AutoMatch: Automated Matching of Compute Kernels to Heterogeneous HPC Architectures

Ahmed E. Helal*, Wu-Chun Feng*†, Changhee Jung†, and Yasser Y. Hanafy*
Department of Electrical and Computer Engineering*,
Department of Computer Science†,
Virginia Tech,
Email: {ammhelal, wfeng, chjung, yhanafy}@vt.edu

***Abstract*—HPC systems contain a wide variety of heterogeneous computing resources, ranging from general-purpose CPUs to specialized accelerators. Porting sequential applications to such systems for achieving high performance requires significant software and hardware expertise as well as extensive *manual* analysis of both the target architectures and applications to decide the best performing architecture and implementation technique for each application. To streamline this tedious process, this paper presents AutoMatch, a tool for automated matching of compute kernels to heterogeneous HPC architectures. AutoMatch analyzes the *sequential* application code and automatically predicts the performance of the best *parallel* implementation of its compute kernels on different hardware architectures. AutoMatch leverages such prediction results to identify the best device for each kernel from a set of devices including multi-core CPUs and many-core GPUs. In addition, it estimates the relative execution cost between the different architectures to drive a workload distribution scheme, which enables end users to efficiently exploit the available compute resources across multiple heterogeneous architectures. We demonstrate the efficacy of AutoMatch, using a set of open-source HPC applications and benchmarks with different parallelism profiles and memory-access patterns. The empirical evaluation shows that AutoMatch is highly accurate across five different heterogeneous architectures, identifying the best architecture for each workload in 96% of the test cases, and its workload distribution scheme has a comparable performance to a profiling-driven oracle.**

***Keywords*-HPC; Automatic Performance Prediction; Workload Distribution; Parallel Architectures; Heterogeneous computing; Performance Modeling; LLVM; CPU; GPU;**

I. Introduction

With the end of Dennard scaling, the performance of sequential CPUs has hit the power wall [1]. To meet the ever-increasing demand for computing performance, driven by the multitude of data sets, computer architectures have shifted to parallel processing. However, unlike the sequential computing era, there is no de facto standard for hardware acceleration. Rather, the parallel architecture landscape is in flux as new platforms are emerging to meet the massive computing needs of new workloads. Therefore, current (and future) HPC systems contain a wide variety of heterogeneous computing resources, due to both the diversity of computation kernels and the lack of a single architecture meeting all their requirements. In addition, integrating different architectures in a heterogeneous platform seems the only promising approach to achieve scalable performance with power efficiency [2].

Porting sequential applications to heterogeneous HPC systems for achieving high performance requires significant effort and time to rewrite and optimize the applications for every target device. In addition, end users need extensive software and hardware expertise to *manually* analyze the target architectures and applications to determine the best performing architecture and implementation technique for each workload. To streamline this tedious process, programmers need appropriate tools to automatically predict the best hardware architecture for their workloads and estimate the potential performance on such architectures without the need for extensive architecture expertise and writing parallel code for every target device.

To this end, this paper presents AutoMatch, a tool for automated matching of compute kernels to heterogeneous HPC architectures. AutoMatch analyzes the *sequential* application code to estimate the benefits of porting this application to heterogeneous systems. It is a hybrid approach that leverages static and dynamic analysis techniques to extract the architecture-agnostic characteristics of the sequential applications. Next, it combines these characteristics with the specifications of the target heterogeneous system to automatically construct high-level performance models, and predict the performance of the best *parallel* implementation on the different architectures. This performance prediction is then used to identify the best performing architecture for each workload from a set of architectures including multi-core CPUs and many-core GPUs. In addition, AutoMatch estimates the relative execution cost on the different hardware architectures to drive a workload distribution and partitioning scheme, which enables end users to efficiently exploit the available compute resources across multiple heterogeneous architectures.

AutoMatch is designed as a *first-order* performance prediction tool to help programmers assess the benefits of porting their sequential applications to heterogeneous systems before investing effort and time in rewriting and refactoring the applications for every target devices. While our automatically-generated models are simple and intuitive, they work surprisingly well on predicting the relative performance across different architectures and the best workload distribution strategy. Moreover, AutoMatch works on the LLVM intermediate representation (IR) [3], which makes it language-independent and applicable to any source code supported by the LLVM front-ends (e.g., C/C++, FORTRAN, and so on).

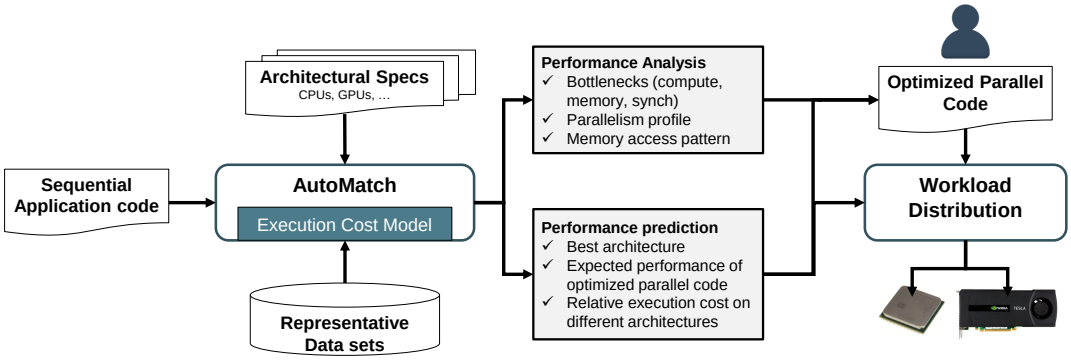

Fig. 1: AutoMatch Framework.

Our tool differs from previous approaches in that it does *not* require the availability of the target platforms or the parallel application code for each platform. In addition, AutoMatch is automated and applicable to different types of hardware architectures with minimal efforts. In summary, the following are the contributions of this work:

- AutoMatch's compiler analyzes the sequential applications on a theoretical architecture with infinite resources and ideal cache-memory model to automatically extract their architecture-agnostic features, such as the inherent parallelism, data locality, memory-access pattern, and synchronization pattern (Section III).
- AutoMatch automatically constructs high-level performance models that can be generalized for different architectures to estimate the computation time, memory-access time, and synchronization overhead, as well as can predict the performance of the best *parallel* implementation on the different architectures (Section III).
- AutoMatch automatically estimates the relative execution cost on the different hardware architectures to drive a workload distribution scheme to efficiently exploit the available compute resources across multiple heterogeneous architectures (Section IV).
- Using a set of open-source HPC applications and benchmarks, with different parallelism profiles and memory-access patterns, we show that AutoMatch is highly accurate across five different heterogeneous architectures, identifying the best architecture for each workload in 96% of the test cases. In addition, the performance of its workload distribution scheme is comparable to an oracle based on profiling of the *parallel* code (Section IV).

