Toward an Analytical Performance Model to Select between GPU and CPU Execution

Publisher: IEEE Cite This Cite This PDF

4 Author(s) Artem Chikin ; Jose Nelson Amaral ; Karim Ali ; Ettore Tiotto **All Authors**

107
Full
Text Views

Abstract: Automating the device selection in heterogeneous computing platforms requires the modelling of performance both on CPUs and on accelerators. This work argues for the use ... **View more**

Metadata

Abstract:
Automating the device selection in heterogeneous computing platforms requires the modelling of performance both on CPUs and on accelerators. This work argues for the use of a hybrid analytical performance modelling approach is a practical way to build fast and efficient methods to select an appropriate target for a given computation kernel. The target selection problem has been addressed in the literature, however there has been a strong emphasis on building empirical models with machine learning techniques. We argue that the applicability of such solutions is often limited in production systems. This paper focus on the issue of building a selector to decide if an OpenMP loop nest should be executed in a CPU or in a GPU. To this end, it offers a comprehensive comparison evaluation of the difference in GPU kernel performance on devices of multiple generations of architectures. The idea is to underscore the need for accurate analytical performance models and to provide insights in the evolution of GPU accelerators. This work also highlights a drawback of existing approaches to modelling GPU performance - accurate modelling of memory coalescing characteristics. To that end, we examine a novel application of an inter-thread difference analysis that can further improve analytical models. Finally, this work presents an initial study of an OpenMP runtime framework for target-offloading target selection.

Published in: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

Date of Conference: 20-24 May 2019

INSPEC Accession Number: 18867604

Date Added to IEEE *Xplore*: 29 July 2019

DOI: 10.1109/IPDPSW.2019.00068

ISBN Information:

Publisher: IEEE

Conference Location: Rio de Janeiro, Brazil, Brazil

Contents

SECTION I. Introduction

High-level programming models such as Open Multi-Processing (OpenMP) [1] and OpenACC [2] provide the means to write architecture-agnostic accelerator code. Target-agnostic programming abstracts the details of accelerator architecture from the developer and makes it the prerogative of the compiler/runtime to handle architecture-specific code generation, optimization, and parameter tuning, within the limits allowed by the programming model. Furthermore, programming models like OpenMP are shifting towards being more descriptive, rather than prescriptive, with the next iteration of the standard poised to introduce constructs that allow compilers most freedom yet on how to generate code and where it should execute (e.g. #pragma concurrent). While there is great value on tuning the source code of applications for each specific accelerator architecture, that is not the target of this research. The goal here is for a compiler/runtime system to deliver the best performance in a given architecture from an existing source code that cannot be modified.

For some tasks, a split of the computation between CPU and GPU execution leads to better performance. Valero-Lara and Jansson implemented a mesh refinement over Lattice-Boltzmann simulation algorithm by scheduling algorithm subtasks across both the host CPU and the GPU, showing a significant speedup over the initial GPU-only version [3]. Valero-Lara et al. have also shown that cooperative CPU-GPU computation scheme, which allocates task to the platform they are most suited for, beats a pure GPU implementation of a classical cyclic reduction algorithm seen in fast finite difference Poisson solvers [4].

Analytical performance modelling, a mature field of research, has been the focus of work in tuning software systems and guiding compiler optimizations [5]–[6][7]. Due to the increasing prevalence of heterogeneous compute platforms, architecture-specific performance modelling becomes a progressively important topic due to the role it has to play when deploying target-agnostic applications [8], [9]. The ability to automatically choose the processing unit which will execute a given section of code can result in a critical performance advantage. Existing analytical models strive to capture the complexity of the architectures that they are modelling, and the interplay between the levels of abstraction used to represent said architectures.

A critically important challenge faced by analytical performance predictors for CPU execution is to model the specifics of CPU resource allocation and how it impacts instruction latencies. To improve the accuracy of CPU instruction-mix latency modelling, we propose an elegant solution that leverages LLVM-MCA - a predictor that uses the compiler's built-in instruction scheduling algorithms [10]. The tool is integrated into an existing analytical model in order to increase its accuracy. In the realm of GPU performance models, Hong's performance model is a seminal approach to runtime prediction [11]. One of the model's biggest losses in abstraction is in characterizing the coalescing characteristics of memory accesses - a critical factor for GPU code performance. We introduce an improvement to the model that applies IPDA, a hybrid symbolic analysis framework that captures the precise coalescing characteristics of OpenMP parallel loops set for GPU code-generation, in order to generate better estimates of the GPU's memory-warp parallelism.

This paper makes progress toward addressing the following research problem: *How to construct runtime target device selection heuristics, what are the biggest challenges involved, and how to make such heuristics suitable for production environments?* Modelling execution of compute kernels on a variety of accelerator architectures is a notoriously challenging task, we highlight this by examining cross-generational GPU architectural differences that have a significant impact on the outcome of deciding whether to execute a kernel on an accelerator target or to keep execution on the host. We study the use of analytical performance modelling to address this research question. Machine-Learning-based algorithms may achieve high degrees of accuracy, but may also suffer from some drawbacks that limit their applications. For instance, dependence on runtime parameters to make an informed decision requires that the learned model be evaluated immediately prior to kernel launch. For some ML models this evaluation results in significant overhead. Moreover, a classical problem of learning approaches - their black box nature - is a serious limitation in compiler/runtime systems due to its effects on understandability, reproducibility and susceptibility to non-linear, and sometimes non-contiguous, relations between model parameters and performance. This paper studies a decision framework for profitability analysis of offloading GPU versions of OpenMP parallel loops that indicates that combining static analysis and runtime parameters is a suitable approach to make such decisions.

