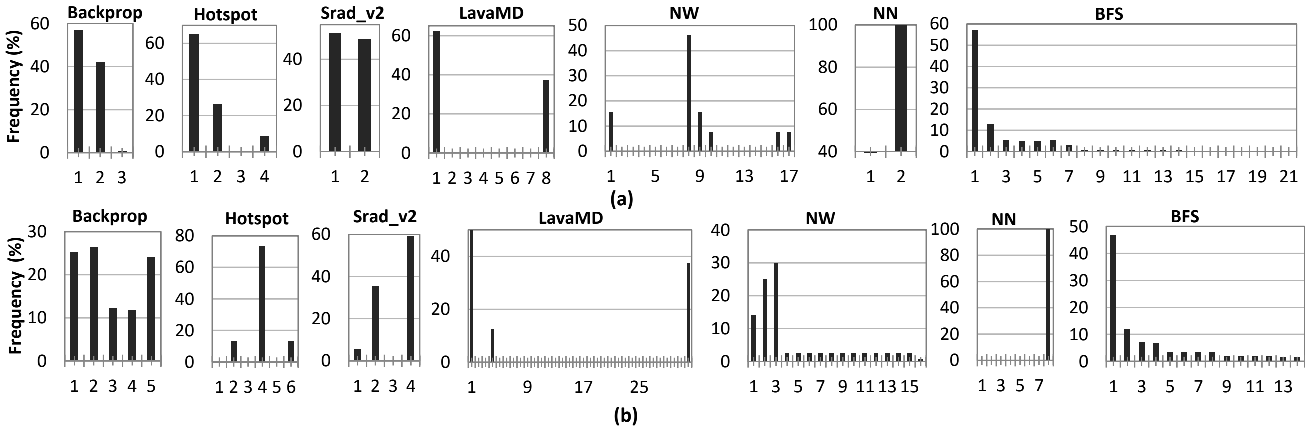


Figure 5. Profiled memory divergence distribution of unique touched cache lines by instructions of an entire application on Kepler. X-axis represents number of unique cache lines touched (min is one and max is 32). (a) Kepler architecture with 128 Byte cache line; (b) NVIDIA Tesla P100 (Pascal) with 32 Byte cache line.

```
1  virtual bool runOnBasicBlock(Function::iterator &B) {
2    /* construct an argument for basic block's name */
3    std::string bb_name = B->getName().str();
4    Value* str_bbname = builder.CreateGlobalStringPtr(
           bb_name);
5    Value* ptr_bbname = builder.CreatePointerCast(
           str_bbname, Type::getInt8PtrTy(C) );
6
7    /* fetch debug information */
8    const DebugLoc &loc = inst->getDebugLoc();
9    int ln = loc.getLine();
10   int cl = loc.getCol();
11
12   /* create a function call to analysis function */
13   builder.CreateCall(hookBB, {ptr_bbname, builder.
           getInt32(ln), builder.getInt32(cl) } );
14 }
```

Listing 3. Implementation of LLVM pass to instrument basic blocks.

degree on memory access for an application, or how well the memory access pattern is structured. For example, we can observe in Figure 5 that `Backprop, Hotspot and Srad_v2` have better code optimization for avoiding memory divergence than the others in the group. Also, architecture plays a role in deciding the distribution: the largest number of unique cache line touched in Pascal is generally larger than that on Kepler primarily due to cache line size. CUDAAdvisor also provides user interfaces for profiling application's memory divergence degree, which is computed using the average of weighted sum of distribution for the number of unique cache lines touched. Memory divergence degree is also an important index for modeling GPU performance.

(C) Branch Divergence

Significance. In addition to the memory-level analysis capability discussed in (A) and (B), CUDAAdvisor also provides profiling for control flow analysis. Modern GPU programming models enables flexibility for programmers to add control flow in their codes. But conditional control flow can significantly affect warp efficiency and overall performance since it could cause threads in a warp to execute different instructions, which is called *branch divergence*. Branch divergence can be very harmful to GPU performance because a subset of threads in a warp will be deferred to a divergent stack until all the other threads finish executing their path. CUDAAdvisor can provide insights of the kernel's divergence, such as how many times a branch is executed, how many threads execute this branch and how often a certain branch causes a warp to diverge.

```
1 /* string of basic block id */
2 @5 = private unnamed_addr constant [6 x i8] c"entry\\00"
3
4 /* one basic block in a certain function */
5 entry:
6   call void @passBasicBlock(i8* getelementptr inbounds
         ([6 x i8], [6 x i8]* @5, i32 0, i32 0), i32 15,
         i32 36), !dbg !620
7   /* here starts the original instructions */
8   %0 = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x(), !
         dbg !625
```

Listing 4. Basic block instrumented bitcode.

