AutoMatch: An Automated Framework for Relative Performance Estimation and Workload Distribution on Heterogeneous HPC Systems

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

***Abstract*— Porting sequential applications to heterogeneous HPC systems requires extensive software and hardware expertise to estimate the potential speedup and to efficiently use the available compute resources in such systems. To streamline this daunting process, researchers have proposed several "black-box" performance prediction approaches that rely on the performance of a training set of parallel applications. However, due to the lack of a diverse set of applications along with their optimized parallel implementations for each architecture type, the predicted speedup by these approaches is not the speedup upper-bound, and even worse it can be misleading, if the reference parallel implementations are not equally-optimized for every target architecture.**

This paper presents AutoMatch, an automated framework for matching of compute kernels to heterogeneous HPC architectures. AutoMatch uses hybrid (static and dynamic) analysis to find the best dependency-preserving parallel schedule of a given sequential code. The resulting operations schedule serves as a basis to construct a cost function of the optimized parallel execution of the sequential code on heterogeneous HPC nodes. Since such a cost function informs the user and runtime system about the relative execution cost across the different hardware devices within HPC nodes, AutoMatch enables efficient runtime workload distribution that simultaneously utilizes all the available devices in performance-proportional way. For a set of open-source HPC applications with different characteristics, AutoMatch turns out to be very effective, identifying the speedup upper-bound of sequential applications and how close the parallel implementation is to the best parallel performance across five different HPC architectures. Furthermore, AutoMatch's workload distribution scheme achieves approximately 90% of the performance of a profiling-driven oracle.

I. Introduction

With the end of Dennard scaling, the performance of sequential CPUs hit the power wall, thus making it hard to improve the performance by increasing the clock frequency [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. Instead, the parallel architecture landscape is in flux as new platforms are emerging to meet the needs of new workloads. Therefore, current (and future) HPC systems contain a wide variety of heterogeneous computing resources, ranging from general-purpose CPUs to specialized accelerators, due to both the diversity of the computational kernels in the applications and the lack of a single architecture meeting all of their requirements [2].

Porting sequential applications to heterogeneous HPC systems for achieving high performance requires extensive software and hardware expertise to *manually* analyze the target architectures and applications not only to estimate the potential speedup, but also to make efficient use of all the different compute resources in such systems. To streamline such a daunting task, end users need appropriate tools to automatically predict the potential application performance on HPC systems. Therefore, researchers have created several tools classified into two categories: automated performance modeling and machine-learning performance prediction.

Automated performance modeling tools [3], [4], [5] use static and/or dynamic analysis to construct a performance model of the target architecture and application and predict the potential speedup. However, these tools are either limited to traditional multi-core processors or require code annotations to indicate the available parallelism and data movement, which might not be possible for non-expert users. In addition, tools based only on static analysis do not work well for irregular applications, whose computation and memory access patterns are data-dependent, due to the difficulty of alias analysis [6], [7].

On the other hand, machine-learning performance prediction tools [8], [9] are heavily influenced by the training data. Thus, their prediction accuracy depends on the availability of a diverse set of applications along with their *optimized parallel* implementations for every target architecture, which is often hard to find [10]. Apart from that, the predicted speedup is not the speedup upper-bound, and even worse it depends on which optimization techniques are applied in the training applications. Unfortunately, open-source heterogeneous applications and benchmark suites are usually *not equally-optimized* for each architecture type [10], [11]. For this reason, machine-learning approaches may not be suitable to predict the relative performance between different architecture types despite the required expertise/effort to collect the ideal training set and the significant time for training. Given all this, there is a compelling need for a more practical approach to serve a wide range of the users of HPC systems.

A. AutoMatch: The First-Order Framework

This paper presents AutoMatch, an automated framework for matching of compute kernels to heterogeneous HPC architectures. Figure 1 shows the proposed framework which analyzes a given *sequential* application code to estimate the benefits of porting this application to heterogeneous systems.

First, AutoMatch generates the architectural specifications via micro-benchmarking to instantiate an abstract hardware model for each architecture in the target heterogeneous system. Second, it leverages compiler-based static and dynamic analysis