SECTION II. Heterogeneous Platforms, Programming Models, and the Need for Performance Analysis

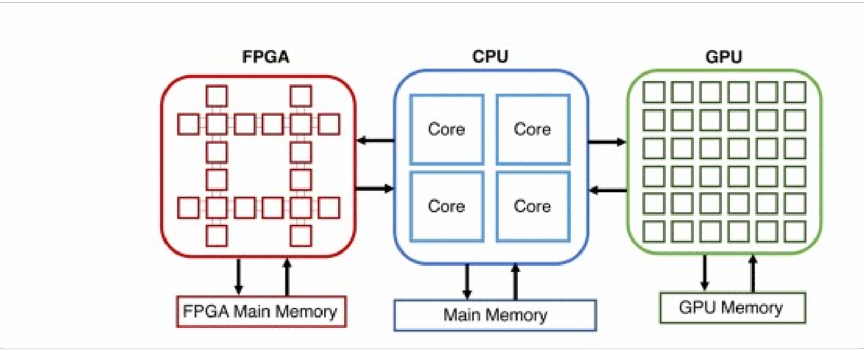


Figure 1:
Example topology of a heterogeneous computing environment

A. Heterogeneous Computing Platforms

A typical heterogeneous system consists of a host machine that operates using an ordinary CPU and contains main memory modules. Attached to the host machine, via a data-transfer bus, are one or more accelerator devices. An example topology of a host computer with two accelerator devices attached is shown in Figure 1. The host - a general-purpose CPU machine - is responsible for the overall system's operation, memory management and control of execution among attached devices. During execution of a program that contains an accelerator kernel - a piece of computation specified to be offloaded to an accelerator - the host machine schedules execution of the kernel on a given computing device and performs the necessary data transfers to and from the computing device. A fallback scenario may occur in which the required accelerator device is unavailable or busy, in which case the host may instead schedule execution of the accelerator kernel on the host machine's CPU. If the programming model allows it, the host may elect to schedule kernel execution either on the host itself or any of the available accelerators.

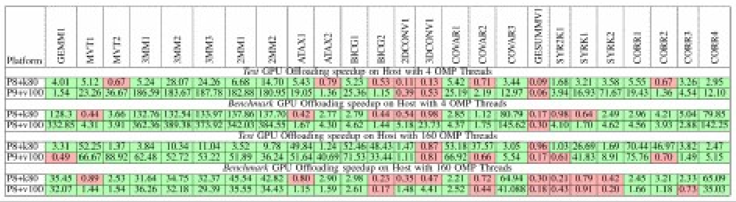
B. OpenMP and Accelerator Programming

OpenMP is a prescriptive directive-based programming model designed for both shared memory multiprocessor programming using C, C++, and FORTRAN. It is made up of a collection of pragma directives for controlling execution of a parallel application, and library routines for interfacing with the runtime environment.

OpenMP 4.0 introduced the capability for the programmer to specify target segments - blocks of code to be executed on an accelerator device. Data is transferred to the device's memory environment for processing. This transfer may occur automatically in case of memory-coherent accelerators or through insertion of explicit data-transferring instructions by a compiler. Code generation for OpenMP constructs enclosed in the target region is specific to the accelerator architecture. For example, consider a case of a simple parallel loop. When generating GPU -executable version of the loop, the compiler is likely to transform it into a data-parallel Single Instruction, Multiple Thread (SIMT) form. If an FPGA device is the compilation target, better performance is achieved through parallelism obtained by pipelining operations of the loop. For a many-core accelerator or a CPU-bound target, a vector SIMD version of the loop would achieve better performance. In an environment that contains multiple heterogeneous computing devices, the compiler may generate multiple versions of the same block of code: each one specialized for a given computing device. OpenMP may not yield hand-tuned CUDA or pthread performance. However, it is popular in scientific computing and thus worth studying. When studying OpenMP, it only makes sense to use a single-source program that targets various architectures with the goal of performance-portability and target agnosticism. Hand-tuning programs for specific architecture contradicts this philosophy.

At execution time, one of the versions of generated kernels is selected for execution on its device. This selection can be based on an environment variable value or on a programmer-specified conditional expression. An important question is whether programming models allow for the compiler/runtime to select a computing device. In the OpenMP 4.x standard the pragma that specifies that a portion of the program is to be offloaded to a device is prescriptive - the computing system has no choice in the matter. However, the upcoming OpenMP 5 programming model will include a loop directive that allows the programmer to specify that the computing system should select the most appropriate computing device for a designated portion of the program.

Table I: Cross-Architectural changes in GPU offloading speedup vs. Host execution



Platform	GEMM1	MVT1	MVT2	3MM1	3MM2	3MM3	2MM1	2MM2	ATAX1	ATAX2	BICG1	BICG2	2DCONV1	3DCONV1	COVAR1	COVAR2	COVAR3	GESUMMV1	SYR2K1	SYRK1	SYRK2	CORR1	CORR2	CORR3	CORR4
Test GPU Offloading speedup on Host with 4 OMP Threads																									
P8+k80	4.01	5.12	0.67	5.24	28.07	24.26	6.68	14.70	5.43	0.79	5.23	0.53	0.11	0.13	5.42	0.71	3.44	0.09	1.68	3.21	3.58	5.55	0.67	3.26	2.95
P9+v100	1.54	23.26	36.67	186.59	183.67	187.78	182.88	180.95	19.05	1.36	25.36	1.15	0.39	0.53	25.19	2.19	12.97	0.06	3.94	16.93	71.67	19.43	1.36	4.54	12.10
Benchmark GPU Offloading speedup on Host with 4 OMP Threads																									
P8+k80	128.3	0.44	3.66	132.76	132.54	133.97	137.86	137.70	0.42	2.77	2.79	0.44	0.54	0.98	2.85	1.12	80.79	0.17	0.98	0.64	2.49	2.96	4.21	5.04	79.85
P9+v100	132.85	4.31	3.91	362.36	389.38	373.92	342.03	384.55	1.67	4.30	4.62	1.44	5.18	23.73	4.37	1.75	145.62	0.30	4.10	1.70	4.62	4.56	3.93	2.88	142.25
Test GPU Offloading speedup on Host with 160 OMP Threads																									
P8+k80	3.31	52.25	1.37	3.84	10.34	11.04	3.52	9.78	49.84	1.24	52.46	48.43	1.47	0.87	53.18	37.57	3.05	0.96	1.03	26.69	1.69	70.44	46.97	3.82	2.47
P9+v100	0.49	66.67	88.92	62.48	52.72	53.22	51.89	36.24	51.64	40.69	71.53	33.44	1.11	0.81	66.92	0.66	5.54	0.17	0.61	41.83	8.91	75.76	0.70	1.49	5.15
Benchmark GPU Offloading speedup on Host with 160 OMP Threads																									
P8+k80	35.45	0.89	2.53	31.64	34.75	32.37	45.54	42.82	0.80	2.90	2.98	0.25	0.35	0.47	2.21	0.72	64.94	0.30	0.21	0.79	0.42	2.45	3.21	2.33	65.09
P8+v100	32.07	1.44	1.54	36.26	32.18	29.39	35.55	34.43	1.15	1.59	2.61	0.17	1.48	4.41	2.52	0.44	41.088	0.18	0.43	0.91	0.20	1.66	1.18	0.73	35.03