CUDAAdvisor Instrumentation. We instruct CUDAAdvisor to instrument at all basic block entries. At each instrumentation site, CUDAAdvisor collects the names of these basic blocks and extracts their source location from debug information before passing them to the *analysis function*. Similar to memory access instrumentation, CUDAAdvisor relies on an LLVM pass to instrument at bitcode level. As shown in Listing 3, the pass retrieves the name of each basic block and creates a pointer to the string where the name is stored (Line 4-6). It then fetches the source code locations from debug information (Line 9-11). Finally, it creates a function call to the analysis function and pass these arguments (Line 14). Listing 4 shows a snippet of instrumented bitcode. Line 5 and 8 are the original bitcodes and they reside in a certain function. Line 5 indicates a basic block named "entry" and Line 8 is the first instruction in this basic block. Line 2 and 6 are inserted by CUDAAdvisor. Line 2 is a global string which stores the basic block's name. Line 6 is a function call to the predefined analysis function *passBasicBlock*() which

Table 3. Results of Branch Divergence on Pascal.

Application	# divergent blocks	# total blocks	% divergence
backprop	26257	95011	27.64%
bfs	408420	1292788	31.59%
hotspot	1372	4197	32.69%
lavaMD	103	744	13.84%
nn	81	2001	4.05%
nw	147875	212992	69.43%
srad_v2	92643	270128	34.30%
bicg	0	1256	0.00%
syrk	0	817	0.00%
syr2k	15	393	3.82%

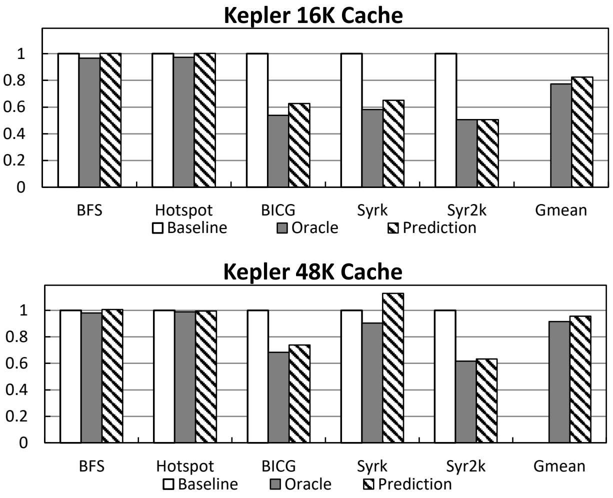


Figure 6. Normalized execution time of different applications on NVIDIA Kepler architecture when using the predicted optimal number of warps per CTA for bypassing. Baseline case is using all the warps (no bypassing). Oracle exhaustively searches the optimal solution. Prediction represents our model.

takes in the basic block's name and source code location. *passBasicBlock*() records arguments along with CTA ID and thread ID in a buffer. Similar to memory access instrumentation, *passBasicBlock*() is written in a separate CUDA source file and merged into application's kernel at bitcode level. For data marshaling, the trace is transferred to CPU upon kernel exit.

Results and Analysis. Table 3 shows the profiling results on the percentage of divergent blocks in an application running on NVIDIA Pascal architecture. This result summary also applies to other NVIDIA GPUs since branch divergence under CUDA is independent of architectures. We can observe that `NN, BICG, Syrk and Syr2k` have very low frequency of branch divergence while the others (especially `NW`) suffer from high frequency of branch divergence. This analysis effectively helps programmers target applications that are in need of branch divergence optimizations, to which previous optimization techniques [13, 29, 56] can be applied.

(D) Optimization 1: Horizontal Cache Bypassing.