II. AutoMatch Overview

AutoMatch analyzes the sequential applications and automatically predicts the best hardware device for them from a set of heterogeneous devices. The key insight is that AutoMatch leverages static and dynamic analysis techniques to quantify the maximum parallelism, the maximum data locality and the minimum synchronization of the sequential code to estimate the potential performance of the best parallel implementation. Moreover, by automatically generating *high-level* performance models, AutoMatch can generalize these models and predict the performance on different types of hardware devices.

Figure 1 shows the overall framework of AutoMatch. It takes as inputs the sequential code, the target architecture specifications (which are automatically generated via micro-benchmarking), and representative input data. AutoMatch automatically constructs and evaluates the Execution Cost (EC) model to generate useful performance predictions, including the best architecture for the target workload, the potential performance of the best parallel implementation on each architecture, and the expected relative execution cost between the different architectures. In addition, AutoMatch provides detailed information about the inherent characteristics of the sequential code, such as the parallelism profile, data reuse, memory access pattern, and bottlenecks (compute, memory, or synchronization). Such information can help the user to decide the best optimization and parallelization strategy for the application. Further, AutoMatch uses the relative execution cost between the heterogeneous architectures to promote the development of a run-time workload distribution service that utilizes multiple heterogeneous devices at the same time.

A. AutoMatch Design and Implementation

To automatically construct the EC model, we use the LLVM compiler framework [3] and combine static and dynamic analyses to utilize the information available on LLVM IR of the sequential code. Figure 2 shows the current design and implementation of AutoMatch. Clang and other front-ends parse the sequential code of the target application and emit its IR without any optimization. In case of multiple IR files, LLVM-LINK merges them into one file. Next, OPT performs a set of canonicalization passes on the unoptimized LLVM IR. While the most important pass is the memory-to-register translation, which promotes all temporal stack memory allocation and accesses to registers and converts IR into the single static assignment (SSA) form, other passes such as function inline and constant propagation simplifies the induction variables and control flow and make the analysis easier. In addition, the user provides the input data and the target kernel name. After that, AutoMatch, which is implemented in the execution engine of the dynamic compiler LLI, statically and dynamically analyzes the optimized IR to extract the architecture-agnostic characteristics of the sequential code, and combine them with specifications of the target heterogeneous system to generate the final performance analysis and predictions.

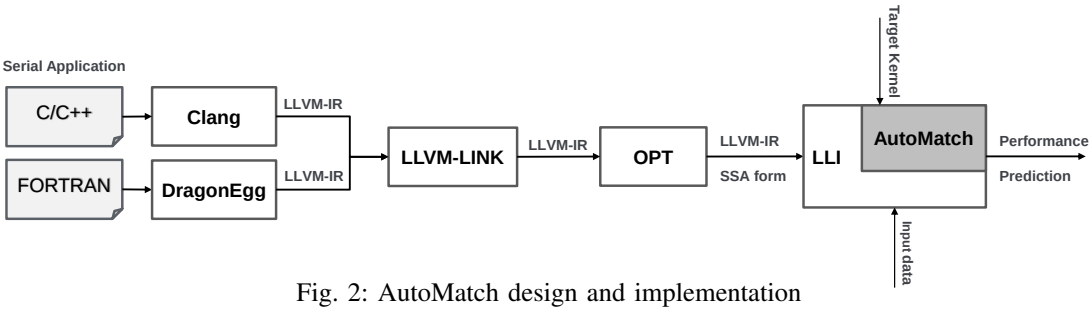

Fig. 2: AutoMatch design and implementation

III. AutoMatch Performance Prediction

To identify the best architecture for the target application, AutoMatch automatically constructs an analytical performance model, the Execution Cost (EC) model, which captures the complex interaction of the application, input data and target architectures. In addition, we consider different types of hardware devices including CPUs and GPUs, and obtain the hardware architecture specifications in terms of the computation performance, the memory system latency and bandwidth, and the synchronization overhead using micro-benchmarks.

A. Hardware Architecture Model

We propose an abstract hardware architecture model that can be generalized to different shared-memory architectures including multi-core CPUs and many-core GPUs. The proposed model extends the classical external memory model [4], [5] to parallel architectures, and considers important constraints on such systems, such as the on-chip memory access time and the synchronization overhead.

Figure 3 shows the proposed hardware architecture model, which consists of multiple compute cores that are connected to a shared on-chip fast memory and off-chip slow memory. The compute cores can only perform operations on data in their on-chip private memory, and each core executes floating-point operations at a peak computing rate of π_0 FLOPs per second. The floating-point throughput of the architecture is $\Pi = n_p \times \pi_0$, where n_p is the number of compute cores. The shared fast memory is a fully associative memory with a size of Z words, and it uses the Least Recently Used (LRU) replacement policy. The data is transferred between the compute cores, the fast memory and the slow memory in messages of L words. The on-chip memory interconnect has a latency α_f and a bandwidth β_f, while the off-chip memory has a memory access latency α_s and a memory bandwidth β_s.

To reach a globally consistent memory state, the compute cores perform synchronization operations whose cost depends on the memory latency and the number of compute cores. Since the synchronization overhead, s_0, significantly affects the execution time on parallel architectures, especially at higher core counts [6], [7], the proposed model considers this overhead. There are two synchronization types: global synchronization, between coarse-grain threads with different control units (threads on CPUs and thread blocks on GPUs), and local synchronization, between fine-grain threads with

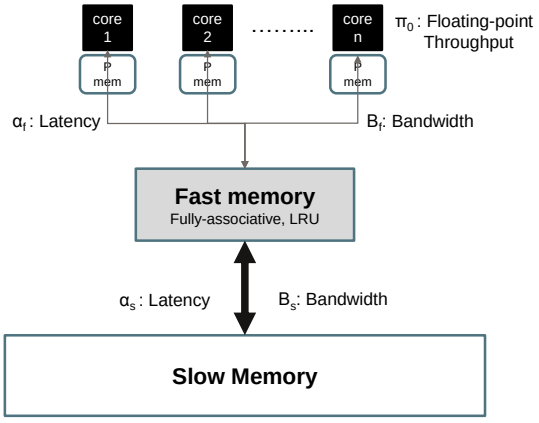

Fig. 3: The abstract hardware architecture

shared control units (SIMD lanes on CPUs and threads on GPUs). Considering the local synchronization overhead is negligible in comparison with the global synchronization (usually more than order of magnitude lower) [6], [7], the proposed model ignores it.