C. Symbolic Analysis of Parallel Loop Performance Characteristics

In the context of high-level parallel programming models, the Iteration Point Difference Analysis (IPDA) analysis framework by Chikin et al. is able to statically determine thread memory access stride of addressing expressions contained in OpenMP parallel loops [12]. IPDA builds symbolic difference expressions for inter-thread access strides and solves them in order to determine access characteristics. For instance, the analysis may ascertain that a memory access leads to the generation of coalesced GPU code - adjacent threads access adjacent memory locations - a characteristic paramount to decide whether high-level parallel programs would map to a GPU architecture in an efficient manner. Similarly, this result may also inform the compiler whether the CPU version of the same kernel would exhibit false-sharing among threads. Alternatively, the IPDA analysis may establish that the memory-access patterns of a kernel is favourable for exploitation of the memory/cache hierarchy of a many-core computing device.

SECTION III. Comparative Offloading Performance Change Across GPU Generations

Significant differences among generations of GPU architecture and bus interconnects mean that performance models must be fine-tuned to the most intricate details of the platform they aim to abstract. Table I displays our experimental measurement of GPU offloading benefit for a series of Polybench OpenMP kernels. The data was collected on two experimental platforms: 1) POWER8 Host + Nvidia Tesla K80 (PCI-E) and 2) POWER9 Host + NVidia Tesla V100 (NVlink 2). The k80 and V100 host's CPUs was clocked at 3000Mhz. All programs were compiled using the IBM XL C/C++ compiler ver. 16.1. Each kernel was evaluated in two execution modes, *test* and *benchmark*, which differ only in the size of the program's input, being 1100×1100 and 9600×9600 respectively, in most programs. Each benchmark was executed 10 times and the average execution time of each kernel is used for relative performance measurements. Kernel execution time includes data transfer, but does not include the CUDA context initialization that occurs on the first kernel launch by a given program. The context creation is an overhead paid once by a program that may repeatedly launch many kernels. Omitting context initialization overhead presents a more typical case of executing a kernel of computation and prevents the results from being skewed on single-kernel benchmarks. In our experiments, on Volta architecture, CUDA context initialization can take upwards of 0.5 seconds. The recorded kernel execution time is used to present speedup over the host execution time of the same target region.

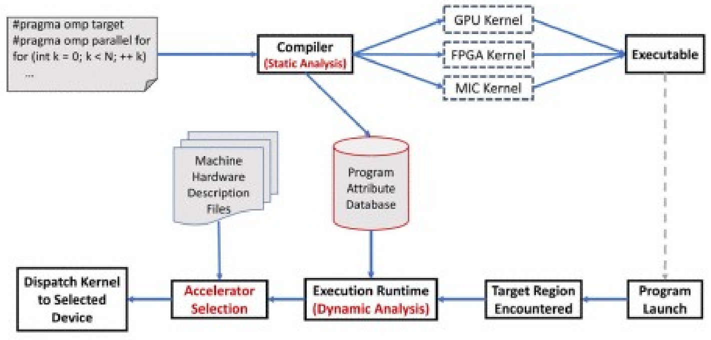
This data shows that a single GPU generation may sway the offloading profitability decision in a drastic fashion. For example, the 3DCONV kernel, in benchmark configuration is a far better fit for execution on the CPU when the accelerator choice is Kepler, with GPU offloading resulting in a slowdown of of $2.1\times$. Yet, a Volta-equipped machine with an even more capable CPU sees a dramatic speedup of $4.41\times$ when offloading the same computation to the GPU. The benchmark's computation kernel has low arithmetic intensity and is heavily memory-bound; thus, benefiting greatly from the Volta's card memory bandwidth of 900GB/s, nearly double of the K80's peak 480GB/s. An example to the contrary is the CORR kernel, which, in benchmark execution mode, is a good candidate for acceleration for a POWER8 host, but should not be offloaded on a POWER9 machine. This outcome holds despite a more capable GPU on a faster interconnect. The four kernels invoked by the benchmark contain sequential loops to be executed by each parallel worker, which are well-suited for SIMD vectorization and stand to benefit from POWER9's broader vector operation support and newly introduced VSX3 operations. In several other cases, despite the decision whether a target region should be offloaded remaining the same, the magnitude of change of speedup is colossal: ATAX2 kernel, in a test run, compared to a 160-thread host, saw an offloading speedup go from $1.24\times$ on K80 to $40.69\times$ on a V100 due to a combination of faster data transfer rates and architectural improvements.

A. Generational Performance Gaps Require Fine-Tuned Performance Estimates

Year-over-year advances in GPU generations are far outpacing development of CPU architecture. This pace of innovation coupled with rise in domain-specific applications particularly well-suited to data-parallel computation mean that rapid evolution of accelerator architectures presents significant challenges for both the compiler developers, and the research community working on analytical performance modelling. Both are chasing a moving target for code optimization and analysis. Meanwhile, CPU platforms too are gaining new features and ever-increasing facilities for vector computation, adapting to the emerging workloads through application-specific gadgets. The increasing importance of performance models means that they need to capture greater amounts of detail intricacies of their target architectures; meanwhile, a growing variability across computing-device architecture types calls for more domain-specific expertise on behalf of those who attempt to model them, attracting more hardware experts to the problem.

SECTION IV.
A New Hybrid Analysis Framework for Deciding the Profitability of GPU Offloading

Designing a compiler/runtime framework for a heterogeneous system that combines multiple processing units is a challenging task. Such framework must amalgamate multiple compilation backends that generate code for several targets and bundle all versions into a single binary, a collection of static analyses that extract relevant program features and characteristics, a means to aggregate relevant dynamic information at a program point prior to the relevant target section, a runtime machine description query mechanism, and detailed performance models that would inform the final offloading selection decision. We describe a prototype of such a framework and outline our approach in detail below, concluding the description by providing some early results of applying the framework on an OpenMP 4 microkernel benchmark suite - Polybench.