Recently, cache bypassing has become a heavily investigated research topic in GPU computing. This is because

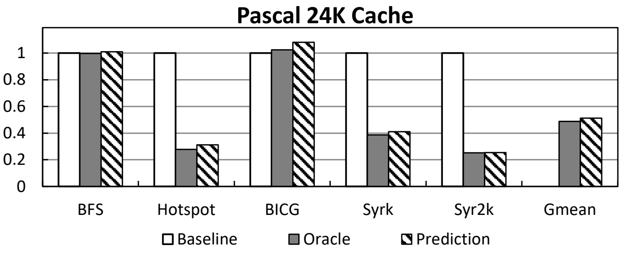


Figure 7. Normalized execution time of different applications on NVIDIA Pascal architecture when using the predicted optimal number of warps per CTA for bypassing. Baseline case is using all the warps (no bypassing). Oracle exhaustively searches the optimal solution. Prediction represents our model.

modern GPUs have very limited L1 data cache and massively threaded GPU applications often exceed the L1 capacity, causing severe thrashing [24, 43]. Additionally, cache-level resources (e.g., MSHR entries and load/store queues) are also very limited, often causing severe resource congestion (e.g., MSHR allocation failures) [30, 32, 33]. To tackle this problem, many architecture solutions are provided, e.g., enabling bypassing threshold in tag store [32] and proposing new bypassing policy [31]. Two general types of software-level bypassing are also proposed: vertical bypassing [55] targeting on every global memory access instruction in PTX (bypassing them for every warp), and horizontal bypassing [31] which focuses on concurrency and only allows certain number of concurrent warps per CTA to access L1 cache. Each of them has certain advantages and disadvantages: vertical bypassing is more fine-grained but requires architectural and runtime information to evaluate every individual load and also bypass all the warps; horizontal bypassing is much simpler and can manage bypassing granularity better but cannot distinguish loads with little reuse. To showcase our tool's prediction capability, we choose the most recent software-level horizontal bypassing proposed in [31] to compare with, which uses a pre-execution sampling period to do exhaustive searching for the optimal number warps per CTA to access L1 on a SM and then follows such suggestion for the remaining execution. Although it can accurately identify the best warp number to access L1, it requires exhaustively searching all the options for number of warps. The actual bypassing in PTX is shown in Listing 5. Using the memory tracing functionality demonstrated in (A) reuse distance and (B) memory divergence from CUDAAdvisor, we can actually model the optimal number warps to access L1 along with

```
//=== compute warp id and set threshold ===
mov.u32 %r0, %tid.x; //Thread index
shr.u32 %r0, %r0, 5; //Warp index
setp.lt.s32 %p0, %r0s, $pi$; //Set Threshold

//=== for each global load ===
@%p0 ld.global.ca.s32 %r9, [%rd6]; //Cache
@!%p0 ld.global.cg.s32 %r9, [%rd6]; //Bypass
```

Listing 5. PTX instrumentation for horizontal bypassing.

other parameters provided by the application and architecture features, without the need of exhaustive searching. We build the optimal warp estimation model in Eq.(1), which can be built into post processing of CUDAAdvisor as an metric. In this model, *R.D.* represents an application's average reuse distance, and *M.D.* is the average memory divergence of the application; both can be calculated through the outputs of CUDAAdvisor. Note that for showcasing purpose we use the average value of *R.D.* and *M.D.* instead of eliminating the outliers (e.g., extreme data points) to rather conservatively estimate the optimal warp number.

$$Opt\_Num\_Warps = \left\lfloor \frac{L1\_Cache\_Size}{R.D. * Cacheline\_Size * M.D. * \#CTAs/SM} \right\rfloor \quad (1)$$

We tested our model on the two GPU architectures shown in Table 1: Kepler with 16KB or 48KB L1 cache and Pascal with 24KB L1/Texture unified cache. We also select cache-bypassing favorable applications (based on the experience from recent simulation work [32]) from Table 2 to demonstrate our model's prediction accuracy for bypassing. Figure 6 and 7 show the prediction evaluation on Kepler and Pascal. The baseline case is the default scenario where cache bypassing is disabled (i.e., using cache), while oracle stands for the exhaustive horizontal bypassing in [31] and prediction represents our model. Both figures clearly show that our model achieves very good performance. It is 6.7% and 4.3% slower than the oracle scenario, for 16KB and 48KB L1 on Kepler, respectively. And for the 24KB unified cache on Pascal, our prediction is 5% slower. We also observe that these supposed bypassing-favorable applications actually are quite different. BFS and Hotspot are quite insensitive applications which match their streaming features discussed in Section 4.2-(A) and Figure 4. The other three who benefit from bypassing actually suffer greatly from capacity misses because increasing cache size from 16KB to 48KB dramatically reduces bypassing benefits (e.g., 23% to 9% for oracle on Kepler). Additionally, architecture features play an important role in bypassing. For instance, bypassing on Pascal performs better than that on Kepler because the unified cache on Pascal locates in texture processor cluster (TPC) instead of SM (i.e., multiple SMs locate on one TPC so the unified cache technically locates between SM and NoC instead of on SM). All these detailed analysis on cache bypassing optimization of real GPU hardware can be easily obtained and modeled through our CUDAAdvisor tool.