Since the main goal is to match the applications to the best architecture from a set of parallel architectures that are fundamentally different, the proposed hardware architecture model is high-level and does not capture architecture-specific parameters and low-level hardware details, e.g., hardware prefetchers and complex memory hierarchies. In addition, it ignores several initialization and finalization overheads, such as threads creation/destruction, kernel launch and host-device data exchanges, which are highly-dependent on the run-time environment and usually are one-time cost. In section IV, we show the sensitivity of our performance prediction to the variations of the architecture capabilities.

1) Inferring the Architecture Specifications: AutoMatch generates the hardware architecture specifications using micro-benchmarks. In particular, it uses ERT [8], pointer-chasing [9], [10], and synchronization [6], [11] micro-benchmarks to estimate the floating-point throughput and memory bandwidth, the memory access latency, and the global synchronization overhead respectively. To analyze the effectiveness of AutoMatch, we consider five architectures (two CPUs and three GPUs) with different core counts and execution models. We further divide these architectures into three subsets: ($ARC1$, $ARC3$, $ARC5$), ($ARC1$, $ARC2$), and ($ARC4$, $ARC5$). The first subset contains three significantly different architectures with

TABLE I: Hardware architecture specifications

Model	**Intel i5-2400**	**Intel i7-4700**	**Tesla C2075**	**Tesla K20C**	**Tesla K20X**
ID	$ARC1$	$ARC2$	$ARC3$	$ARC4$	$ARC5$
Clock (GHz)	3.1	2.4	1.15	0.732	0.732
Process (nm)	32	22	40	28	28
n_p	4	4	448	2496	2688
π_0 (GFLOPS)	20	33	0.9	0.41	0.42
Z (MB)	6	6	1.6	2.3	2.3
L (Byte)	64	64	128	128	128
β_f (GB/s)	285	349	2117	2018	2424
α_f (us)	0.004	0.004	0.028	0.045	0.045
β_s (GB/s)	18.88	11.5	87.92	129.73	160.1
α_s (us)	0.065	0.052	0.71	0.68	0.68
s_0 (us)	0.2	0.44	7.22	6.5	6.5

few cores, hundreds of cores and thousands of cores, while the second and third subsets have two slightly different CPUs and GPUs respectively. Table I summarizes the specifications of the target architectures.

Since modern on-chip memories have inclusive memory levels, AutoMatch chooses the fast memory size, Z, to be the effective on-chip memory capacity. On CPUs, Z is the last level data cache; on GPUs, Z is the shared (local) memory and L2 cache. While the proposed architecture model represents on-chip memory as a unified fast memory, actual on-chip memories have complex hierarchies with multiple levels and some levels are physically distributed (such as L1/L2 on CPUs and local memory on GPUs). Therefore, AutoMatch estimates the fast memory bandwidth and latency, β_f and α_f, as the average memory bandwidth and latency of the on-chip memory hierarchy. In comparison with the slow memory, the fast memory of the target architectures is better by a factor of 15 approximately in terms of memory bandwidth and latency. The only exception is $ARC2$, where the memory bandwidth ratio between the fast and slow memories is ≈ 30. Finally, AutoMatch estimates the global synchronization cost, s_0, using barrier synchronization between threads on CPUs and thread-blocks on GPUs. There are several inter-block synchronization methods on GPUs, AutoMatch uses the host-implicit inter-block synchronization, which is the simplest and most popular one [6]. Since the number of active threads can significantly affect the synchronization overhead, AutoMatch estimates the global synchronization cost at full occupancy, i.e. it launches one thread per logical core on CPUs and four thread-blocks of dimension 32×32 per streaming multiprocessor on GPUs.

B. Computation Time Prediction

AutoMatch combines both static and dynamic analysis techniques to automatically quantify the inherent parallelism in the sequential applications, and estimate their computation time on the different architectures for a given input data. In particular, it schedules the application on a theoretical architecture with infinite number of registers and compute units, and zero memory access latency, such that each operation is executed as soon as its true dependencies are satisfied.

Figure 4 depicts the As Soon As Possible (ASAP) schedule of the application on the theoretical architecture, where the nodes are dynamic instances of the floating-point instructions (operations), denoted as I_{nm}, and the edges are true dependencies between the operations. Each dynamic instance m of a floating-point instruction I_n is scheduled at an execution level j as soon as its true dependencies are satisfied, hence, I_{nm} must have dependencies at the execution level $j-1$.

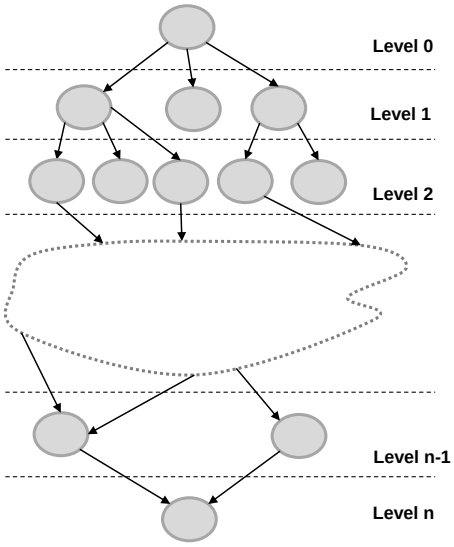

Fig. 4: The application ASAP schedule on a theoretical architecture with infinite resources

This ASAP schedule is similar in spirit to the classical work-depth model [12], [5], which represents the computations and inherent parallelism of a given algorithm using a directed acyclic graph (DAG), where nodes represent operations and edges are their dependencies. While the classical work-depth model requires manual analysis to quantify the sequential part and average parallelism of a given algorithm, AutoMatch automatically generates the ASAP schedule to estimate the computation time, and considers the workload imbalance, the vectorization potential, the instructions mix and the resource constraints of the target architectures.

To identify the true dependencies between operations, AutoMatch uses several static and dynamic analysis techniques. First, it leverages the existing def-use static analysis to track data-flow dependencies through registers. Due to the use of infinite number of registers, only read-after-write true dependency exists. Second, AutoMatch extends the execution engine of LLI to dynamically track the incoming instructions to the phi nodes in every basic block, and uses this analysis to flatten the control flow dependencies. Third, AutoMatch implements a dynamic analysis technique to track data-flow dependencies through the memory operations. It uses hash tables that resembles Content-Addressable Memory (CAM) to record the load and store accesses to the memory addresses, and record which instruction generated them and when they are generated in terms of the execution level. Next, it dynamically detects read-after-write, write-after-read and

write-after-write memory dependencies by tracking memory accesses on the use-def chain and examining the CAM data structure. Finally, AutoMatch adjusts the execution level of the operations based on the detected true dependencies to construct the final operation schedule.