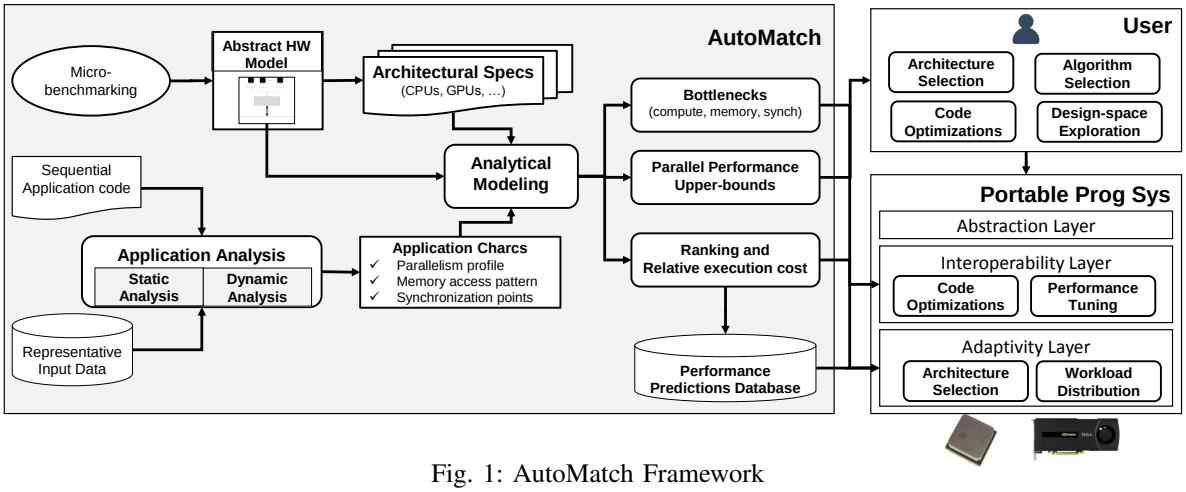

Fig. 1: AutoMatch Framework

techniques to quantify the maximum parallelism, the maximum data locality, and the minimum synchronization of the sequential code for estimating the *upper bounds* of the parallel performance on the different architectures. Third, AutoMatch generates *high-level* analytical models that combine the abstract hardware model, the application characteristics, and the architectural specifications to predict the performance on *different types* of hardware devices. This performance prediction is then used to estimate the relative execution cost across a set of different architectures including multi-core CPUs and many-core GPUs, thereby driving a workload distribution scheme, which enables end users to efficiently exploit the available heterogeneous devices in the HPC system.

It is important to note that AutoMatch is designed as a *first-order* framework for users to estimate the potential parallel performance of their sequential applications on heterogeneous HPC systems in the early stages of the development process, i.e., without having to pay the high cost of developing the optimized parallel code (or painfully collecting training data) for every target architecture. While our automatically-generated models are simple, they work well for predicting the relative performance across different architectures and the best workload distribution strategy.

1) Use Cases: AutoMatch accelerates the application development process and supports the emerging programming systems for performance portability and interoperability across different accelerators (such as MetaMorph [12], Kokkos [13] and RAJA [14]).

Architecture Selection. AutoMatch predicts the relative ranking and performance of heterogeneous architectures using sequential code. It serves not only those who either have not determined the target device or cannot afford to buy multiple candidate devices, especially when the application and the inputs are often changed, but also those who lack enough expertise to develop the optimized parallel implementation for each architecture type. Moreover, AutoMatch's architecture ranking enables the adaptivity layer of portable programming systems to select the best performing architecture at runtime.

Algorithm Selection. AutoMatch predicts upper bounds on the parallel performance of a given sequential application by finding the best dependency-preserving schedule of its operations, which performs the same operations as the sequential algorithm but in a different order. If AutoMatch's prediction of the original algorithm is already good enough, the user can save the time to consider other algorithms. Conversely, if the predicted performance is unsatisfactory, it motivates the user to explore/develop different algorithms. Nevertheless, AutoMatch can still play a critical role even for this case. The user can analyze the sequential code of different algorithms using AutoMatch to estimate their parallel upper bounds beforehand without developing the optimized parallel code(s).

Code Optimizations. AutoMatch provides detailed information about the inherent parallelism, data locality, and bottlenecks of the sequential code to help the user to decide on the best optimization and parallelization strategy for the target application. Moreover, since it is often hard to find reference applications and benchmarks that are *equally-optimized* for each architecture type [10], AutoMatch's prediction of the performance upper bounds serve as a reference to show how close the parallel implementation is to the best possible performance. That way AutoMatch can guide not only manual code optimization, but also customization/tuning of the different backends of portable programming systems.

Workload Distribution. AutoMatch's estimation of the relative performance (execution cost) on heterogeneous systems promotes the development of a run-time workload distribution on top of programming systems that support the seamless execution of parallel applications on multiple heterogeneous devices (e.g., MetaMorph [12]) to efficiently exploit the available compute resources across these devices.

Design-Space Exploration. Even if the applications and/or the architectures are not yet available, AutoMatch can still be used with synthetic application features and/or architectural parameters to automatically explore the design space. Detailed discussion of the architecture selection, code optimizations and workload distribution is provided in Section III. Due to the space limit, detailed study of algorithm selection and design-space exploration is reserved for future work.

B. Contributions

AutoMatch differs from the previous approaches in that it does *not* require the availability of the target platforms or the parallel application code for each platform, thereby expanding the range of users. In addition, it is automated and applicable to different types of hardware architectures with minimal effort, i.e., generating the architecture specifications. With the automated and repeatable methodology of AutoMatch, users can easily adapt it to future heterogeneous architectures. In summary, the following are the contributions of this work:

- We propose an automated framework (AutoMatch) that uses a combination of compiler analysis techniques, abstract hardware model, analytical modeling, and micro-benchmarking to estimate the performance upper bounds of sequential applications on heterogeneous HPC systems and the relative ranking and performance of the architecture alternatives in such systems (Section II).
- AutoMatch's estimation of the relative performance across the heterogeneous architectures enables a run-time workload distribution scheme that simultaneously utilizes them all in performance-proportional way, i.e., the architecture with higher performance is assigned more workload (Section II).
- Using a set of open-source HPC applications, with different parallelism profiles and memory-access patterns, we show the efficacy of the proposed white-box framework across different HPC architectures (Section III).
- We present case studies on both regular and irregular workloads to pinpoint the issues with black-box performance prediction approaches, e.g., profiling and machine-learning. Since they rely on the performance of a training set of parallel applications, unlike AutoMatch, their results can be fooled by heterogeneous implementations that are *not equally-optimized* for each target architecture (Section III).

II. The Proposed Framework

A. Hardware Architecture Model

This work proposes an abstract hardware architecture model that can be generalized to different shared-memory architectures including multi-core CPUs and many-core GPUs. This model extends the classical external memory model [15], [16] to parallel architectures and considers important constraints on such systems, such as the on-chip memory access time and the synchronization overhead.