(E) Optimization 2: Code- and Data-Centric Debugging.

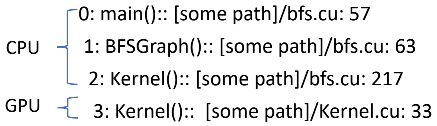
When understanding or debugging a large project, it can be cumbersome and error-prone to read the code and track all data objects. As discussed in Section 3.2, a major contribution of CUDAAdvisor is to provide code-centric and data-centric profiling to show insights of the program and guide debugging. To showcase these features, we take BFS as an example. A code snippet is shown in Listing 6. Since BFS's kernel uses data types of bool and float, a warp can ideally touch only one cache line on Kepler (128 Byte cache line) and up to four cache lines on Pascal (32 Byte cache line), assuming the program has no memory divergence. However, BFS has a portion of memory accesses that touch more than the limits, as previously shown in Figure 5.

```
    // bfs.cu
57  int main( int argc, char** argv) {
      ...
63    BFSGraph( argc, argv);
      ...
    }

75  void BFSGraph( int argc, char** argv) {
      ...
172   cudaMalloc((void**) &d_graph_visited, bytes) ;
173   cudaMemcpy(d_graph_visited, h_graph_visited, bytes,
            cudaMemcpyHostToDevice) ;
      ...
217   Kernel<<<grid,threads,0>>>(..., d_graph_visited, ...);
      ...
    }

    // Kernel.cu
22  __global__ void Kernel(...,bool *g_graph_visited,...) {
      ...
33    if(!g_graph_visited[g_graph_edges[i]])
      ...
    }
```

Listing 6. Code Snippet of BFS.

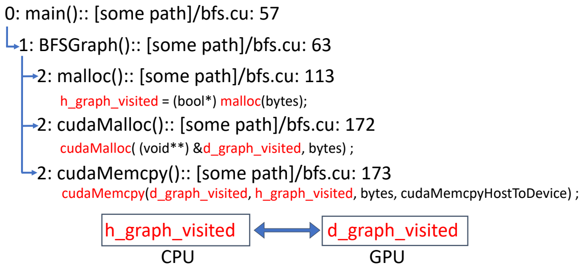
If a programmer is interested to know which memory accesses suffer from memory divergence, CUDAAdvisor can show not only the source code location, but also the calling context. Figure 8 is an illustrative example. Each rows lists the index, the function name, the source file and line number. This example shows that Line 33 of Kernel.cu has significant memory divergence. As can be seen in the figure, CUDAAdvisor concatenates the call path from both host and device to show the calling context starting from main function on host all the way to the suspicious site on device, to better guide programmers to understand the program's behavior. Note that CUDAAdvisor is able to capture and display function calls in CUDA kernel as well.

CUDAAdvisor also detects which data object is associated with memory divergence. An illustrative output is shown Figure 9. CUDAAdvisor shows the calling context to malloc(), cudaMalloc() and cudaMemcpy(). It shows programmer that an array of bool d_graph_visited allocated at Line 172 of *bfs.cu* suffers from memory divergence, and that its counterpart on host is h_graph_visited allocated at Line 113 of *bfs.cu*. These features of CUDAAdvisor can significantly reduce debugging time for a fairly large project.



```
0: main():: [some path]/bfs.cu: 57
1: BFSGraph():: [some path]/bfs.cu: 63
2: Kernel():: [some path]/bfs.cu: 217
3: Kernel():: [some path]/Kernel.cu: 33
```

Figure 8. Code-centric view shows concatenated calling context from both host and device.



```
0: main():: [some path]/bfs.cu: 57
└1: BFSGraph():: [some path]/bfs.cu: 63
   →2: malloc():: [some path]/bfs.cu: 113
       h_graph_visited = (bool*) malloc(bytes);
   →2: cudaMalloc():: [some path]/bfs.cu: 172
       cudaMalloc( (void**) &d_graph_visited, bytes) ;
   →2: cudaMemcpy():: [some path]/bfs.cu: 173
       cudaMemcpy(d_graph_visited, h_graph_visited, bytes, cudaMemcpyHostToDevice) ;
```

Figure 9. Data-centric view shows the interesting data objects, where it is allocated on host and device and where it is transferred.