#pragma omp target
#pragma omp parallel for
for (int k = 0; k < N; ++ k)
...
Compiler
(Static Analysis)
GPU Kernel
FPGA Kernel
MIC Kernel
Executable
Machine Hardware Description Files
Program Attribute Database
Dispatch Kernel to Selected Device
Accelerator Selection
Execution Runtime (Dynamic Analysis)
Target Region Encountered
Program Launch

Figure 2:
Example compilation and execution flow of an offloading decision compiler/runtime framework.

Figure 2 shows the flow of program compilation and execution. The IBM XL Compiler is used for this prototype. This compiler is a fully compliant implementation of the OpenMP 4.5 standard, capable of outlining target regions specified in the program and translating them into GPU kernels. The outlined region is duplicated prior to code-generation, and a host-bound CPU-parallel version is generated to provide a fallback mechanism in case a GPU device is unavailable. After the creation of optimized versions of the compute kernel for CPU and GPU, the compiler was augmented with a static analysis that collects relevant program features that will form skeletons of respective performance models. The evaluation of these models may depend on values that cannot be known at compilation time and that can be only discovered at runtime; thus, statically constructed performance predictors are inherently incomplete. During program execution, on reaching the target region, the OpenMP runtime is invoked that initializes the accelerator, queues up required data-transfers and launches kernel execution. In our proposed method, the runtime is augmented to instead extract the compiler-collected program features from the program attribute database, and collect runtime values that were missing from the static attributes. A compiler transformation is required that supplies the OpenMP runtime with dynamic information e.g. array sizes, loop trip counts, arbitrary variable values that may be required to determine memory access stride/characteristics. The above data is then used to generate predictions of potential performance gain or loss of offloading the target region to the GPU. Finally, based on the decision, either of the two generated versions of the region code is invoked for execution.

The measurements were performed on an IBM POWER9 (AC922) machine with an Nvidia V100 GPU accelerator connected via the NVlink 2 interface. The machine runs RHEL Server 7.3 Operating System with CUDA version V9.2.88. Due to the requirements placed on the LLVM's instruction scheduler by LLVM-MCA, POWER9 is the only viable host architecture for our experiments at the time of writing. OpenMP loops from the Polybench benchmark suite, representing kernels of the more common high-level computation operations is used to demonstrate the performance model's efficacy and applicability in deciding GPU offloading profitability [13].

A. OpenMP CPU Performance Model

In this work, we leverage a compile-time cost model for OpenMP proposed by Liao and Chapman [14]. The cost model was originally built to augment existing performance estimators of the OpenUH optimizing open-source OpenMP compiler for C/C++ and Fortran programs [15]. OpenUH, in turn, inherits its performance models largely from the Open64 loop nest optimizer infrastructure [16]. Liao's adaptation of the compile-time model implements extensions that account for specifics of OpenMP work-sharing constructs, estimating execution time of a parallel region as determined by the execution time of the most time- consuming thread between each pair of synchronization points. It also adds factors such as scheduling overhead cycles and parallel loop chunk size overheads. An appealing quality of this model is that values of its parameters can be obtained from micro-benchmarks [17]–[18][19]. Figure 4 contains the OpenMP model's equations, directly derived from the equations of the original OpenUH parallel model. Our input kernels consist of strictly parallel loop code and therefore other types of work-sharing constructs described by the model are not exercised. Table II contains various parameters used in the model. Some obtained from the POWER9 Processor User Manual [20]. The TLB miss penalty is estimated using the TLB cost measurement tools included in the Linux libhugetlbfs utility [21]. We used the EPCC OpenMP micro-benchmark suite to measure scheduling and synchronization overhead parameters of the execution model on our hardware configuration [22].

Figure 3: - Equations of cost model for openMP from [14].

Figure 3:
Equations of cost model for openMP from [14].

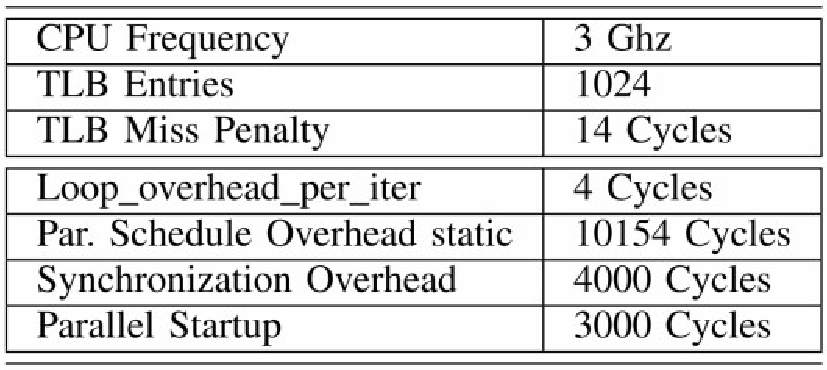
1)

1) Cycles Per Iteration of a Parallel Loop

A key metric in Liao's model is the $Machine_{c}\_per\_iter$ value, computed based on cycles from FP and ALU units, processor memory units, and issue units. Deep ties to the OpenUH compiler's inner instruction scheduler have made it challenging to obtain this estimate in other contexts until recently. We forego the original model's calculations on processor resource, dependency latency and register allocation cycle estimates in favour of the LLVM Machine Code Analyzer (MCA). Spearheaded by SONY, MCA is a performance analysis tool that uses the LLVM infrastructure's rich hardware backend ecosystem to estimate the value of IPC for a given sequence of assembly instructions [23]. Both the compiler-instruction-scheduler-driven analysis and the tool's reporting style were heavily influenced by Intel's IACA tool [24]. The prediction for the number of cycles required to execute an assembly sequence is based on throughput and processor resource consumption as the backend's instruction scheduling model already specifies. The backend module is used to emulate execution of machine code sequence, while collecting a number of statistics which are then presented as a report. The tool is able to handle the presence of long data dependency chains and other bottlenecks. Due to its reliance on the instruction scheduler, it is limited by the quality of the information present in the scheduler. For example, common machine instruction schedulers omit information on the number of retired instructions per cycle or the processor's number of read/write ports in the register file. The tool's known limitations also include a lack of a cache hierarchy and memory type model.