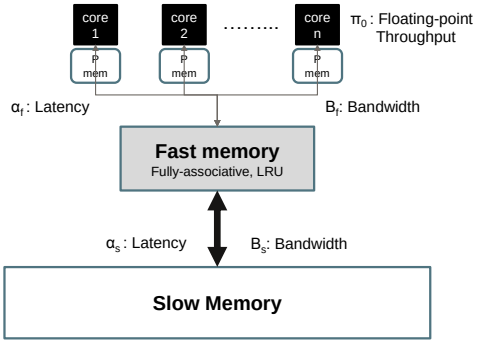

Fig. 2: The abstract hardware architecture

Figure 2 shows the proposed hardware architecture model comprised of multiple compute cores that share a fast on-chip memory connected to a slow off-chip memory. The compute cores can only perform operations on data in their private on-chip memory, and each core executes floating-point operations at a peak rate of π_0 FLOPs per second. The floating-point throughput is $\Pi = n_p \times \pi_0$, where n_p is the number of compute cores. The fast on-chip shared memory is fully associative 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 fast on-chip shared memory has a latency α_f and a bandwidth β_f, while the slow off-chip memory has a latency α_s and a 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 [17], [18], the proposed model considers this overhead. There are two synchronization types: (1) global synchronization, between coarse-grain threads with different control units (threads on CPUs and thread blocks on GPUs), and (2) local synchronization, between fine-grain threads with shared control units (SIMD lanes on CPUs and threads on GPUs). Given that local synchronization overhead is negligible in comparison to global synchronization (usually by at least an order of magnitude) [17], [18]. the proposed model ignores it.

Note, the main goal is to match the workloads to the best architecture from a set of parallel architectures that are fundamentally different, for which our proposed high-level hardware model works well. In light of this, the proposed model abstracts away architecture-specific parameters and low-level hardware details, e.g., hardware prefetchers and complex memory hierarchies. Similarly, it ignores one-time cost overheads, such as thread creation/destruction, kernel launching, and host-device data exchanges, which are highly-dependent on the run-time environment and the system/expansion bus rather than the target architectures.

B. Inferring the Architectural Specifications

AutoMatch figures out the specifications of the hardware architectures using micro-benchmarking. In particular, it uses ERT [19], pointer-chasing [20], [21], and synchronization [17], [22] 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, this work considers five architectures (two CPUs and three GPUs) with different core counts and considers three subsets of architectures: ($ARC1$, $ARC3$, $ARC5$), ($ARC1$, $ARC2$), and ($ARC4$, $ARC5$). The first subset contains three significantly different architectures with 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.

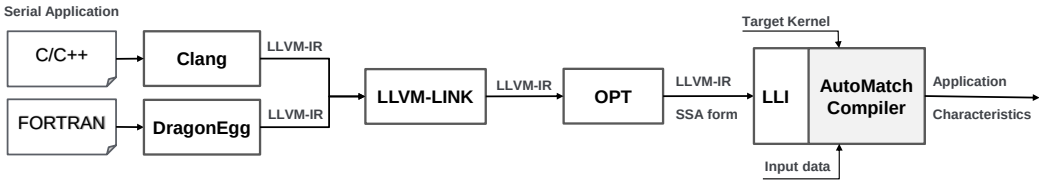

Fig. 3: The design and implementation of AutoMatch compiler

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
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

Since modern on-chip memories support the inclusion property in their hierarchy [23], AutoMatch chooses the fast memory size, Z, to be the effective on-chip memory capacity. On CPUs, Z is the last level 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 (e.g., GPU's local memory). 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. It turns out that the fast memory of the target architectures is better than the slow memory by approximately a factor of 15 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. While 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 [17]. Because 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.

C. Compiler-based Application Analysis

1) Design and Implementation: AutoMatch uses the LLVM compiler framework [24] and works on the LLVM intermediate representation (IR) of the sequential code, 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). Figure 3 shows the design and implementation of the AutoMatch compiler. 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 [25], other passes such as function inlining and constant propagation simplifies the induction variables and control flow and makes the analysis easier. In addition, the user provides the input data and the target kernel name. After that, the AutoMatch compiler, 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, which are combined with the specifications of the target heterogeneous system to generate the final performance analysis and predictions.

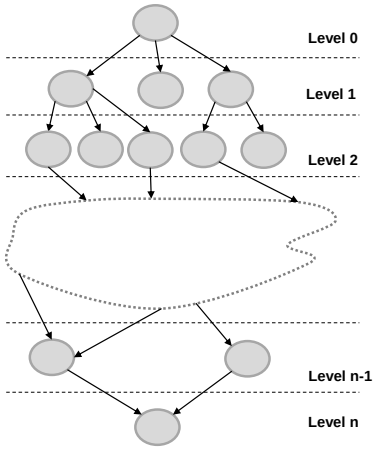

Fig. 4: The application ASAP schedule on a theoretical architecture with infinite resources

2) Parallelism Analysis: AutoMatch leverages both static and dynamic analysis techniques to automatically quantify the inherent parallelism in the sequential applications, thereby estimating their computation time on the different architectures for a given input data. In particular, it schedules the application on a theoretical architecture with an infinite number of registers and compute units and a zero memory access latency, such that each operation is executed as soon as its true dependencies are satisfied. Figure 4 depicts our 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. For example, if each dynamic instance m of a floating-point instruction I_n

is scheduled at an execution level j, then I_{nm} must have dependencies at the execution level $j-1$.