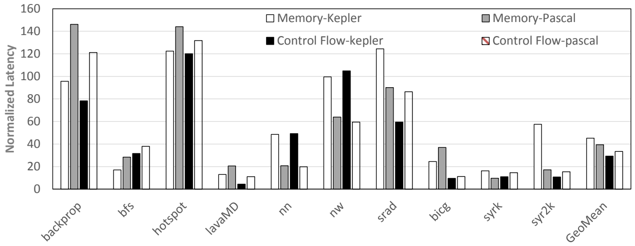


Figure 10. Overhead of memory and control flow instrumentation, on Kepler and Pascal Architectures.

5 Tool's Overhead Analysis

Figure 10 shows the runtime overheads of running GPU kernels for each benchmark listed in Table 2. The baseline used is the original benchmark compiled by Clang 5.0. We perform overhead analysis on Kepler and Pascal architectures, with detailed information in Table 1. The experimental results are averaged on five runs. As shown in Figure 10, the runtime overhead mostly ranges from 10× to 120×. It is much faster than simulators such as GPGPU-Sim that usually incurs 1-10 millions of slow down to the native execution [47].

With further analysis, the overhead mostly comes from three sources. First, CUDAAdvisor utilizes atomic operations to serialize memory access or control flow events. Second, CUDAAdvisor inserts a function call to each instrumentation site. In the future, we will explore a more efficient way to insert instructions rather than heavyweight function calls. Third, the buffer on the device side lives in GPU's global memory. It competes the GPU kernel with shared resources such as cache and MSHR.

6 Related Work

We have distinguished CUDAAdvisor from the most related work—SASSI [47] in Section 2. In this section, we review other profilers that work on GPUs.

NVIDIA provides its own tools to support profiling CUDA code, such as Visual Profiler (NVP) [16], nvprof [1], and NSight [12]. These profilers collect performance data via hardware performance counters and lightweight binary instrumentation. They are able to identify inefficient CPU-GPU interactions and pinpoint performance bottlenecks in CUDA code. However, these production-quality tools are not open sourced and very inflexible for novel research exploration and various analysis since they only provide very limited preselected metrics. For example, unlike CUDAAdvisor, these tools do not provide intuitive optimization guidance with cache bypassing, detailed reuse distance analysis, and memory divergence distribution frequency.

In the HPC community, several profilers that can support coarse-grained GPU-level analysis have been proposed, including Vampire [20], TAU [34], Scalasca [6], G-HPCToolkit [7] and [46]. They collect data via CUPTI tool [11] and hardware performance counters available on GPUs and associate these data with GPU kernels and timestamps. These tools usually incur small overhead and provide insights into problematic CPU-GPU interactions with profiles and traces. However, they fail to identify detailed performance insights into GPU kernels and lack the root cause analysis on performance bottlenecks inside CUDA kernels.

In the architecture community, GPGPU-sim [5] is a widely-used simulator to perform fine-grained simulation tracing. However, compared to runtime profilers, simulation is very slow, preventing it from working on real inputs. Moreover, simulation may not simulate all the features of each hardware component, so the analysis result may not reflect the real execution on newer GPU architectures (i.e., GPGPU-sim only models Fermi architecture).

7 Conclusions

This paper presents CUDAAdvisor, a general framework that supports fine-grained analysis to identify performance bottlenecks of CUDA code. CUDAAdvisor is built atop LLVM to instrument CUDA code running across different GPU architectures and CUDA runtimes. Moreover, CUDAAdvisor supports novel code- and data-centric profiling to provide intuitive guidance for understanding abnormal behaviors in CUDA code. Finally, CUDAAdvisor supports various analysis techniques. We have shown three different analyses for reuse distance, memory and branch divergence. Based on these analyses, CUDAAdvisor is able to further devise new metrics to guide GPU optimization of cache bypassing, which achieves up to 2× speedup. It also provides unique code-centric and data-centric debugging capability.

