DySel: Lightweight Dynamic Selection for Kernel-based Data-parallel Programming Model

Li-Wen Chang Hee-Seok Kim Wen-mei Hwu
University of Illinois
{lchang20, kim868, w-hwu}@illinois.edu

Abstract

The rising pressure for simultaneously improving performance and reducing power is driving more diversity into all aspects of computing devices. An algorithm that is well-matched to the target hardware can run multiple times faster and more energy efficiently than one that is not. The problem is complicated by the fact that a program's input also affects the appropriate choice of algorithm. As a result, software developers have been faced with the challenge of determining the appropriate algorithm for each potential combination of target device and data. This paper presents DySel, a novel runtime system for automating such determination for kernel-based data parallel programming models such as OpenCL, CUDA, OpenACC, and C++AMP. These programming models cover many applications that demand high performance in mobile, cloud and high-performance computing. DySel systematically deploys candidate kernels on a small portion of the actual data to determine which achieves the best performance for the hardware-data combination. The test-deployment, referred to as micro-profiling, contributes to the final execution result and incurs less than 8% of overhead in the worst observed case when compared to an oracle. We show four major use cases where DySel provides significantly more consistent performance without tedious effort from the developer.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Runtime environments

Keywords Graphics Processing Unit, Dynamic Profiling

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA.

DOI: http://dx.doi.org/10.1145/2872362.2872373

1. Introduction

The demand for computing devices with increased performance at reduced power continues to grow. In the mobile community, such devices enable more functionality and longer battery life. In the high-performance computing community, such devices make exascale computing more feasible. As improvements from the semiconductor fabrication process diminish, architects are compelled to introduce more diversity into all aspects of computing devices: function units, interconnect fabrics, and memory hierarchies. The complicated interactions between these diverse devices and input data present a major challenge in predicting the effect of optimizations performed by the programmers or compilers.

Programmers of these devices must understand and exploit a wider set of architectural entities to achieve high performance, far beyond the traditional instruction set architecture and uniform memory space. CUDA and OpenCL, for instance, are designed for highly parallel execution based on lightweight threads, but high performance often requires carefully crafted work assignment to threads, multi-level tiling, and scheduling of the threads. While achieving high-performance on one device is challenging, preparing code that can achieve high-performance across a diverse set of devices is a daunting and tedious task for even the most skilled programmers.

It is therefore desirable to support performance portability over different device architectures. Popular approaches commonly involve compiler and/or runtime solutions to address this problem. Architecture-specific code transformations must be done by considering detailed facts about the target architecture. Achieving high performance for a particular device and runtime data combination involves finding a good selection and ordering of optimizations along with appropriate values for associated parameters.

Figure 1 demonstrates how different choices of optimizations can result in substantially disparate results of the state-of-the-art Intel OpenCL stack [21] on an Intel i7-3820 CPU. It compares the performance of heuristically selected optimization [13] of `sgemm` and `spmv` (denoted as `spmv-jds`) in Parboil [28] against that of a non-vectorized version and two

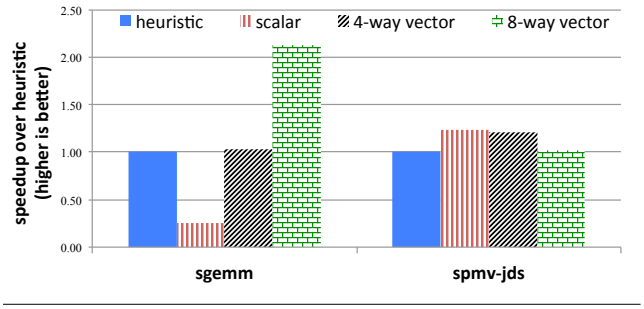

Figure 1. Performance of Intel CPU OpenCL stack with different vectorization strategies.

alternative vectorized versions. The Intel compiler counter-intuitively chooses 4-way vector for regular and control divergence free `sgemm` kernel, while it uses 8-way vector for `spmv` kernel which exercises control divergence. Under control divergence, code generation for work-items using SIMD instructions typically comes with a large overhead due to masking, packing and unpacking, which gets larger with wider SIMD datapath width. In the end, the Intel's heuristic has made suboptimal decisions for both cases, falling short of the best achieved performance by a factor of 2.13x and 1.24x, respectively. One observation is that it clearly demonstrates the importance of choosing the most optimal code. Another observation is any single static heuristic for choosing optimizations will likely fall short due to the complexity of interactions between the device, the computation, and the data.