This ASAP schedule is similar in spirit to the classical work-depth model [26], [16], which represents the computations of a given algorithm using a directed acyclic graph (DAG) in which nodes and edges represent operations and their dependencies, respectively. While the classical work-depth model requires manual analysis to quantify the sequential part and average parallelism of a given algorithm, AutoMatch not only generates the ASAP schedule automatically to estimate the computation time, but also 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 constructs the def-use chains [25] at compile time to track data dependencies through registers; due to the infinite number of registers, only true dependencies exist. Second, it uses LLVM's dynamic compiler, LLI, to profile the application and collect the execution history of the compute instructions and memory operations. Third, using the application execution history, AutoMatch implements a dynamic points-to analysis [27], [28] to track data dependencies through the memory operations. Leveraging a hash table that resembles Content-Addressable Memory (CAM), AutoMatch dynamically detects true (read-after-write), anti (write-after-read) and output (write-after-write) memory dependencies.

Based on the detected true dependencies, AutoMatch constructs the ASAP schedule of the sequential application on the theoretical architecture and computes 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, AutoMatch computes f_{im}, the instruction mix of the sequential application to estimate 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 defined as:

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

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:

$$f_v = \frac{W_{vec}}{W} \quad (2)$$

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

3) Data Locality Analysis: AutoMatch quantifies the inherent data locality in the sequential applications by analyzing their memory access pattern on the above abstract architecture model, which assumes an ideal cache-memory model. The main goal is to estimate the number of data transfers between the compute cores and the fast memory, Q_f, and between the 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 [29]. The LRU stack distance (or reuse distance) is the number of distinct memory locations accessed between two consecutive accesses to the same memory location; 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: LRU stack distance analysis example

In a fully-associative cache with the LRU replacement policy, a memory reference with an LRU stack distance larger than the fast memory 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 than or equal to the fast memory size, respectively. While the LRU stack distance analysis ignores the conflict and contention misses, AutoMatch assumes that the memory transfers on the parallel architectures are bounded by Q_s and Q_f, as in prior work [30].

AutoMatch estimates the memory access cost of the target application as follows. First, it dynamically analyzes the LLVM IR instruction stream (execution history) to capture the load and store memory operations, and then records the referenced memory locations (addresses) along with their last access time in a self-adjusting binary search tree [31], which is sorted by the last access time. Second, whenever a memory location is referenced, AutoMatch examines the memory tree to find the last access time; 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 distinct nodes accessed between the last access to the target memory location and the current access; the number of such nodes is the LRU stack 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.

4) Synchronization Analysis: While the parallelization overheads consist of thread creation/destruction, kernel launching, and synchronization, AutoMatch focuses on synchronization for two reasons. First, unlike the other overheads, synchronization is not a one-time cost and can increase with the problem size (e.g., LUD has $O(n)$ synchronization points, where n is the matrix dimension). Second, the synchronization overhead is significant on massively-parallel GPU architectures; As shown in Table I, their overhead is an order of magnitude higher than multi-core CPUs.