A Artifact Description

A.1 Abstract

For artifact evaluation, we provide binary of CUDAAdvisor's instrumentation engine and source codes of CUDAAdvisor's analyzer and all benchmarks. To create binaries from the source code, we supply scripts for compiling binaries to test. We also provide scripts used to conduct the experiments and reproduce results presented in the paper.

A.2 Description

A.2.1 Check-List (Artifact Meta Information)

- **Program:** Rodinia and Polybench
- **Compilation:** , gcc-4.8.4, nvcc-7.0.27
- **Binary:** CUDA executables
- **Data set:** default data set provided by Rodinia
- **Run-time environment:** Ubuntu 12.04 with CUDA and GPU computing SDK
- **Hardware:** GPU with compute capability ≥ 3.5 (NVIDIA K40c GPU recommended)
- **Execution:** See below.
- **Output:** Tool analysis include: reuse distance, memory divergence and branch divergence.
- **Experiment workflow:** See below.
- **Publicly available?:** Yes.

A.2.2 How Delivered

For artifact evaluation purpose, we pre-installed CUDAAdvisor on a GPU server equipped with NVIDIA K40c GPU. CUDAAdvisor is also available on Github at https://github.com/sderek/CUDAAdvisor.git. The Github repository will be maintained and can be used to submit issues.

A.2.3 Hardware Dependencies

The GPU code requires CUDA and a device of compute capability 3.5 and up. Please note that changing the compute capability requires updating some Makefiles and environment configurations. The code was tested for compute capability 3.5 and 6.0.

A.2.4 Software Dependencies

GPU requires the NVIDIA CUDA Development Kit. CUDAAdvisor relies on LLVM. The code was tested for LLVM 4.0 and 5.0

A.3 Installation

CUDAAdvisor has been installed on a remote server for artifact evaluation purpose. Server information and credentials to access the server should be shared privately.

To install CUDAAdvisor on your own server, please ensure your system has all software and hardware dependencies. Details of installation instructions can be found in README.md file. Here are the steps of installation.

Go to /lib/Transforms subdirectory under LLVM's source tree root, and checkout CUDAAdvisor from repository:

```
cd [llvm source]/lib/Transforms/
git clone https://github.com/sderek/CUDAAdvisor.git
```

Then copy the following line into file [llvm source]/lib/Transforms/CMakeLists.txt:

```
add_subdirectory(CUDAAdvisor)
```

Edit the configurations in file [llvm source]/lib/Transforms/CUDAAdvisor/env.mk:

```
SM = #sm_xx
CP = #compute_xx
LLVM = #[LLVM source]
PASS = #[LLVM build]/lib/LLVMCudaAdvisor.so
```

Go to the top level of your LLVM build directory and rebuild, you should have CUDAAdvisor's instrumention engine ready at [llvm build]/lib/LLVMCudaAdvisor.so. You are able use the **opt** tool to access it.

A.4 Experiment Workflow

As described in Section 4.2, reuse distance, memory divergence and branch divergence are shown as case studies. For the convenience of the artifact evaluation, we provide scripts which build and run CUDAAdvisor we have described in the paper and store the results in the output text files. Below are the steps to build, run the experiments, and observe the results.

- **build and run:**

```
cd CUDAAdvisor/expr/
./run.sh
```

- **organize the results:**

```
./showoutput.sh
```

A.5 Evaluation and Expected Result

Observe the results for each case study under three directories: RD_mode (reuse distance), MD_mode (memory divergence) and BD_mode (branch divergence). Code-centric and data-centric debugging results can be observed in output text file of every benchmark.

Acknowledgments

This research is partially supported by National Science Foundation (NSF) under Grant No. 1618620 and 1464157. This research is also partially supported by U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under the "CENATE" project (award No. 66150). The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830.

References

[1] 2017. NVIDIA Visual Profiler. NVIDIA. http://docs.nvidia.com/cuda/profiler-users-guide

[2] Jun. 2017. Top500 supercomputer sites. https://www.top500.org/lists/2017/06. (Jun. 2017).

[3] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22 (2010), 685–701.