After building the application schedule on the theoretical architecture, AutoMatch analyzes it to compute D, the number of execution levels (i.e. the depth of the critical path), and w_i, the total number of operations for each execution level i. In addition, it considers the instructions mix of the sequential application to estimate f_{im}, the performance degradation factor relative to the peak floating-point throughput (π_0) on parallel architectures with Fused Multiply-Add (FMA) units. The instruction mix factor f_{im} is computed as:

$$f_{im} = \frac{W_{add} + W_{mul}}{2 \times \max(W_{add}, W_{mul})} \tag{1}$$

where W_{add} is the number of addition and subtraction operations, and W_{mul} is the number of multiplication operations.

Moreover, AutoMatch leverages the LLVM vectorizer to identify the loops that are amenable to vectorization, and computes W_{vec}, the number of floating-point operations that can efficiently utilize the vector (SIMD) units. Next, it estimates f_v, the performance degradation factor relative to the peak floating-point throughput on parallel architectures with vector units, as follows:

$$f_v = \frac{W_{vec}}{W} \tag{2}$$

where W is the total number of floating-point operations.

Finally, AutoMatch combines the computation characteristics of the sequential application with the specifications of the target architectures to predict the computation cost on each architecture. The computation time T_{comp} is estimated as:

$$T_{comp} = \frac{D}{\pi_0} + \underbrace{\sum \frac{w_i}{\min(w_i, n_p) \times (\pi_0 \times f_v \times f_{im})}}_{\forall i} \tag{3}$$

where n_p is the number of cores, π_0 is the maximum operations throughput per core, f_v is the vectorization factor, and f_{im} is the instruction mix factor.

C. Memory Access Time Prediction

To predict the memory access time, AutoMatch quantifies the inherent data locality in the sequential applications by analyzing their memory access pattern on the abstract hardware architecture model, which has an ideal cache-memory model. The main goal is to estimate the number of data transfers between the compute cores and the shared fast memory Q_f, and between the shared fast and slow memories Q_s. Since the proposed architecture model assumes that the fast memory is fully associative and uses the LRU replacement policy, AutoMatch adopts the LRU stack distance analysis [13].

The LRU stack distance is defined as the number of distinct memory locations accessed between two consecutive accesses to the same memory location, given that the LRU stack distance of the first reference to a memory location is ∞. Figure 5 shows an example of the LRU stack distance analysis on a memory access trace of 10 memory references.

Memory location accessed	a	c	d	b	c	e	g	e	d	d
LRU stack distance	∞	∞	∞	∞	2	∞	∞	1	4	0

Fig. 5: An example of the LRU stack distance analysis

In fully-associative caches with LRU replacement policy, a memory reference with an LRU stack distance larger than the cache size results in a miss or an access to the slow memory. Hence, Q_s and Q_f can be estimated from the number of memory references with an LRU stack distance larger than the fast memory size, and the number of memory references with an LRU stack distance less or equal to the fast memory size respectively. While the LRU stack distance analysis ignores the conflict and contention misses, AutoMatch assumes that the actual number of memory transfers on the parallel architectures are bounded by Q_s and Q_f [14].

AutoMatch automatically estimates the memory access cost of the target application and input data as follows. First, it dynamically analyzes the IR instructions stream to capture the load and store memory operations, and uses a binary search tree to record the referenced memory locations along with the index of the last access to these locations in the memory access stream. The nodes of this binary tree are sorted by the last access index. Second, whenever a memory location is referenced, AutoMatch examines the memory tree to find the last access index; if the target memory location does not exist in the memory tree, the current memory access has an LRU stack distance of ∞, otherwise, AutoMatch finds the nodes with a last access index between the last access to the target memory location and the current access; the number of such nodes is the reuse distance of the current memory reference. Third, AutoMatch counts the number of memory references with a particular LRU stack distance to generate the LRU stack distance histogram. Finally, it combines this histogram with the specifications of the target architectures and the ASAP schedule of the application to compute Q_f and Q_s, and estimate T_{mem}, the memory access time of the target application on each architecture, as follows:

$$T_{mem} = (\alpha_f + \alpha_s) \times D + (\frac{Q_f}{\beta_f} + \frac{Q_s}{\beta_s}) \times L \tag{4}$$

where α_f and α_s are the access latency of the fast and slow memories, β_f and β_s are the memory bandwidth of the fast and slow memories, D is the depth of the application ASAP schedule, and L is the memory transfer size.

D. Synchronization Overhead Prediction

AutoMatch uses a heuristic for estimating the required number of global synchronization points to reach a globally consistent memory state on parallel architectures. The proposed heuristic is based on detecting loop-carried memory dependencies. AutoMatch dynamically analyzes the loop nests of the sequential application to find the inherently sequential

TABLE II: Target workloads

Workload	Description	Input data
CUTCP	Molecular-dynamics simulation of explicit-water biomolecular model that computes the Cutoff Coulombic Potential over a 3D grid	watbox.sl40.pqr
STENCIL	Iterative Jacobi solver on a structured 3-D grid	Grid 512x512x64
SPMV	Sparse matrix vector multiplication	Dubcova3.mtx
LBM	Lid-driven cavity simulation using the Lattice-Boltzmann Method	120_120_150_ldc.of
LUD	LU decomposition on a dense matrix	Matrix 512^2
LavaMD	Molecular-dynamics simulation that calculates the potential due to mutual forces between particles in a 3D space	boxes1d 10
HotSpot	Thermal simulation and modeling for VLSI designs	temp_1024 power_1024
SRAD	Image processing used to remove locally correlated noise, known as speckles	image 512^2

loops, i.e. loops that can not run in parallel due to loop-carried memory dependencies, and the parallel loops. It estimates the number of global synchronization points as the trip counts of the inherently sequential loops with inner parallel loops. Figure 6 shows an example of this case, where the i-loop is inherently sequential, and the j-loop is parallel and the number of global synchronization points is $n-2$.

```
for(i=1; i<n; i++)
{
  for(j=1; j< n; j++)
  {
    a[i][j] = a[i-1][j] + 2;
  }
}
```

Fig. 6: Detection of global synchronization

While this heuristic successfully identified the number of global synchronization points in the target benchmarks and applications, AutoMatch enables the user to override the synchronization estimation heuristic and to manually annotate the source code to indicate the global synchronization points. Finally, the synchronization time, T_{syn} , is estimated as:

$$T_{syn} = S \times s_0 \tag{5}$$

where S is the total number of global synchronization points, and s_0 is the global synchronization cost.

E. The Execution Cost

After analyzing the parallelism profile, the data locality and the synchronization pattern of the target sequential application, AutoMatch evaluates equations 1-5 to predict the execution cost on each architecture, which is estimated as the overall computation time, memory access time and global synchronization overhead. Next, AutoMatch combines the execution cost on the different architecture with the floating-point work of the target application to predict the performance of the best *parallel* implementation on the different architectures, the best architecture for the user workload, and the relative execution cost between the different architectures.