In our experimental implementation, the tool is integrated into the compilation process. The body of a parallel loop is extracted and MCA is used to estimate the total number of cycles required to execute it, yielding the number of cycles spent by a thread participating in the parallel region to do the work of one iteration - $Machine_{c}\_per\_iter$ - in the performance model. The cache hierarchy model, missing from the analysis tool, remains a limitation of the performance model described here and is a primary future work direction to improve the model's accuracy.

Table II: CPU processor/parallel parameters as used in the execution model



CPU Frequency	3 Ghz
TLB Entries	1024
TLB Miss Penalty	14 Cycles
Loop_overhead_per_iter	4 Cycles
Par. Schedule Overhead static	10154 Cycles
Synchronization Overhead	4000 Cycles
Parallel Startup	3000 Cycles

B. GPU Performance Model

An analytical model for a GPU architecture with Memory-level and Thread-level parallelism awareness by Hong and Kim is a seminal approach to performance prediction of GPGPU kernels [11]. Our work implements their model adapted to the Volta architecture by combining static-analysis-driven feature gathering, dynamic kernel information acquired on encountering a target region, and micro-benchmark acquired hardware parameters for values not directly disclosed by the vendor.

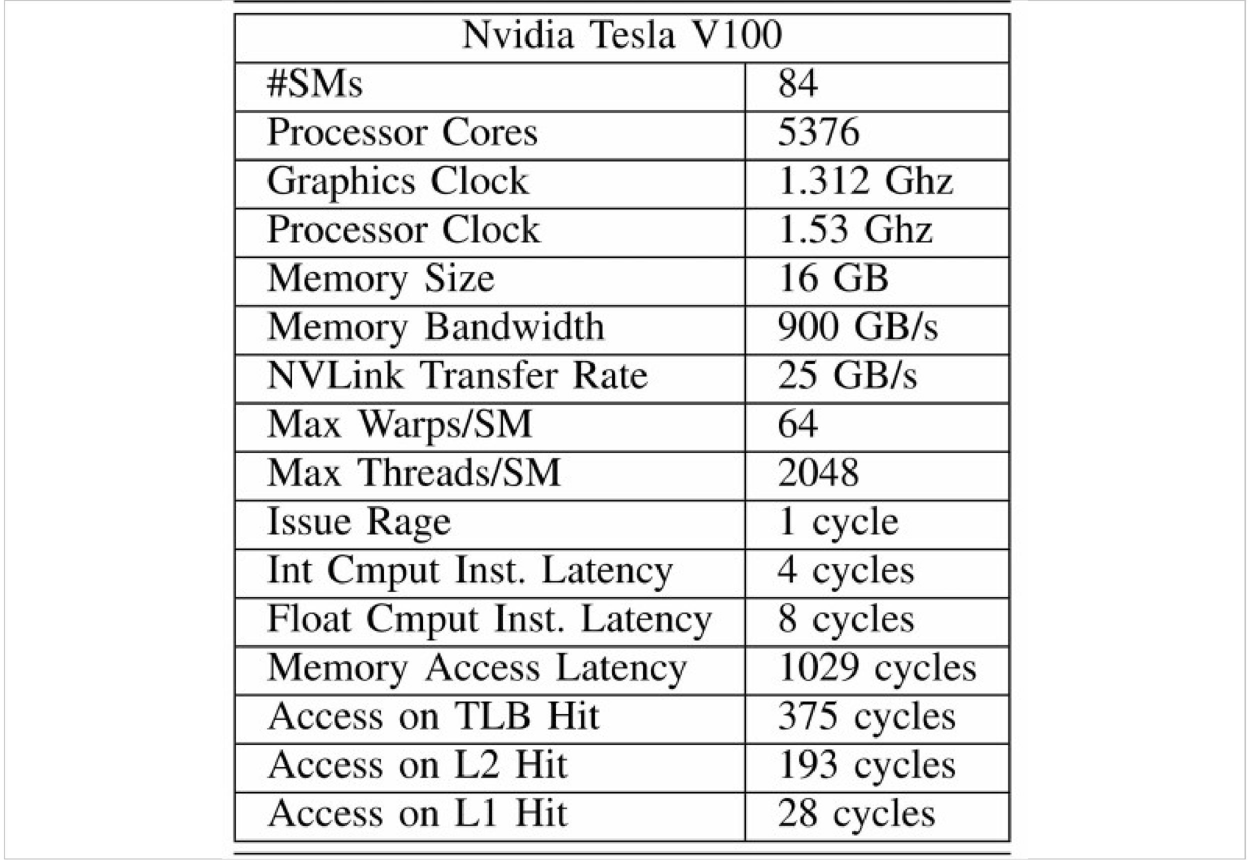
Static Features

The IBM XL compiler generates a GPU kernel version of encountered target regions. Static analyses were integrated into the compilation process that gather program features that are required by the model or are otherwise important indicators of performance.

Instruction Loadout

A key factor in the performance model is the amount of work performed by individual threads. For example, if the amount of computation done by each thread is very small, threads will finish execution very quickly and will have to be queued to be scheduled for more work. In this case, the overhead of scheduling more work to be performed on a GPU for a very short amount of time will be larger than the actual kernel computation, most likely leading to poor performance. The model's thread execution cycle estimate is computed using the number of dynamic instructions. We implement a simple static analysis to count the number of IR instructions, which will be translated into native micro-instructions later. Given the closed nature of the true GPU assembly ISA, this serves as a good estimate. Our static analysis groups collected instructions into I/O and compute categories. Control-flow constructs are abstracted in an identical way across CPU and GPU analyses: all loops are assumed to execute 128 iterations and all conditional blocks of code are assumed to execute half of the time. While the absolute prediction accuracy of this approach might suffer, it should provide a reasonable point of comparison of *relative* performance between the two platforms. Extending this model to include profiling information to improve on these assumptions could result in more accurate modelling at the cost of adding the profiling step to the framework. Profiling is sensitive to the ability of selecting a collection of workloads that can reliably predict the runtime behaviour of future workloads.