[4] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu. 2015. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 25–38. https://doi.org/10.1109/PACT.2015.38

[5] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 163–174. https://doi.org/10.1109/ISPASS.2009.4919648

[6] David Böhme, Markus Geimer, Lukas Arnold, Felix Voigtlaender, and Felix Wolf. 2016. Identifying the Root Causes of Wait States in Large-Scale Parallel Applications. *ACM Trans. Parallel Comput.* 3, 2, Article 11 (July 2016), 24 pages. https://doi.org/10.1145/2934661

[7] Milind Chabbi, Karthik Murthy, Michael Fagan, and John Mellor-Crummey. 2013. Effective Sampling-driven Performance Tools for GPU-accelerated Supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 43, 12 pages. https://doi.org/10.1145/2503210.2503299

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization. IEEE International Symposium on*. IEEE, 44–54.

[9] Guoyang Chen and Xipeng Shen. 2015. Free launch: optimizing GPU dynamic kernel launches through thread reuse. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 407–419.

[10] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. 2014. Adaptive Cache Management for Energy-Efficient GPU Computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 343–355. https://doi.org/10.1109/MICRO.2014.11

[11] NVIDIA Corp. 2011. CUDA Tools SDK CUPTI User's Guide DA-05679-001_v01. https://developer.nvidia.com/nvidia-visual-profiler. (October 2011).

[12] NVIDIA Corp. 2017. NVIDIA Nsight. http://www.nvidia.com/object/nsight.html. (2017).

[13] Zheng Cui, Yun Liang, Kyle Rupnow, and Deming Chen. 2012. An accurate GPU performance model for effective control flow divergence optimization. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 83–94.

[14] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT '10)*. ACM, New York, NY, USA, 353–364. https://doi.org/10.1145/1854273.1854318

[15] Chen Ding and Zhong Yuntao. 2001. Reuse Distance Analysis. In *Computer Science at University of Rochester Tech report UR-CS-TR-741*. U of Rochester.

[16] Jayesh Gaur, Raghuram Srinivasan, Sreenivas Subramoney, and Mainak Chaudhuri. 2013. Efficient Management of Last-level Caches in Graphics Processors for 3D Scene Rendering Workloads. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 395–407. https://doi.org/10.1145/2540708.2540742

[17] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar)*. IEEE.

[18] LLVM Group. 2016. LLVM: User Guide for NVPTX Back-end. http://llvm.org/docs/NVPTXUsage.html. (2016).

[19] NVIDIA Group. 2017. NVIDIA DGX-1 AI Supercomputer. http://www.nvidia.com/object/deep-learning-system.html. (2017).

[20] Daniel Hackenberg, Guido Juckeland, and Holger Brunst. 2012. Performance analysis of multi-level parallelism: inter-node, intra-node and hardware accelerators. *Concurrency and Computation: Practice and Experience* 24, 1 (2012), 62–72. https://doi.org/10.1002/cpe.1725

[21] Google Inc. 2017. TensorFlow: An open-source software library for Machine Intelligence. https://www.tensorflow.org. (2017).

[22] Intel 2017. Intel VTune Amplifier XE 2017. http://software.intel.com/en-us/intel-vtune-amplifier-xe. (April 2017).

[23] Hyeran Jeon, Gunjae Koo, and Murali Annavaram. 2014. CTA-aware Prefetching for GPGPU. *Computer Engineering Technical Report Number CENG-2014-08* (2014).

[24] W. Jia, K. A. Shaw, and M. Martonosi. 2014. MRPB: Memory request prioritization for massively parallel processors. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 272–283. https://doi.org/10.1109/HPCA.2014.6835938

[25] Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. 2013. Orchestrated scheduling and prefetching for GPGPUs. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 332–343.

[26] Onur Kayıran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. 2013. Neither more nor less: optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 157–166.

[27] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

[28] Jaekyu Lee, Nagesh B Lakshminarayana, Hyesoon Kim, and Richard Vuduc. 2010. Many-thread aware prefetching mechanisms for GPGPU applications. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*. IEEE, 213–224.

[29] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. AsanoviÄĞ. 2013. Convergence and scalarization for data-parallel architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–11. https://doi.org/10.1109/CGO.2013.6494995

[30] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, and Henk Corporaal. 2017. Locality-Aware CTA Clustering for Modern GPUs. In *Proceedings of 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XXII)*. ACM, New York,NY, USA. https://doi.org/10.1145/3037697.3037709

[31] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. 2015. Adaptive and transparent cache bypassing for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 17.

[32] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. 2015. Locality-driven dynamic gpu cache bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15)*. ACM, 67–77.