IV. Experimental Results

In this section, we demonstrate the efficacy of AutoMatch, and its utility as a first-order performance prediction tool for sequential applications on heterogeneous HPC architectures. We evaluate AutoMatch using eight HPC workloads from Rodinia [15] and Parboil[16] benchmark suites with different parallelism profiles and memory access patterns, and MiniGhost, a representative Computational Fluid Dynamics (CFD) application [17]. We choose Rodinia and Parboil benchmark suites as they provide sequential and multi-threaded CPU implementations, and GPU implementations.

Table II presents the workloads considered in this study, and we use the input data sets provided by their benchmark suites. In the experiments, we use the following compilers: gcc 4.8.2, icc 13.1.1 and nvcc 6.0.1, and AutoMatch is implemented in LLVM-3.6.2. While AutoMatch works with any data-type supported by LLVM, we consider double-precision floating point only for brevity. In addition, the reported performance is for the core computation kernels and ignores one-time cost overheads such as I/O, data initialization (including host-device data transfer), profiling, timing and debugging.

In the evaluation, we answer the following questions:

- What is the accuracy of AutoMatch's prediction of the best architecture (and the relative ranking of the different architectures) for the test workloads?
- Can AutoMatch predict the performance upper-bound on heterogeneous parallel systems?
- Is the predicted performance of the best parallel implementation attainable by actual implementations?
- What is the sensitivity of AutoMatch to the variations of the architecture characteristics and capabilities?
- What is the performance of AutoMatch-driven workload distribution in comparison with a profiling-driven oracle?

A. Performance Prediction

We use AutoMatch to analyze the sequential implementation of the target applications and show the performance prediction in comparison with the actual performance of the parallel implementations on the different architectures.

Figure 7 and 8 present the parallelism and LRU stack distance profiles of the target workloads. Due to the space limit, we show the detailed profiles of only three workloads:

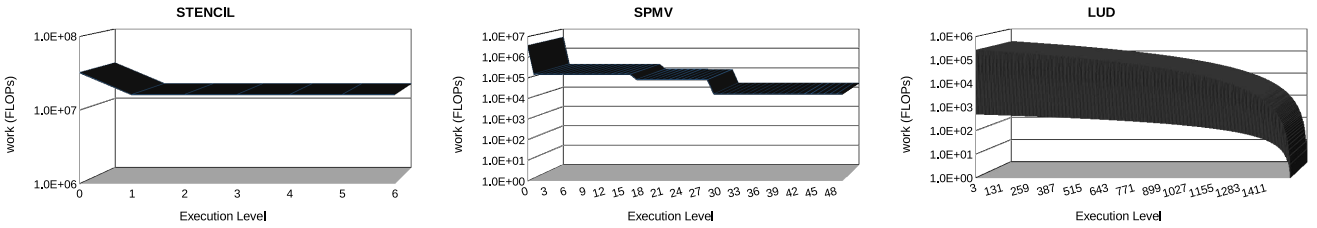

Fig. 7: Parallelism profile

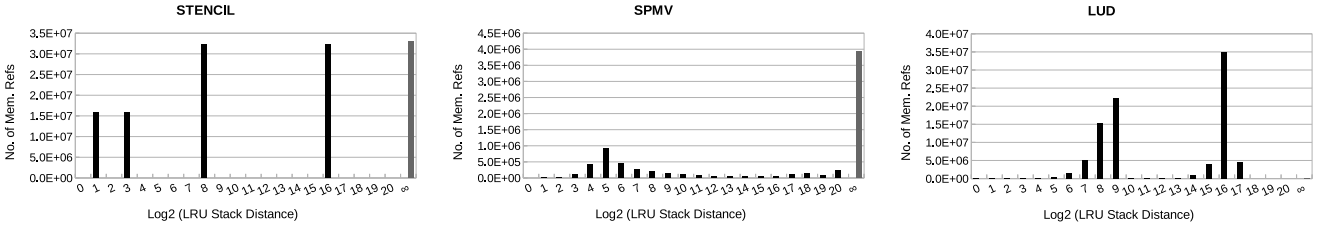

Fig. 8: LRU stack distance profile

STENCIL, SPMV and LUD. AutoMatch indicates that the STENCIL benchmark is inherently parallel with few execution levels and massive amount of work in every execution level, and it has a uniform memory access pattern with few memory streams corresponding to the dimensions of the data grid. The SPMV benchmark has relatively small number of execution levels, however, the amount of work per execution level is significantly lower than the STENCIL benchmark, due to the sparsity of the input matrices. In addition, SPMV suffers from low data locality, as the compulsory misses (memory operations with LRU stack distance ∞) dominates the memory accesses. The LUD benchmark has an irregular parallelism profile that alternate between two bounds corresponding to the computation of the pivot column and the update of the trailing sub-matrix respectively, and the amount of work per execution level decreases as we move down the critical path of the application schedule, which results in workload imbalance. Moreover, LUD has scattered memory access streams, because the data accessed decreases as the execution progress due to the workload imbalance.

Figure 9 shows AutoMatch's performance prediction in comparison with the actual performance of the parallel OpenMP and CUDA implementations, and Figure 10 provides AutoMatch analysis of the execution bottlenecks on the different architectures. We consider the first subset of the target architectures ($ARC1$, $ARC3$ and $ARC5$), which contains heterogeneous architectures with significantly different hardware characteristics and capabilities. The results show that the actual parallel implementations never exceed AutoMatch's prediction, which indicates that AutoMatch accurately predicts the performance upper bound (i.e. the performance of the best parallel implementation). Moreover, AutoMatch accurately identifies the best architecture and the relative ranking of the different architectures in all the test cases. While the gap between the predicted performance and the actual performance on the many-core GPUs ($ARC3$ and $ARC5$) is small in most cases (except lavaMD and CUTCP), the performance prediction gap is very large on the multi-core CPU ($ARC1$). After inspecting the actual parallel implementations, we found that the benchmark suites provide a baseline and unoptimized OpenMP implementation, while the CUDA implementation is optimized for Nvidia GPUs.

To show that the predicted performance is attainable, we optimized the STENCIL benchmark with the help of AutoMatch's analysis. AutoMatch indicates that STENCIL is bounded by the off-chip memory access time, and it has few memory access streams corresponding to the dimensions of the input data grid. We found that the original workload distribution strategy (of the baseline OpenMP implementation) partitions the input data grid along the X-axis, which has the smallest reuse distance or highest locality, and distributes chunks of Y-Z planes over the different threads. Hence, we changed the workload distribution strategy to distributes chunks of X-Y planes over the different threads. As shown in Figure 9, the performance of our implementation, named STENCIL-OPT, is significantly better than the original implementation on $ARC1$, which means that the performance predicted by AutoMatch can be achieved with platform-specific optimizations and tuning.