There is a wealth of prior work [2, 7, 11, 12, 14, 15, 17, 21–23, 27, 35] on estimating the performance of code or determining the effect of a certain type of compile time code transformation. Performance models are widely used to prune the design space for autotuning [11, 22], or to guide optimization strategies [7, 15, 17, 22, 35]. For example, locality-centric scheduling [17] statically analyzes memory access patterns with respect to work-items and kernel loops and employs a heuristic cache model to determine a locality-friendly schedule of OpenCL work-item loops on CPUs. PORPLE [7] relies on GPU memory or cache models to analyze work-item access patterns of regular applications for data placement. However, these approaches address model-specific aspects of the device architecture of interest, while other important factors such as vectorization are not considered or assumed to be decoupled from the aspects being considered. Such assumptions considerably reduces the accuracy of the model-based approach. Moreover, they are limited by ignoring factors that are only known at runtime, such as the actual data shape. As a result, accurately predicting the effect of optimizations is not likely viable at compile time. Optimizations based on inaccurate predictions can lead to disappointing performance outcomes.

Several runtime-based approaches [7, 10, 18, 24, 33] have been proposed to mitigate the problem with static performance prediction approaches. Reactive tiling [24] uses online trained tiling model and chooses likely optimal tiling parameters for the given working set size and system load. PORPLE [7] leverages runtime micro simulation on a CPU to refine the GPU memory or cache models when inputs are irregular and cannot be statically analyzed. Although more information is accessible at runtime, model-driven approaches at runtime can still have limitations and blind spots of unconsidered factors of models like static model-driven approaches, resulting in suboptimal decisions.

In this paper, we propose DySel, a runtime framework that matches the best code arrangement with the actual device and data combination, thereby improving performance and efficiency. With DySel, we remove the burden of determining the most optimal code from an optimizing compiler and allow it to produce several likely candidate variants from the input code. Then the runtime performs *micro-profiling*, a process of deploying the candidates on a small portion of the actual data on the actual device and determines the best version to be used to process the rest of workload. The advantage of this approach is that it can work with virtually any combination of compiler and device architecture as modeling is not required at runtime. When dealing with a large workload, the cost of micro-profiling can be easily amortized and the overall benefit can be much greater than unconditionally executing a single code selected by the compiler. Note DySel is not intended to completely replace whole decision-making procedures for compiler optimizations with run-time profiling. Instead, DySel is designed to relax the conditions for determining the most optimal code from an optimizing compiler. Therefore, with DySel, the optimizing compiler still needs to produce several (typically 2 to 10) likely candidate variants.

We have prototyped our idea on OpenCL and CUDA, two popular kernel-based parallel programming models today. Experimental results demonstrate that our proposed approach correctly chooses the optimal code version with less than 8% overhead in the worst observed case compared to oracle results for both CPU and GPU architectures.

We make the following contributions in this paper:

1. We propose a combined compiler and runtime approach towards optimal code selection.
2. We propose a lightweight runtime micro-profiling technique and discuss the engineering tradeoffs involved in its implementation.
3. We evaluate the DySel idea by implementing a prototype system and performing real-hardware measurements for four important use cases on CPU and GPU.

The rest of this paper is organized as follows. Section 2 introduces the concept of DySel. Section 3 details our implementation. Section 4 and 5 evaluates and discusses the performance of DySel. Section 6 outlines related work and Section 7 concludes.

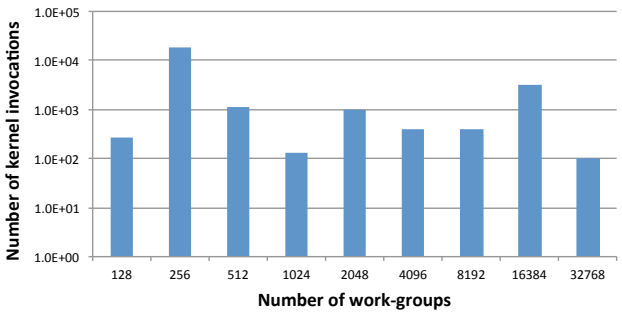

Figure 2. Distribution of number of work-groups among kernel launches in Parboil and Rodinia benchmarks

2. DySel Design

This section provides a background of profiling for kernel-based data-parallel programming, and introduces the idea of *DySel* and the three productive profiling techniques used in DySel.