AutoMatch uses a heuristic for estimating the required 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 loops, i.e., loops that cannot 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, i.e., the number of global synchronization points is $n-2$. In addition, AutoMatch allows users to annotate the source code to indicate the global synchronization points.

```
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

D. Analytical Modeling

1) Execution Cost Estimation: AutoMatch constructs the Execution Cost (EC) model that captures the complex interaction of the application, input data, and target architectures. In addition, it can be generalized to different types of hardware architectures. After analyzing the architecture-agnostic features of the sequential application, AutoMatch combines these features with the specifications of the target architectures to generate *first-order* analytical models to estimate the computation, memory access time and synchronization overhead.

The computation time T_{comp} is estimated as:

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

where D is the number of dependency levels, w_i is the total operations for each dependency level i, n_p is the number of cores, π_0 is the maximum operation throughput per core, f_v is the vectorization factor, and f_{im} is the instruction mix factor. This equation extends the classical Amdahl's law. The first term models the sequential execution, which depends on the inherent dependency chain, while the second term models the parallel execution that is limited by either the available cores or work in a given execution (dependency) level. In addition, it considers the effect of the instruction mix and vectorization on the computation throughput.

The memory access time T_{mem} is computed as follows:

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

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, Q_f is the number of data transfers between the compute cores and the fast shared memory, Q_s is number of data transfers between the fast and slow memories, D is the depth of the application ASAP schedule, and L is the memory transfer size. This equation accounts for the memory latency once per execution (dependency) level, and assumes that the memory transfers are effectively pipelined by the memory system such that they are limited by the memory bandwidth.

The synchronization time T_{syn}, is estimated as:

$$T_{syn} = S \times s_0 \quad (5)$$

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

Finally, 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. Moreover, AutoMatch predicts the parallel resource contention by considering the access time to shared resources such as the fast and slow memories (assuming that they are shared fairly among threads). Next, AutoMatch combines the execution cost on the different architecture with the floating-point work of the application to predict the *parallel* performance upper bounds on each architecture.

2) Workload Distribution: AutoMatch also estimates the relative execution cost across the different architectures to drive a workload distribution service for parallel compute kernels on heterogeneous CPU-GPU nodes. The main objective of this run-time service is to distribute the workload (i.e., iteration space and data) over the available heterogeneous devices to minimize the overall execution time. Instead of distributing the workload evenly across the CPU and GPU devices, AutoMatch reduces the overall execution time by considering the relative computing power of each architecture with the execution cost prediction above. For example, if AutoMatch's relative execution cost of a compute kernel on the CPU and the GPU is 3 to 1, its workload distribution scheme partitions the workload into four parts and assigns the three to the GPU and the other to the CPU.

III. Case Studies

This section shows our experiments to demonstrate the efficacy of AutoMatch and its utility as a *first-order* performance prediction framework for sequential applications on heterogeneous HPC systems.

The experiments use AutoMatch to analyze the *sequential* implementation of the target applications and show its estimation in comparison to the actual heterogeneous parallel implementations. The applications are built by the following compilers: gcc 4.9, icc 13.1, and nvcc 7.5, and AutoMatch is implemented in LLVM-3.6. While AutoMatch works with any data type supported by LLVM, this work considers double-precision floating point only for brevity. Since the key evaluation point is in the relative performance of the different HPC architectures at the chip level, the reported performance is for the core computation kernels and ignores one-time cost overheads such as I/O, data initialization (including host-device transfer), profiling, and debugging.

TABLE II: Rodinia and Parboil workloads

Workload	Description	Input data
CUTCP	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 3D grid	Grid 512x512x64
SPMV	Sparse matrix vector multiplication	Dubcova3.mtx [32]
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

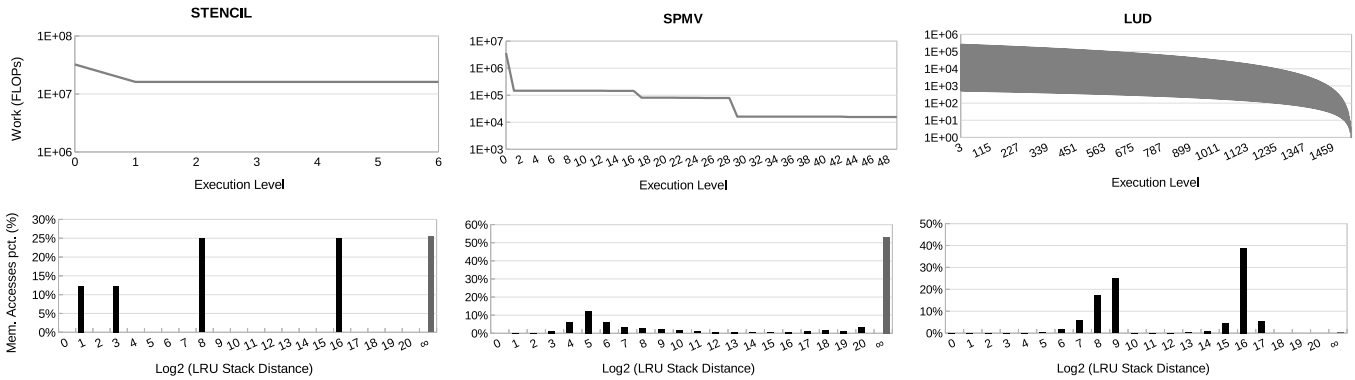

Fig. 7: Parallelism and LRU stack distance profiles

A. *Performance Forecasting and Analysis Case study*

This case study shows the effectiveness of AutoMatch in identifying the performance upper-bound of sequential applications on heterogeneous HPC architectures and how close the parallel implementation is to the best parallel performance. In addition, the study presents the sensitivity of AutoMatch to variations in the architectural characteristics and its ability to predict the relative ranking of the architecture alternatives. We consider eight HPC workloads from Rodinia [33] and Parboil [11] benchmarks with different parallelism profiles and memory access patterns. The reason for choosing Rodinia and Parboil is because they provide sequential/multithreaded CPU implementations and GPU implementations, which are used as reference for several black-box performance prediction approaches [8], [9]. Table II presents the target workloads and the input data sets provided by their benchmark suites.