Table III: GPU device/bus parameters as used in the execution model



Nvidia Tesla V100	
#SMs	84
Processor Cores	5376
Graphics Clock	1.312 Ghz
Processor Clock	1.53 Ghz
Memory Size	16 GB
Memory Bandwidth	900 GB/s
NVLink Transfer Rate	25 GB/s
Max Warps/SM	64
Max Threads/SM	2048
Issue Rage	1 cycle
Int Cmput Inst. Latency	4 cycles
Float Cmput Inst. Latency	8 cycles
Memory Access Latency	1029 cycles
Access on TLB Hit	375 cycles
Access on L2 Hit	193 cycles
Access on L1 Hit	28 cycles

Architectural Model Parameters

Figure III shows the Volta architecture-specific values used by the model. These values were gathered from either the CUDA API queries, vendor manuals, and the excellent technical report by Zhe Jia who obtained them in a deep examination of the architecture through micro-benchmarking [25].

Runtime Model Parameters

The dynamic aspect of the hybrid approach to performance estimation is essential because only with runtime values the analytical models can be complete. The sizes of kernel inputs prescribe the amount of data that will be sent to the device and back over the interconnect. The size of the iteration space of the original parallel loop affects the number of parallel work-items in the resulting data-parallel program and the grid geometry the runtime will select. In order for the runtime to obtain these values, they are stored into a Program Attribute Database which is queried at execution time, indexed by the target region's program and location.

The original cycle count estimate from the Hong model needed to be modified to adjust for one OpenMP specific aspect of GPU code-generation. The $\#OMP\_Rep$ parameter, highlighted in Figure 4, represents cases where the maximum grid-geometry selected by the runtime does not result in a sufficient number of threads to cover all parallel work items - iterations of the original parallel loop. In that case, a thread performs the work that comprises the body of the original parallel loop, then, depending on the specified schedule of the parallel loop, is assigned another iteration to execute, either by advancing by a static chunk size, or querying the OpenMP runtime. This parameter is set to account for the number of distinct loop iterations a single thread will execute if the number of loop iterations is higher than the product of $num\_thread\_blocks \times threads\_per\_block$. Each iteration of a loop is a work item, or a “repetition.” An OpenMP parallel loop executes $\#OMP\_Rep$. For instance, in a statically scheduled parallel for loop with 1024 iterations executing in a kernel with 1 thread block of 128 threads, each thread executes 8 distinct iterations.

C. GPU Memory Access Pattern

Improved Coalescing Detection
Given the different memory organizations in different accelerators, the memory access pattern is an important input to the performance model. Existing approaches to performance modelling rely on either crude estimates or trace and profile-driven analysis that requires an application to be executed in order to determine its coalescing characteristics [26]. The latter require the code to execute prior to the model being able to generate an accurate prediction. This constitutes a key shortcoming in a production runtime and is a key improvement of this approach in relation to solutions that appear in related work.

The accuracy of model parameters related to memory-throughput can be improved by increasing the accuracy of modelling the memory-coalescing characteristics of code. To this end we build memory-access related parameters of the model using the IPDA analysis framework [12]. Our prototype deploys IPDA to construct a symbolic equation for the inter-thread stride of each memory access. For example, suppose the kernel in question contains the following parallel loop.

IPDA creates a symbolic expression for the inter-thread access stride on the store to array A in line 3:

$$\begin{aligned} & IPD_{t1}(A[\text{max} \;\; * \;\; \text{a}]) - IPD_{t0}(A[\text{max} \;\; * \;\; \text{a}]) \\ & = [\text{max}] \times 1 - [\text{max}] \times 0 \\ & = [\text{max}] \end{aligned}$$

View Source

where a value contained in [] indicates a symbolic unknown. Two possibilities exist in which the framework determines the stride for this memory access:

1. the value of max is known at compile-time and IPDA is able to statically determine, for example, whether or not this kernel would result in coalesced GPU code.

2. the value of max is not known statically, but is known at runtime, prior to kernel launch.

In our proposed compiler/runtime framework, the IPDA symbolic expression of the access stride is stored in a Program

If (MWP == N == CWP):

$$Exec\_Cycles = (Mem\_Cycles + Comp\_Cycles + \frac{Comp\_Cycles}{\#Mem\_Inst} \times (MWP - 1)) \times \#Rep \times \boldsymbol{\#OMP\_Rep}$$

If (MWP > CWP):

$$Exec\_Cycles = (Mem\_Cycles \times \frac{N}{MWP} + \frac{Comp\_Cycles}{\#Mem\_insts} \times (MWP - 1)) \times \#Rep \times \boldsymbol{\#OMP\_Rep}$$

If (CWP > WWP):

$$Exec\_Cycles = (Mem\_L + Comp\_cycles \times N) \times \#Rep \times \boldsymbol{\#OMP\_Rep}$$

Figure 4:
Hong and kim performance model program exection prediction [11]. Highlighted is the additional factor that describes the side-effects of openMP thread-loop-iteration scheduling.

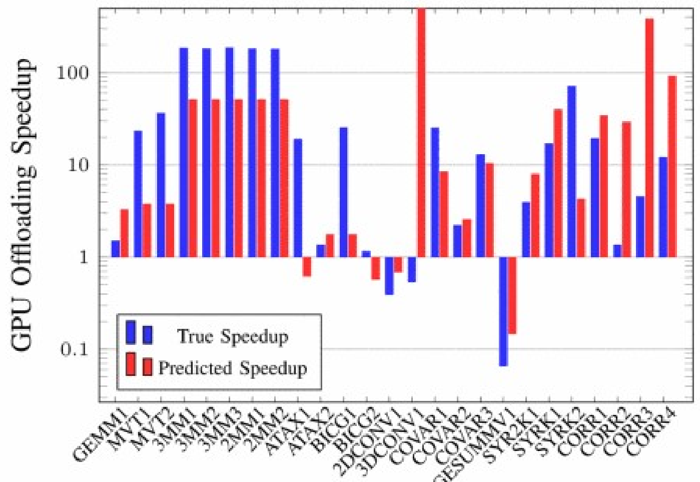
$$
\begin{aligned}
MWP &= min(MWP\_Without\_BW, MWP\_peak\_BW, N) \\
BW\_per\_warp &= \frac{Freq \times Load\_bytes\_per\_warp}{Mem\_L} \\
MW\_peak\_BW &= \frac{Mem\_Bandwidth}{BW\_per\_warp \times \#ActiveSM} \\
\hline
CWP\_full &= \frac{Mem\_Cycles + Comp\_Cycles}{Comp\_Cycles} \\
CWP &= MIN(CWP\_full, N)
\end{aligned}
$$

Figure 5:
Equations of memory-warp and compute-warp parallelism used in GPU execution cost model from [11].