Finally, the gap between the predicted performance and the actual performance of the many-core GPUs is relatively large in the lavaMD and CUTCP benchmarks, which are bounded by the compute time and on-chip memory access time according to AutoMatch analysis. The analysis of the actual CUDA implementations of lavaMD and CUTCP show that they suffer from low occupancy (37% and 27%) and the number of the concurrently active threads is low. The main reason is that the two benchmarks have high registers usage, which limits the number of concurrent threads and thread-blocks. Hence, extending AutoMatch to predict the possible occupancy on many-core GPUs would improve the performance prediction in these benchmarks. However, there is a trade-off between this additional prediction accuracy, and the generalization of the Execution Cost model to different types of architectures.

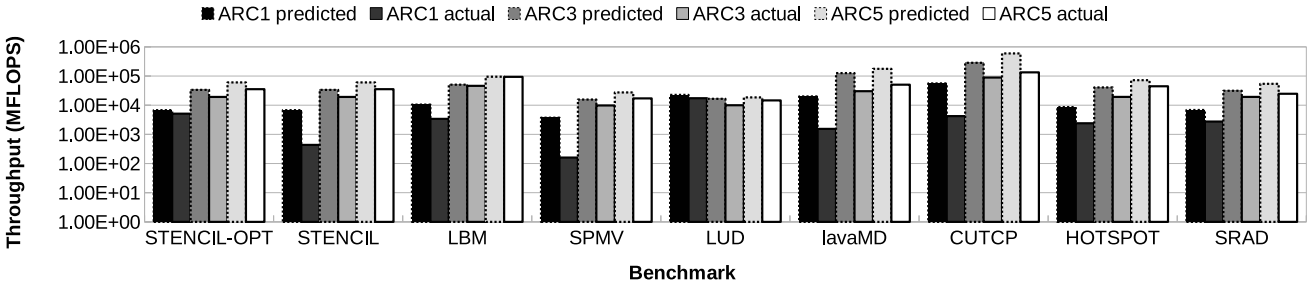

Fig. 9: AutoMatch performance prediction

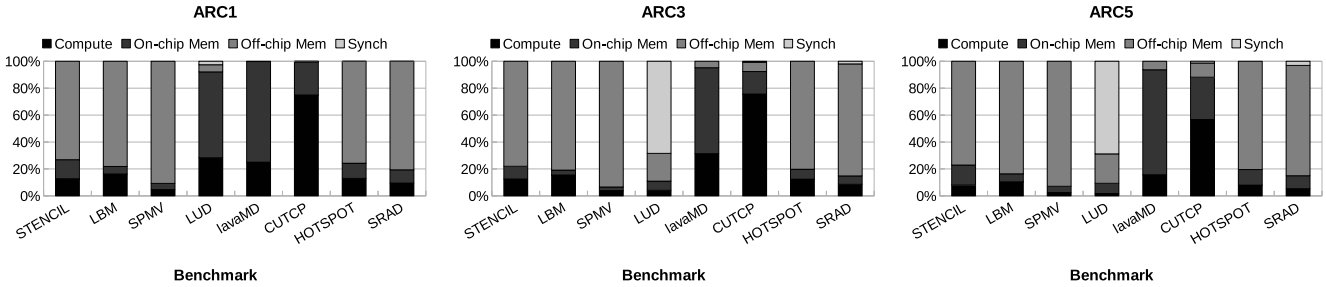

Fig. 10: AutoMatch bottlenecks prediction

B. AutoMatch Sensitivity

To analyze the sensitivity of AutoMatch's performance prediction to the variations of the architecture capabilities, we consider the second and third subsets of the target architectures, which contain multi-core CPUs ($ARC1$ and $ARC2$) and many-core GPUs ($ARC4$ and $ARC5$) architectures with similar hardware characteristics and capabilities.

Figure 11 shows AutoMatch's performance prediction and the actual performance on the two architectures subsets. Surprisingly, AutoMatch accurately predicts the best architecture in all the test cases, except the LUD benchmark on multi-core CPUs, which shows that our high-level performance models are sensitive to the small variations of the target architectures and can match the compute kernels to different types of parallel architectures. For the LUD benchmark, AutoMatch indicates that it is bounded by the fast memory access time on multi-core CPUs ($ARC1$ and $ARC2$), and its parallelism and LRU stack distance profiles show a non-uniform memory access pattern, where the data accessed decreases as the execution progress due to the workload imbalance. Hence, our hypothesis is that the higher memory bandwidth of $ARC2$ is underutilized due to the non-uniform memory access pattern of the LUD benchmark, which leads to the incorrect performance prediction. While our high-level memory access model captures the data locality of the target applications, it does not consider the uniformity of the memory access pattern and its effect on several hardware features, such as hardware prefetchers, memory coalescing units and write buffers. In addition, the micro-benchmarking approach has the same limitation, as it uses a stream-like memory access pattern to measure the memory bandwidth of the target architecture.

C. Workload Distribution

We use MiniGhost [17], [18], a representative CFD application to show the effectiveness of our workload distribution scheme (based on the EC model generated by AutoMatch) in comparison with an oracle (based on run-time profiling of actual parallel implementations). MiniGhost is a proxy for multi-material, hydrodynamics code that models hydrodynamic flow and dynamic deformation of solid materials [19]. The main computation kernel is the finite difference solver, which applies a difference stencil and explicit time-stepping scheme on a homogenous 3D grid. We use the implementation provided by the MetaMorph library [20], which supports the seamless execution of structured grid applications on multiple heterogeneous devices, including CPUs (OpenMP back-end) and GPUs (CUDA back-end). In addition, we configure MiniGhost to apply a 3D 7-point stencil on a single global grid and to use an explicit time-stepping with 100 time steps.