Figure 7 presents the parallelism and LRU stack distance profiles of the target workloads. Due to space limitations, it shows only three workload results: STENCIL, SPMV and LUD. AutoMatch indicates that STENCIL is inherently parallel with a few execution levels and massive amounts of work per level, and it has a uniform memory access pattern with few memory streams corresponding to the dimensions of the data grid. SPMV has a small number of execution levels; however, the amount of work per level is significantly lower than STENCIL, due to the sparsity of the input matrices. In addition, SPMV suffers from low data locality, as the compulsory misses (memory references with LRU stack distance ∞) dominate the memory accesses. LUD has an irregular parallelism profile that alternates between two bounds corresponding to the computation of the pivot column and the update of the trailing sub-matrix, respectively. For LUD, the amount of work per execution level decreases as it moves 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 progresses due to the workload imbalance.

Figure 8 shows AutoMatch's estimation of the upper bounds on the parallel performance compared to the achieved performance of the OpenMP and CUDA implementations, while Figure 9 provides AutoMatch's analysis of the execution bottlenecks on the different architectures. The experiment considers the first subset of the target architectures ($ARC1$, $ARC3$ and $ARC5$) that contains heterogeneous architectures with significantly different hardware capabilities. The results show that AutoMatch accurately identifies the best architecture and the relative ranking of the different architectures in all the test cases. Moreover, the actual parallel implementations do not exceed AutoMatch's prediction, which indicates that AutoMatch accurately predicts the performance upper-bound.

Performance Gap. AutoMatch helps the user to determine how close the parallel implementations is to the performance upper bounds. The results show that the gap between the achieved performance and the estimated upper bounds on the many-core GPUs ($ARC3$ and $ARC5$) is small in most cases; however, this gap is quite large on the multi-core CPU ($ARC1$). In particular, the actual parallel implementations show that GPU architectures achieve more than two orders-of-magnitude speedup (up to 120X) over the CPU architecture, while AutoMatch reports lower relative speedup between the two architectures (up to an order-of-magnitude speedup). The important question here is *whether this performance gap is due to AutoMatch's prediction error or because the benchmark suites are not equally-optimized for each architecture type.*

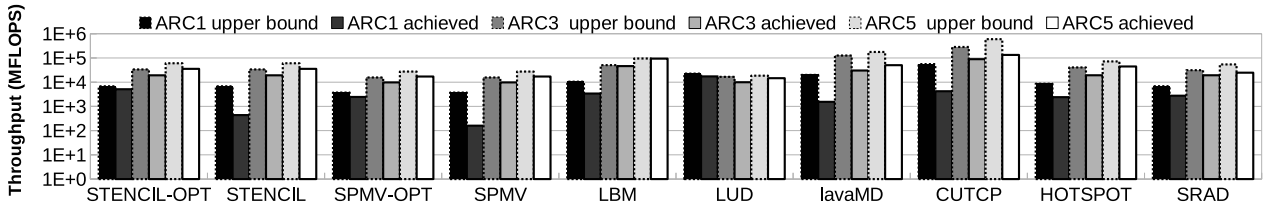

Fig. 8: Achieved performance vs. AutoMatch's upper bounds

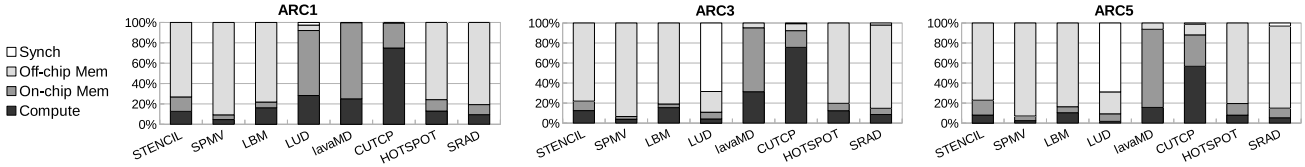

Fig. 9: AutoMatch bottlenecks prediction

As Lee et al. [10] debunk the unrealistic 100X speedup of GPUs vs. CPUs and show that it results from an unfair comparison with inferior CPU implementations, our hypothesis is that *the benchmark suites are not equally-optimized for each architecture type*. After inspecting their actual parallel implementations, it turns out that the CUDA implementation is optimized whereas the OpenMP implementation is unoptimized, which is justified by the following. First, the OpenMP implementation does not utilize the vector units, which reduces the performance of compute-bound workloads (e.g., IavaMD and CUTCP). Second, the OpenMP code is cache-unfriendly, e.g., it distributes loop iterations with unit-stride memory accesses on different threads (STENCIL) and uses array of structures (IavaMD, CUTCP and LBM). Simple data-layout optimization can dramatically improve the performance of the CPU caches [34]. Third, the CPU code incorrectly uses GPU-specific optimizations, e.g., irregular applications (SPMV) use the GPU-friendly compressed format (JDS format).

Optimization Studies. To verify our hypothesis and to show that the estimated upper bounds are attainable, the experiment here considers two cases: a regular workload (STENCIL) and an irregular workload (SPMV). First, AutoMatch indicates that STENCIL is bounded by the off-chip memory access time (Figure 9), and it has few memory access streams corresponding to the dimensions of the input data grid (Figure 7). 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. Second, AutoMatch shows that SPMV suffers from low data locality and has limited parallelism (Figure 7), which increases the load imbalance especially for architectures with massive number of threads. While the original OpenMP implementation uses the JDS format, which is more suitable for data-parallel architectures with fine-grain parallelism [35], we use the CSR format that outperforms JDS on coarse-grain parallel architectures with large caches. In addition, we used a dynamic workload distribution strategy that distributes chunks of 32 compressed rows over the available cores.

As shown in Figure 8, the performance of our implementations, named STENCIL-OPT and SPMV-OPT, are significantly better than the original implementations on $ARC1$, which means that the estimated performance upper bounds can be achieved with platform-specific optimizations and tuning. Moreover, while AutoMatch's prediction error of the relative speedup is *91%* on average for STENCIL and SPMV, it dramatically drops to *15.5%* on average for STENCIL-OPT and SPMV-OPT. These case studies pinpoint the critical issue with the "black-box" prediction approaches (e.g., profiling-driven and machine-learning). Since they rely on the performance of a training set of parallel applications, their results can be easily fooled by heterogeneous implementations that are *not equally-optimized* for each target architecture. In other words, their predicted relative speedup (and prediction accuracy) can be misleading without the availability of a diverse set of applications along with their optimized implementations for each architecture type; in general, finding such applications is another dauting task.

Finally, the gap between the the performance upper bounds and the achieved performance on many-core GPUs is relatively large in lavaMD and CUTCP, which are bounded by the compute time and on-chip memory access time according to AutoMatch's analysis. The investigation of the CUDA implementations of lavaMD and CUTCP shows that they suffer from low occupancy (37% and 27%), due to high registers and local memory usage which limits the number of concurrently active threads and thread-blocks. Kernel fission [36] can be used to improve the occupancy by partitioning the kernel into smaller kernels with less resources usage.

Sensitivity Analysis. The experiment here considers 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 (Table I).

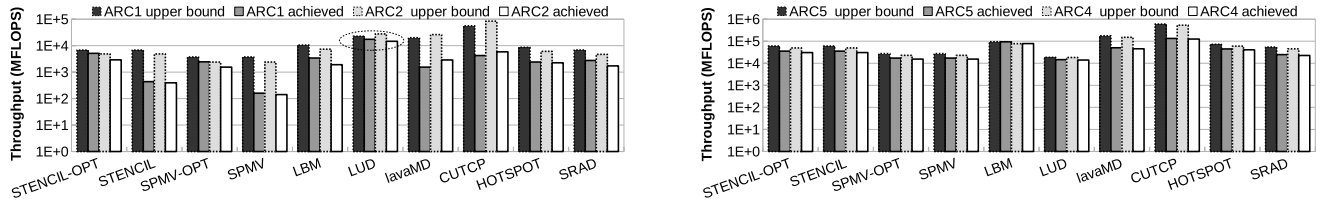

Fig. 10: AutoMatch prediction sensitivity

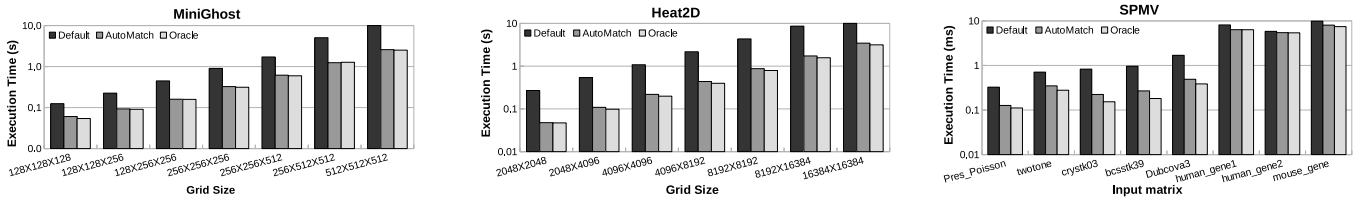

Fig. 11: Performance (compute kernels) on a heterogeneous CPU-GPU node ($ARC1$ & $ARC5$) with the different workload distribution strategies: Default, AutoMatch and Oracle.