Attribute Database (Figure 2). At the program point when the target region is encountered, the unknown values are extracted and used in the symbolic expression to compute the actual stride, informing the analytical model with whether or not the kernel's accesses are coalesced and to what degree ($\#Uncoal\_Mem\_inst$ and $\#Coal\_Mem\_inst$ values used to compute Mem_Cycles).

D. Putting it all Together

With both the CPU and GPU analytical performance models defined in the OpenMP runtime system, the compiler must alter the code generated for invoking an encountered target region. Instead of simply launching GPU kernel execution, the generated code configures the runtime to extract static features of the generated versions of the region, feeds in the necessary runtime values, and queries the results of performance models. The model that results in the lowest predicted runtime is chosen as the winner and execution is queued up on the architecture the model describes, either the host CPU or GPU. Because of the analytical nature of the model, generating a prediction for either target is equivalent to solving an equation, making decision time negligible in the context of the amount of work already performed by the OpenMP runtime to initiate parallel execution (either on GPU or Host). This goes in a stark contrast to an approach that would employ machine learning to perform model inference at runtime, a step that may, in fact, take longer than the kernel execution itself [27].



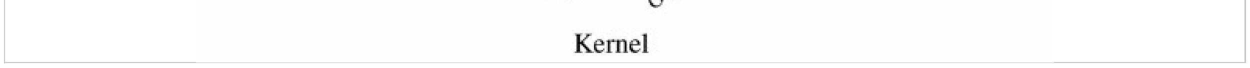


Figure 6:
Actual versus predicted GPU offloading speedup for test kernel execution mode versus a host using 4 threads.

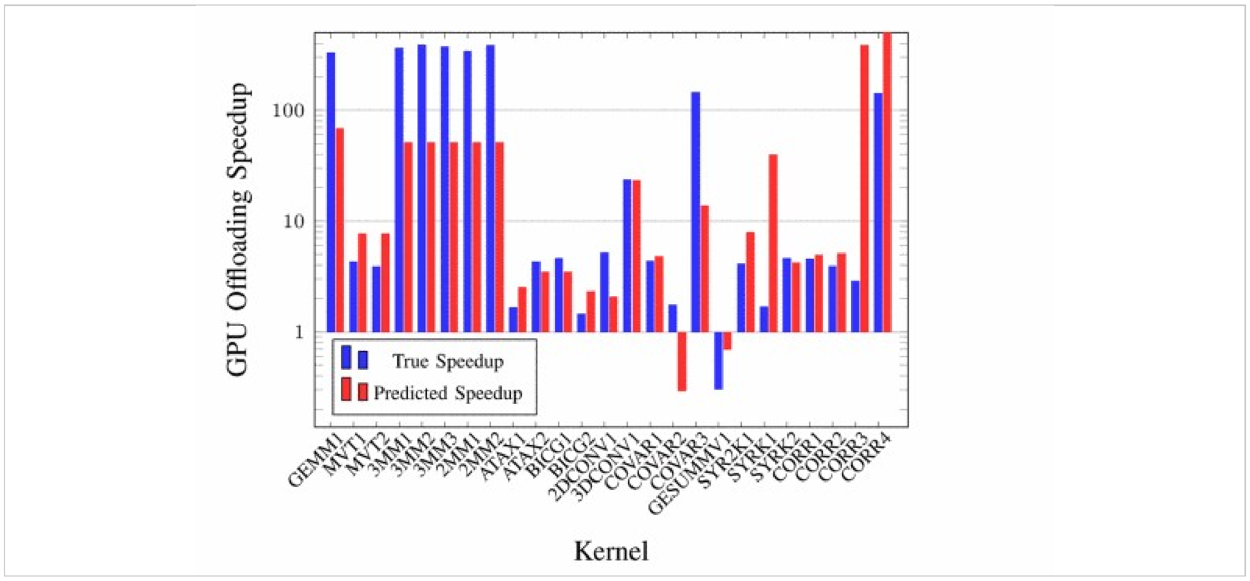


Figure 7:
Actual versus predicted GPU offloading speedup for benchmark kernel execution mode versus a host using 4 threads.

E. Evaluation

This section presents some preliminary results based on an early prototype of the hybrid decision analysis by evaluating it on a collection of parallel OpenMP loops found in the Polybench benchmark suite. These parallel loops represent common atoms of computation found across a variety of applications. 25 kernels from 12 different benchmarks are executed: GEMM, MVT, 3MM, 2MM, ATAX, BICG, 2DCONV, 3DCONV, COVAR, GESUMMV, SYR2K, SYRK, and CORR. Each benchmark in the suite has two modes of execution: test and benchmark. The execution modes differ only in the size of the program's input, being 1100×1100 and 9600×9600 respectively. We also include results for restricting the host execution environment to just 4 threads to demonstrate both the adaptability characteristics of the models, and a scenario that resembles a more typical execution environment, when compared to our experimental machine's 20-core 8-SMT CPU running at full capacity of 160 threads. Execution runtimes were recorded as average kernel runtimes across 10 runs of each benchmark. For a fair comparison of kernel execution times across platforms, the GPU context initialization overhead is omitted in order to demonstrate a generic case of computation offloading in a running application, similarly to the experiment described in Section III.

Figure 6 and Figure 7 demonstrate the true versus predicted speedup of offloading the kernel execution to the GPU in *test* and *benchmark* execution modes. The main sources of prediction error are abstractions in the framework, some of which can be reduced via more detailed models (e.g. a lack of detailed memory hierarchy model), and some stemming from incomplete information available to the predictor (e.g. assuming that all loops inside kernels execute a fixed constant number of iterations and that conditional branches are taken with a 50%

probability). The framework assumes that as long as the same abstractions/heuristics are used across different versions of the kernel, the relative error among versions of the kernel is more important than errors in the prediction of actual execution time.