The target platform is a heterogeneous CPU-GPU node that includes $ARC1$ and $ARC5$ devices, and the main goal is to partition and distribute the global grid over the available devices to reduce the overall execution time. We evaluate three different workload partitioning: default, AutoMatch-driven and Oracle-driven partitioning. The default strategy is to partition the input grid evenly into two parts and assign each part to one of the available devices. The AutoMatch-driven workload partitioning uses AutoMatch to analyze the sequential implementation and predict the execution cost on the heterogeneous devices. Next, based on the predicted execution cost, it distributes the global grid to minimize the overall execution time. For example, when AutoMatch predicts that the execution cost on the CPU and the GPU is 3 and 1,

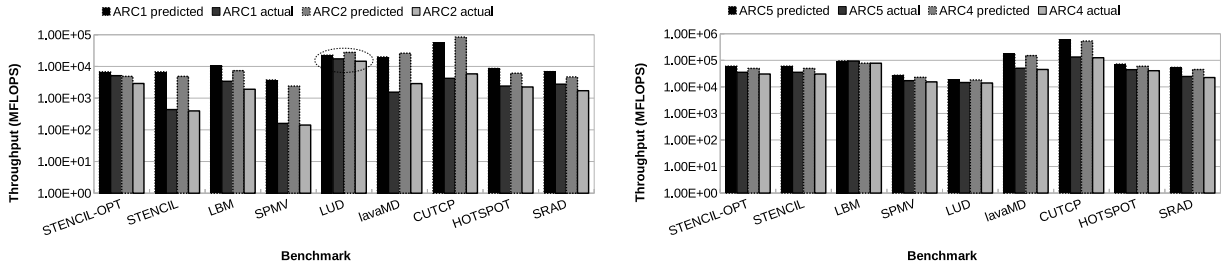

Fig. 11: AutoMatch prediction sensitivity

we partition the global grid to four parts and assign three parts to the GPU and one part to the CPU. The Oracle-driven partitioning is similar to AutoMatch-driven strategy; however, instead of predicting the execution cost, it profiles the parallel code on the target CPU and GPU and distributes the global grid over them based on the *measured* execution time.

Figure 12 shows the overall execution time of MiniGhost with the different workload distribution strategies, and the runtime distribution between the CPU and the GPU. Surprisingly, the AutoMatch-driven and Oracle-driven partitioning achieve the same performance and outperform the default strategy by a factor of 3 on average. However, the AutoMatch-driven strategy has a higher workload imbalance, between the CPU and the GPU, than the Oracle. In particular, AutoMatch underestimates the CPU performance relative to the GPU at the small grid sizes, and assigns more work to the GPU. One reason is that the small grids fit on a higher memory hierarchy level, and AutoMatch approximates the on-chip memory hierarchy as a single fast memory. While this workload imbalance did not affect the overall execution time, it would be interesting to investigate its effect on the power consumption, since the CPU and the GPU have different power characteristics.

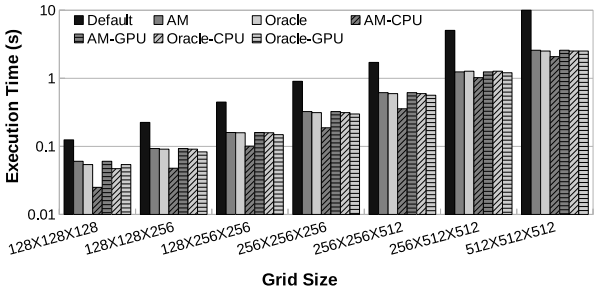

Fig. 12: MiniGhost performance (compute kernels) on a heterogeneous CPU-GPU node ($ARC1$ & $ARC5$) with the different workload distribution strategies: Default, AutoMatch (AM) and Oracle.

D. Limitations and Extensions

While the results show that AutoMatch works surprisingly well as a first-order performance prediction tool regardless of its simple and intuitive model, the tool has several limitations. First, similar to any dynamic analysis approach, AutoMatch's prediction depends on the input data and can change across multiple inputs; however, programmers can use AutoMatch with input data sets that represent their typical use cases. Second, AutoMatch ignores one-time overheads such as host-device data transfers, and assumes that the performance is dominated by the compute kernels. While this is a valid assumption for long-running HPC applications, extending AutoMatch to model the host-device interconnect and data transfers enables the users to explore their effect on the overall performance. Third, AutoMatch ignores low-level, architecture-specific features such as HW prefetchers, memory coalescing, thread divergence and occupancy. Although AutoMatch can be extended, beyond its main goal as a first-order performance prediction tool, to incorporate more sophisticated models (e.g. [21]), there is a trade-off between the tighter performance bounds and both the generalization to different architecture types and the limited insight about the critical parameters that affect the performance.

V. Related Work

According to Hoefler et al. [22], the main approaches to predict the performance of an application on computing architectures are profiling, simulation, and analytical modeling.

Profiling. Profiling runs the actual code on the target platform, and uses performance counter and timers to measure the achieved performance. Although profiling captures the interactions between the application and the execution architecture, it provides limited information about the critical parameters that affect the performance. In addition, profiling requires the availability of the target platform and actual parallel code.

Simulation. Detailed simulation [23], [24], [25] can provide accurate performance prediction of the application without the availability of the target hardware. However, similar to profiling, it needs the parallel code, and the predicted performance depends on the end user ability to prallelize and optimize the application to the simulated architecture.

Analytical Modeling. Analytical modeling maps both the application and architecture to a set of parameters and mathematical expressions that can be evaluated to predict the execution time. Usually, there is a tradeoff between the number of parameters and the accuracy of the model. Performance-bound models (e.g. Roofline [26]) have few parameters to provide high-level view of the interaction between the application and the architecture. However, they are not suitable for performance predication, as they abstract away critical factors, e.g. parallelism, data locality and synchronization overhead. In addition, they are not automated and require extensive manual analysis of the applications and hardware architectures.

TABLE III: Comparison of recent performance prediction tools for heterogeneous HPC architectures (CPUs and GPUs)

	COMPASS	**XAPP**	**AutoMatch**
Input code	Annotated	Sequential	Sequential
Features extraction	Static analysis	Dynamic analysis	Static/dynamic analysis
Arch model generation	By users	Training data	Micro-benchmarking
Performance modeling	ASPEN model	Machine-learning	Execution cost model
Cache-aware	No	Yes	Yes
App generality	Low	High	High
HW generality	High	Low	High
The tool speed	Fast	Slow	Moderate

Helal et al. [11] show an example of matching the compute kernels to heterogeneous HPC platforms with a large-scale circuit simulation application. However, the the performance models are manually constructed, which requires extensive analysis of both the target architectures and the application.

Automatic performance modeling. Recently, several approaches have been proposed to automate the performance prediction using static and/or dynamic analysis. While static analysis is fast, it requires guidance from the user (e.g. via annotations). Conversely, dynamic analysis can identify the application characteristics without the user guidance, but it is more complex and slower than static analysis. Table III summarizes the comparison of the recent performance prediction tools for heterogeneous HPC architectures (CPUs and GPUs).

COMPASS [27] is a tool for automated performance modeling. It generates a structured performance model (ASPEN Model) from the parallel application code using static analysis. However, the user must indicates the available parallelism and data movement to generate an accurate model. Otherwise, COMPASS uses Banerjee-Wolfe dependency analysis, which can not detect the data dependency through memory operations and generates a conservative parallelism profile. In addition, COMPASS can not be used with irregular applications, where the computation and memory access patterns are data-dependent, such as sparse linear algebra (SPMV and SPLU).