Figure 10 shows AutoMatch's performance prediction and the actual performance on these architecture subsets. Surprisingly, AutoMatch accurately predicts the best architecture in all the test cases, except for the LUD benchmark on multi-core CPUs, which shows that our automatically-generated, high-level performance models are sensitive to the small variation of the target architectures. For LUD, 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 being accessed decreases as the execution progresses 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 LUD, leading to the incorrect ranking. While AutoMatch' high-level memory 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.

B. Workload Distribution Case Study

This study shows the effectiveness of our workload distribution scheme based on the execution cost model generated by AutoMatch from analyzing the sequential code. Our scheme is compared to an oracle distribution scheme obtained by runtime profiling of the optimized parallel implementations.

To evaluate AutoMatch's workload distribution, we use three applications from the structured grids and sparse linear algebra design patterns that are heavily-used in Computational Fluid Dynamics (CFD). MiniGhost [37] is a representative application for multi-material, hydrodynamics code that models hydrodynamic flow and dynamic deformation of solid materials. The main computation kernel is the finite difference solver, which applies a difference stencil and explicit time-stepping scheme on a homogenous 3D grid. Heat2D solves the poisson partial differential equations (PDEs) for heat diffusion in homogenous two-dimensional grid [38]. SPMV is a canonical sparse-matrix dense-vector multiplication. The experiment uses the implementations provided by the MetaMorph library [12], which supports the seamless execution of CFD applications on multiple heterogeneous devices, including CPUs (OpenMP back-end) and GPUs (CUDA back-end). In addition, MiniGhost and Heat2D are configured to apply a 3D 7-point and 2D 5-point stencils, respectively, 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. Three different workload distributions are tested: default, AutoMatch, and Oracle distribution. The default strategy is to distribute the workload evenly across the available devices. The AutoMatch workload distribution uses AutoMatch to analyze the sequential implementation and to predict the execution cost on the heterogeneous devices. Next, based on the predicted execution cost, it distributes the workload to minimize the overall execution time. The Oracle distribution is similar to AutoMatch strategy; however, instead of predicting the execution cost, it profiles the parallel code on the target CPU and GPU and distributes the workload based on the *measured* execution time.

Figure 11 shows the overall execution time of the target applications with the different workload distribution strategies. The results show that the AutoMatch and Oracle strategies achieve comparable performance and outperform the default strategy by a factor of *3.5X and 3.8X* on average, respectively. In summary, AutoMatch's workload distribution achieves approximately 90% of the oracle performance, due to its accurate estimation of the relative execution cost across CPU and GPU architectures. It would be interesting to investigate how the different workload distribution strategies affects the energy efficiency in that the CPU and the GPU have different power characteristics. However, it is beyond the scope of this paper and we leave it for our future work.

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

	COMPASS [5]	XAPP [8]	**AutoMatch**
Input code	Annotated	Sequential	Sequential
Features extraction	Static analysis	Dynamic analysis	Hybrid analysis
Arch model generation	By users	Training data	Benchmarking
Performance modeling	ASPEN model	Machine-learning	Exe. Cost model
Cache-aware	No	Yes	Yes
App generality	Low	High	High
HW generality	High	Low	High
The tool speed	Fast	Slow	Moderate

C. Caveats and Extensions

While AutoMatch generates simple and intuitive models, the results show that it works well as a *first-order* framework; however, it has several limitations. First, AutoMatch ignores one-time overheads such as host-device data transfers, which depend on the run-time system and the expansion bus rather than the devices, 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. In particular, the recent compiler algorithms for analyzing the value-flow chains of the program data can be used to estimate the communication cost between the host and devices (accelerators) [39]. Second, 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* prediction tool, to incorporate more sophisticated models (e.g., [40]), 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 performance parameters.

IV. Related Work

Recently, several tools have been proposed to automate the performance modeling and prediction using static/dynamic analysis and machine-learning. Table III summarizes the comparison of the recent performance prediction tools for heterogeneous HPC architectures (CPUs and GPUs).

COMPASS [5] generates a structured performance model from the parallel application code using static analysis. However, the user must indicate the available parallelism and data movement to generate an accurate model. Otherwise, COMPASS may generate a conservative parallelism profile, due to the difficulty of alias analyses [6], [7]. Therefore, COMPASS does not work well for irregular applications whose computation and memory access patterns are data-dependent.

XAPP [8] uses machine-learning to find the correlation between the CPU execution profile of the application and the GPU speedup. 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 huge effort to rewrite and re-optimize each training application to the target architectures. Moreover, to predict the performance on a specific GPU device, the user needs to run all the training applications on this device, which takes days. Such long-running model generation of ML-based tools end up being orders of magnitude slower than AutoMatch, which generates the device parameters using micro-benchmarks that takes few minutes. In addition, XAPP's predicted speedup is not the speedup upper-bound, and it depends on which optimization techniques are applied in the training applications.