[33] Lingda Li, Ari B Hayes, Shuaiwen Leon Song, and Eddy Z Zhang. 2016. Tag-Split Cache for Efficient GPGPU Cache Utilization. In *Proceedings of the 2016 International Conference on Supercomputing (ICS'17)*. ACM, 43.

[34] Allen D. Malony, Scott Biersdorff, Sameer Shende, Heike Jagode, Stanimire Tomov, Guido Juckeland, Robert Dietrich, Duncan Poole, and Christopher Lamb. 2011. Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs. In *Proceedings of the 2011 International Conference on Parallel Processing (ICPP '11)*. IEEE Computer Society, Washington, DC, USA, 176–185. https://doi.org/10.1109/ICPP.2011.71

[35] Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 235–246. https://doi.org/10.1145/1815961.1815992

[36] C. Nugteren, G. J. van den Braak, H. Corporaal, and H. Bal. 2014. A detailed GPU cache model based on reuse distance theory. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 37–48. https://doi.org/10.1109/HPCA.2014.6835955

[37] NVIDIA. 2015. CUDA 7.5: Pinpoint Performance Problems with Instruction-Level Profiling. https://devblogs.nvidia.com/parallelforall/cuda-7-5-pinpoint-performance-problems-instruction-level-profiling. (2015).

[38] NVIDIA. 2015. CUDA Programming Guide. (2015). http://docs.nvidia.com/cuda/cuda-c-programming-guide

[39] Oracle. 2012. Oracle Solaris Studio. http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html. (2012).

[40] Keshav Pingal. 2014. Galois. http://iss.ices.utexas.edu/?p=projects/galois. (2014).

[41] Steve Plimpton. 1995. Fast Parallel Algorithms for Short-range Molecular Dynamics. *J. Comput. Phys.* 117, 1 (March 1995), 1–19. https://doi.org/10.1006/jcph.1995.1039

[42] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. 2013. A Locality-aware Memory Hierarchy for Energy-efficient GPU Architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 86–98. https://doi.org/10.1145/2540708.2540717

[43] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 72–83.

[44] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2013. Divergence-aware Warp Scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 99–110. https://doi.org/10.1145/2540708.2540718

[45] A. Sethia, D. A. Jamshidi, and S. Mahlke. 2015. Mascar: Speeding up GPU warps by reducing memory pitstops. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 174–185. https://doi.org/10.1109/HPCA.2015.7056031

[46] Shuaiwen Leon Song, Chunyi Su, Barry Rountree, and Kirk W Cameron. 2013. A simplified and accurate model of power-performance efficiency on emergent GPU architectures. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 673–686.

[47] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. 2015. Flexible Software Profiling of GPU Architectures. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 185–197. https://doi.org/10.1145/2749469.2750375

[48] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering* 12, 3 (2010), 66–73.

[49] Jingweijia Tan, Shuaiwen Leon Song, Kaige Yan, Xin Fu, Andres Marquez, and Darren Kerbyson. 2016. Combating the Reliability Challenge of GPU Register File at Low Supply Voltage. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 3–15. https://doi.org/10.1145/2967938.2967951

[50] Dominic A. Varley. 1993. Practical experience of the limitations of Gprof. *Software: Practice and Experience* 23, 4 (1993), 461–463.

[51] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. 2016. GPUCC - An Open-Source GPGPU Compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. New York, NY, 105–116. http://dl.acm.org/citation.cfm?id=2854041

[52] P. Xiang, Y. Yang, and H. Zhou. 2014. Warp-level divergence in GPUs: Characterization, impact, and mitigation. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 284–295. https://doi.org/10.1109/HPCA.2014.6835939

[53] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. 2015. Enabling coordinated register allocation and thread-level parallelism optimization for GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 395–406.

[54] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. 2013. An Efficient Compiler Framework for Cache Bypassing on GPUs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '13)*. IEEE Press, Piscataway, NJ, USA, 516–523. http://dl.acm.org/citation.cfm?id=2561828.2561929

[55] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. 2015. Coordinated static and dynamic cache bypassing for GPUs. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 76–88. https://doi.org/10.1109/HPCA.2015.7056023

[56] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011. On-the-fly Elimination of Dynamic Irregularities for GPU Computing. *SIGPLAN Not.* 46, 3 (March 2011), 369–380. https://doi.org/10.1145/1961296.1950408