XAPP [28] uses machine-learning to find the correlation between the execution profile of the application on a CPU and the GPU execution time. However, XAPP is heavily influenced by the training data and its prediction accuracy depends on the availability of a diverse set of applications along with their optimized GPU implementation. So, extending XAPP to new architecture types requires massive effort to rewrite and re-optimize the training set to these architectures. Moreover, to predict the performance on a specific GPU device, the user needs to run the whole training set on this device, which takes hours. On the contrary, AutoMatch generates the device parameters using micro-benchmarks, which takes few minutes. In addition, XAPP's predicted speedup is not the speedup upper-bound, and it depends on which optimization techniques are used in the training application set.

Kismet [29] predicts the speedup of serial applications on multi-core processors. It instruments the code to build the self-parallelism profile, and estimates the memory access latency by profiling the cache misses of the input application on a CPU cache simulator. Hence, it requires simulating the memory system hierarchy of each target architecture. On the contrary, AutoMatch analyzes the application once to estimate its data locality and memory access time. In addition, Kismet optimistically assumes that the memory bandwidth is scalable with the number of threads, which is unrealistic assumption especially for massively parallel architectures such as GPUs. Therefore, its predicted speedup is unattainable at higher core counts and for memory-bound workloads. Parallel Prophet [30] predicts the speedup of the annotated code on multi-core CPUs. Unlike Kismet, it does not require parallelism discovery and uses annotations to identify the available parallelism. In addition, it uses hardware performance counters, such as instruction counts and cache misses, to build the performance model, which requires the availability of the target CPUs and the parallel (or annotated) code to predict the speedup.

Shen et al. [31] present a workload partitioning framework for heterogeneous platforms. The framework computes the partitioning ratio by profiling the actual parallel code to estimate the relative hardware capabilities and the host-device data transfer overhead. Conversely, AutoMatch estimates the workload distribution ratio by analyzing the sequential code. While AutoMatch assumes that the performance is dominated by the compute kernels, it can be extended to model the host-device interconnect and the data transfer overhead. LACross [32] is a framework for performance and power prediction of single-core workloads on embedded platforms. It uses statistical learning approach to find the correlation between the execution on the host and the target embedded devices. AutoMatch can be extended to such embedded platforms to improve the overall performance and power efficiency by mapping the workloads to the appropriate core.

VI. Conclusion

In this paper, we proposed a tool to predict the realizable performance upper bounds of sequential applications on heterogeneous HPC platforms, and the best hardware device for each compute kernels. We implemented AutoMatch in the LLVM compiler framework, and used different static and dynamic analysis techniques to quantify the application performance on different target architectures, including multi-core CPUs and many-core GPUs. The experimental results show that AutoMatch is highly accurate across five different heterogeneous architectures and a set of HPC benchmarks with different parallelism and memory access patterns. It achieves 96% prediction accuracy in identifying the best architecture for each compute kernel. In addition, the performance of its workload distribution scheme is comparable to an oracle based on run-time profiling of actual parallel code. Currently, AutoMatch is dedicated to shared-memory architectures, but can be extended to distributed-memory architectures by automatically constructing the communication models. Our technique is not restricted only to the performance criteria, but can also be extended to programmability and power efficiency.

REFERENCES

[1] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobbs journal*, vol. 30, no. 3, pp. 202–210, 2005.

[2] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[3] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," 2004, pp. 75–86.

[4] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, no. 9, pp. 1116–1127, Sep. 1988. [Online]. Available: http://doi.acm.org/10.1145/48529.48535

[5] K. Czechowski, C. Battaglino, C. McClanahan, A. Chandramowlishwaran, and R. Vuduc, "Balance principles for algorithm-architecture co-design," in *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, ser. HotPar'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2001252.2001261

[6] S. Xiao and W. c. Feng, "Inter-block gpu communication via fast barrier synchronization," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–12.

[7] W. c. Feng and S. Xiao, "To gpu synchronize or not gpu synchronize?" in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, May 2010, pp. 3801–3804.

[8] Y. J. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker, "Roofline model toolkit: A practical tool for architectural and program analysis," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 2014, pp. 129–148.

[9] L. McVoy and C. Staelin, "lmbench: portable tools for performance analysis," in *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*. USENIX Association, 1996, pp. 23–23.

[10] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 235–246.

[11] A. E. Helal, A. M. Bayoumi, and Y. Y. Hanafy, "Parallel circuit simulation using the direct method on a heterogeneous cloud," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[12] G. E. Blelloch, "Programming parallel algorithms," *Commun. ACM*, vol. 39, no. 3, pp. 85–97, Mar. 1996. [Online]. Available: http://doi.acm.org/10.1145/227234.227246

[13] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[14] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri, "Low depth cache-oblivious algorithms," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 189–199. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810519

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 44–54.

[16] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[17] R. F. Barrett, C. T. Vaughan, and M. A. Heroux, "Minighost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing."

[18] R. F. Barrett, S. D. Hammond, C. T. Vaughan, D. W. Doerfler, M. A. Heroux, J. P. Luitjens, and D. Roweth, "Navigating an evolutionary fast path to exascale," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, ser. SCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 355–365. [Online]. Available: http://dx.doi.org/10.1109/SC.Companion.2012.55

[19] E. S. Hertel, Jr., R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. Mcglaun, S. V. Petney, S. A. Silling, P. A. Taylor, and L. Yarrington, "Cth: A software family for multi-dimensional shock physics analysis," in *in Proceedings of the 19th International Symposium on Shock Waves, held at*, 1993, pp. 377–382.

[20] A. E. Helal, P. Sathre, and W.-c. Feng, "Metamorph: A library framework for interoperable kernels on multi- and many-core clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, 2016.

[21] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 152–163. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555775

[22] T. Hoefler, W. Gropp, W. Kramer, and M. Snir, "Performance modeling for systematic performance tuning," in *State of the Practice Reports*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 6:1–6:12. [Online]. Available: http://doi.acm.org/10.1145/2063348.2063356

[23] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[24] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, 2015.

[25] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 52.

[26] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[27] S. Lee, J. S. Meredith, and J. S. Vetter, "Compass: A framework for automated performance modeling and prediction," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 405–414. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751220

[28] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 725–737. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830780

[29] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor, "Kismet: Parallel speedup estimates for serial programs," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 519–536. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048108

[30] M. Kim, P. Kumar, H. Kim, and B. Brett, "Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, ser. IPDPS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1318–1329. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2012.128

[31] J. Shen, A. L. Varbanescu, Y. Lu, P. Zou, and H. Sips, "Workload partitioning for accelerating applications on heterogeneous platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2766–2780, Sept 2016.

[32] X. Zheng, L. K. John, and A. Gerstlauer, "Accurate phase-level cross-platform power and performance estimation," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 4:1–4:6. [Online]. Available: http://doi.acm.org/10.1145/2897937.2897977