2.1 Profiling for Kernel-based Data-parallel Programming

DySel evaluates different code variants at runtime in order to determine which variant performs best. It measures performance of each variant on a small portion of the actual data and identifies the best performing one, a process which we call *micro-profiling*. The chosen version will be used to process the rest of workload. The key ingredients of DySel are efficient profiling and accurate performance projection.

Popular kernel-based data-parallel programming models, such as OpenCL, CUDA, OpenACC, and C++AMP, allow over-decomposition of workloads for maximized parallelism. Work-groups in OpenCL, for example, are meant to run **independently** from each other by design and this enables efficient parallel execution on a variety of architectures, such as CPUs and GPUs. With these programming models, workload processing is done via repeatedly executing the kernel code over a small subset of the workload, which often takes place in parallel.

The decomposition makes the number of independent kernel invocations fairly **large** in practice, which helps to amortize the cost of allocating a few of them for evaluation of code variants. The overhead for evaluating code variants can often be amortized over the large number of invocations.

Figure 2 shows accumulated occurrences of kernel invocation in different numbers of work-groups from all OpenCL benchmarks in Parboil [28] and Rodinia [6] benchmark suites. It supports the low-cost profiling hypothesis based on workload decomposition, as significant number of kernel invocations fall into the range of 128 to 32768 work-groups. Kernel invocations with less than 128 work-groups are rarely observed and so dropped from the figure. Kernel launches with small number of work-groups can be sensitive to profiling overhead, but a small number of work-groups also indicates relatively small workloads and performance variation from the level of optimization might not be critical. DySel mainly targets kernels with large work-groups and profiling-based kernel selection is deactivated for small workload.

An individual kernel invocation is assumed to have **similar performance** throughout subsequent invocations. This is due to the nature of data-parallel computing where the same code is used to process large data. Thus, observed performance from a kernel invocation is likely to be indicative for others, which helps keep the required sampling frequency and thus the overhead low. These properties make work-group an ideal granularity for micro-profiling.

Accurate performance projection means that the variant that performs best in micro-profiling is the one that performs best on the whole workload, even in irregular workload or complex variants. Accuracy of DySel can be evaluated by measuring performance difference between the variants on the whole workload, and is further discussed in Section 4.

2.2 Productive Profiling

DySel employs *productive* micro-profiling, where execution from profiling also contributes to workload processing. Each kernel invocation during profiling takes a different part of the workload data. This is a departure from offline profiling, where performance characteristic is extracted while the result is simply ignored. This strategy reduces the overhead of profiling, since workload processed during profiling does not require reprocessing.

Figure 3 shows the three productive profiling techniques used in DySel. In this example, we assume that the compiler produces two implementations as follows. The ratio of workload per work-group between Version A and B is 3 to 2. According to safe point analysis [24] (Section 3.4), DySel launches 2 and 3 work-groups for Version A and B, respectively, in order to make a fair throughput comparison during profiling.

Fully-productive profiling, shown in Figure 3(a), is the most efficient profiling in DySel. Each kernel invocation during profiling takes a different part of the workload data and computes valid contribution to the final output. In Figure 3(a), both versions compute and profile different parts of the workload, and write to the final output. After profiling, Version B is chosen to compute the remaining workload, as Version B turns out to run faster during profiling. Fully-productive profiling can take place as long as the individual invocations do not have overlap in the final output, which is a dominating pattern in data-parallel programming. It is accurate when parts of the workload are roughly equal, which is common in regular applications.