When deploying the decision analysis framework to select the execution target, overall benchmark suite execution time is improved. When following the compiler's default policy of always offloading target regions to an accelerator, GPU offloading of all kernels yields a geometric mean speedup of $10.2\times$ and $2.9\times$ versus a host using 160 CPU threads (*Test* and *Benchmark* execution modes, respectively). Switching the runtime to evaluate the relative performance of GPU offloading through analytical modelling and only do so when predicted to be profitable profitable results in a geomean speedup of $14.2\times$ and $3.7\times$ on an otherwise identical configuration. Figure 8 shows the speedups achieved under both experimental setups. Note that the speedup provided by the GPU is captured in most cases, with few notable outliers: in the 160-thread *Benchmark* execution model, the model's decision on the convolution kernels is incorrect, predicting a speedup of $0.913\times$, whereas the true offloading speedup is $1.48\times$ in the 2D case. Discrepancies in scenarios where the decision is a close one, such as these, require further tuning of the model to increase its accuracy. Improved representation of the memory hierarchy impacts is a sure way to improve prediction efficacy for these scenarios. The SYRK2 kernel in *Test* execution mode has the performance model severely over-estimate the GPU execution time relative the the CPU running at 160 threads, likely due to over-accounting for the kernel's poor coalescing characteristics without taking the details of cache hierarchy into account.

While the OpenMP specification does not, currently, allow compliant runtime systems to elect to not offload target regions, there is a clear need to provide this ability to runtime vendors. Even among highly-regular OpenMP parallel loops - a construct best-suited for translation into data-parallel code, there are computation patterns ill-suited to GPU acceleration. While more difficult to model, common OpenMP programs that utilize mixtures of construct types to express parallelism alongside sequential code within target regions are even more likely to see better performance on the host fallback path. We demonstrate an early but successful attempt at guiding compile/runtime system architecture to handle a more descriptive programming model approach. The upcoming OpenMP 5.0 standard is set to introduce new constructs that allow implementors exactly this kind of freedom [28].

SECTION V.
Related Work

A. Performance Modelling of Parallel Programs

There exists much prior art in prediction and modelling the performance of parallel programs on a given architecture. Many approaches used to predict whether or not a program will achieve good performance focus on simulation. Aversa et. al. describes a simulation environment for hybrid distributed heterogeneous applications that uses application traces to analyze performance [29]. FASE is another framework for performance prediction of heterogeneous HPC systems that relies on constructing a simulation environment to evaluate architectural options available to heterogeneous system designers [30]. FASE's focus is on overall performance of a hybrid distributed and heterogeneous platform, rather than application-specific or kernel-specific, which is the focus of the hybrid analysis introduced in this paper. Snavely et al. at the San Diego Supercomputing Center (SDSC) proposes mapping a machine signature to an application profile to arrive at a single processor performance prediction [31]. Snavely's work is focused on Simultaneous-Multi-Processing (SMP) platforms and does not apply to accelerator devices or heterogeneous computing platforms. Moreover, the SDSC work is also based on profiles of previous executions of the applications. A large amount of published literature focuses on performance modelling, estimation and task-scheduling for distributed-memory computing environments [32]–[33][34][35]. This work studies the use of hybrid performance modelling in the shared-memory context of a single heterogeneous compute node. Solutions to this problem compliment/augment distributed work-sharing techniques.

A recent addition to the OpenMP standard is OMPT: a Tools API for performance analysis [36]. OMPT affords a running application API-level access to the OpenMP runtime event-handling loop at a fine-grained per-thread level. Ghane et al. combine OMPT with hardware performance tools to build a predictor to identify false sharing behaviour among OpenMP threads [37]. Unlike the approach proposed in this paper, these tools need execution profiles. Profiling could compliment our methodology by feeding the program attribute database with more actionable data over time.

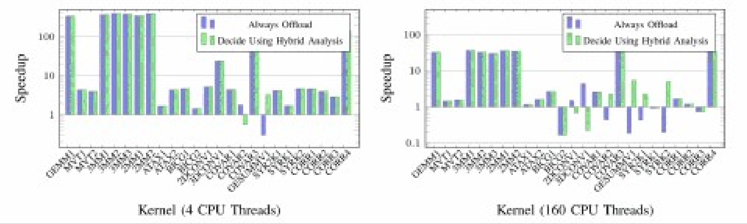


Figure 8:
Speedup achieved by always offloading to GPU versus offloading when determined profitable by the analytical hybrid decision model. Benchmarks executed in benchmark mode.

B. Machine Learning Techniques

Some machine-learning approaches apply multi-layer neural networks trained on execution traces/samples of programs to predict performance [38], [39]. Others apply clustering and correlation analysis to tune parallel application parameter space [40]. Wang et al. use neural networks to map parallelism of CPU-parallel OpenMP programs to NUMA computing clusters [41]. Despite the popularity of ML approaches in research, industrial applications of such techniques for the selection of computing device are hard to find. Industrial-strength compilers must be able to not only produce reproducible performance, but to do so in a transparent manner. Also, the need for runtime-available parameters in ML solutions leads to overhead for inference at runtime. For instance, a simple matrix multiplication kernel makes little sense to accelerate with a GPU when operating on 16×16 matrices, but stands to benefit dramatically when matrices are very large. An informed decision cannot be made without data available only immediately prior to executing the computation which can be offloaded. For instance, Lloyd et al. have successfully implemented a machine-learning predictor for selecting the GPU grid geometry to execute parallel OpenMP loops [27]. The predictor resulted in kernel execution times that are superior to the default selection made by the compiler, yet the time taken to generate the prediction generated an additional overhead that overshadowed all benefits. In fact, the delay imposed by the inference was often longer than the time that it took to execute the computing kernel. ML-based approaches have a role to play and may complement this study, however practical considerations motivate this study of an analytical approach that does not require prior profile runs or traces of the application. Our proposed hybrid analytical model combines static program analysis and dynamic program information to make decisions about the computational device to execute a kernel of computation in an efficient manner with negligible overhead.

SECTION VI. Discussion and Future Work

OpenMP 4.0 standard greatly expands the functionality of the programming model by introducing support for programming heterogeneous computing systems. Newly written applications can take advantage of powerful accelerators like GPUs by annotating the code with appropriate target constructs. Meanwhile, a great wealth of existing OpenMP code can be upgraded by users through fairly minor modifications and additions of new directives to existing constructs. This work is a first step in developing support for automatic offloading decisions for loop nests in OpenMP 4.x applications. Automatic offloading reduces the effort of porting legacy code to state-of-the-art heterogeneous computing platforms to a simple act of

Authors
Figures
References
Keywords
Metrics