Kismet [3] predicts the potential 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 application on a CPU cache simulator. 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. As an alternative, Parallel Prophet [4] predicts the speedup of the annotated code on multi-core CPUs. Unlike Kismet, it does not require parallelism discovery, but relies on user annotations to identify the available parallelism. To build the performance model, Parallel Prophet collects architectural parameters such as instruction counts and cache misses through hardware performance counters, which requires the availability of the target CPUs and the parallel (or annotated) code.

Shen et al. [41] 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 easily extended to model the host-device data transfer overhead.

V. Conclusion

This paper proposes AutoMatch, an automated framework that combines compiler-based analysis techniques, an abstract hardware model, analytical modeling, and micro-benchmarking to project (1) the realizable performance upper bounds of sequential applications on heterogeneous parallel architectures, (2) the relative ranking/performance of the architecture alternatives, and (3) the best workload distribution strategy on heterogeneous nodes with different parallel devices. The experimental results show the efficacy of the proposed framework across five different heterogeneous architectures and a set of HPC workloads, with different parallelism and memory access patterns. Moreover, AutoMatch's workload distribution turns out to be very effective, achieving comparable performance to a profiling-driven oracle.

Currently, AutoMatch is dedicated to shared-memory parallel architectures, but can be extended to non-uniform and distributed-memory architectures by automatically constructing the communication models [39]. Finally, AutoMatch is not applicable only to the performance criteria, but can also be extended to programmability and power efficiency.

ACKNOWLEDGMENTS

The authors would like to thank Mark Gardner, Vignesh Adhinarayanan, and Paul Sathre of the Synergy Lab at Virginia Tech for helpful discussions. We also thank the reviewers for their constructive comments and feedback. This work was supported in part by the Air Force Office of Scientific Research (AFOSR) Computational Mathematics Program via Grant No. FA9550-12-1-0442, NSF I/UCRC IIP-1266245 via the NSF Center for High-Performance Reconfigurable Computing (CHREC), and the Synergistic Environments for Experimental Computing (SEEC) Center via the Institute for Critical Technology and Applied Science (ICTAS), an institute dedicated to transformative, interdisciplinary research for a sustainable future.

REFERENCES

[1] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," 2005.

[2] K. Bergman, S. Borkar, D. Campbell *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[3] 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, 2011, pp. 519–536.

[4] 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, 2012, pp. 1318–1329.

[5] 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*, 2015.

[6] V. Kuncak and M. Rinard, "Existential heap abstraction entailment is undecidable," in *Proceedings of the 10th International Conference on Static Analysis*, ser. SAS'03, 2003, pp. 418–438.

[7] G. Ramalingam, "The undecidability of aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1467–1471, Sep. 1994.

[8] 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, 2015, pp. 725–737.

[9] I. Baldini, S. J. Fink, and E. Altman, "Predicting gpu performance from cpu runs using machine learning," in *Proceedings of IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, ser. SBAC-PAD'14, 2014, pp. 254–261.

[10] V. W. Lee, C. Kim, J. Chhugani *et al.*, "Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, 2010, pp. 451–460.

[11] J. A. Stratton, C. Rodrigues, I.-J. Sung *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[12] 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.

[13] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[14] R. Hornung, J. Keasler *et al.*, "The raja portability layer: overview and status," 2014.

[15] 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.

[16] 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, 2011, pp. 9–9.

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

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

[19] Y. J. Lo, S. Williams, B. Van Straalen *et al.*, "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*, 2014, pp. 129–148.

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

[21] 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*, 2010, pp. 235–246.

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

[23] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture, A Hardware/Software Approach*. Morgan Kaufmann, 1996.

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

[25] S. Muchnick, *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[26] G. E. Blelloch, "Programming parallel algorithms," *Commun. ACM*, vol. 39, no. 3, pp. 85–97, Mar. 1996.

[27] M. Mock, M. Das, C. Chambers, and S. J. Eggers, "Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '01, 2001, pp. 66–72.

[28] M. Kim, H. Kim, and C.-K. Luk, "Sd3: A scalable approach to dynamic data-dependence profiling," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[29] 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.

[30] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri, "Low depth cache-oblivious algorithms," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10, 2010.

[31] R. A. Sugumar and S. G. Abraham, "Efficient simulation of caches under optimal replacement with applications to miss characterization," in *In Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, 1993.

[32] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[33] S. Che, M. Boyer, J. Meng *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009, pp. 44–54.

[34] T. M. Chilimbi, "Cache-conscious data structures: design and implementation," Ph.D. dissertation, 1999.

[35] Y. Saad, "Krylov subspace methods on supercomputers," *SIAM Journal on Scientific and Statistical Computing*, vol. 10, no. 6, 1989.

[36] H. Wu, G. Diamos, J. Wang *et al.*, "Optimizing data warehousing applications for gpus using kernel fusion/fission," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, 2012.

[37] 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," Tech. Rep., 2011.

[38] N. Ozisik, *Finite difference methods in heat transfer*. CRC press, 1994.

[39] A. E. Helal, W.-c. Feng, C. Jung, and Y. Y. Hanafy, "Commanalyzer: Automated estimation of communication cost on hpc clusters using sequential code," Tech. Rep., 2017.

[40] 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, 2009, pp. 152–163.

[41] 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, 2016.