Two variants of productive profiling, called partial-productive profiling, are proposed to overcome the limitations of fully-productive profiling. Hybrid-based partial-productive profiling, shown in Figure 3(b), is designed for irregular workload with a non-overlapping final output. By running a set of ker-

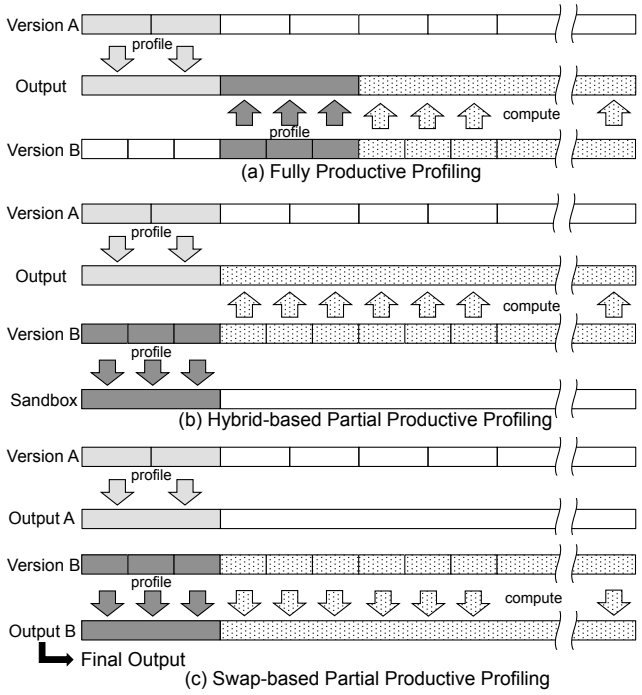

Figure 3. Illustration of proposed productive profiling

nels over the same workload, profiling can be still fair among different kernel innovations. While only the first kernel (Version A in Figure 3(b)) can contribute to the final output, results from others (Version B in Figure 3(b)) are directed into a sandbox and thus unused. After profiling, Version B is chosen to process the remaining workload. Since the final output contains contributions from both Version A and B, we call this profiling hybrid-based.

Swap-based partial-productive profiling, shown in Figure 3(c), is proposed for handling output range variations among work-groups from different kernels by running a set of kernels over the same workload but with their own private output spaces. After profiling, the one selected kernel and output (Version B and Output B in this Figure 3(c)) will remain for the rest of execution while others (Version A and Output A in this Figure 3(c)) are discarded. Since the final output is swapped with Output B, we call this profiling swap-based. It is worth mentioning that swap-based partial-productive profiling can be considered as a speculation approach to version selection.

DySel relies on compilers or programmers to specify which kind of productive profiling a set of kernels. Interaction between DySel and compilers and programmers is discussed with more details in Section 3.

2.3 Applicability

The choice of profiling mode is determined based on programming patterns and optimizations for kernels. First, fully-productive profiling is appropriate for kernels with regular or near-regular workloads. Large workload variations can significantly impact the fairness of comparison among kernels. Therefore, fully-productive profiling is only suitable for applications with regular workload, such as BLAS, or stencil. It can select between kernels with different levels of optimizations such as tiling, thread coarsening, data layout transformation (including padding), input binning [26, 29], loop-interchange, locality-centric scheduling [17], vectorization, software prefetching, data placement [7, 15], and input format transformation [4]. Some optimizations require special treatment during profiling. Tiling and thread coarsening require normalization of throughput using safe point analysis [24] (Section 3.4) to ensure fairness. Data layout transformation, input binning, and input format transformation may require duplication of inputs to meet the assumption of different kernel implementations.

Second, hybrid-based partial-productive profiling supports all patterns and optimizations supported by fully-productive profiling. Additionally, it is applicable to irregular workload. By profiling the same portion of workload across different kernels, unfair throughput comparison can be avoided. The applicable kernels typically have in-kernel loops with varying bounds across work-groups, such as sparse BLAS. Uniform workload analysis [14] (Section 3.4) can be used to detect such in-kernel loops with varying bounds.

Finally, swap-based partial-productive profiling further supports output range variation across kernels, and in theory is applicable to any optimizations, such as privatization, regularization, compaction, output binning, scatter-to-gather [26, 29], kernel fusion, kernel fission, optimizations using atomic operations, and even algorithm change. Although swap-based partial-productive profiling is the most applicable profiling in DySel, it has less output contribution efficiency than fully-productive profiling. It also requires more storage than the hybrid-based profiling due to the need to maintain private output spaces during micro-profiling.

2.4 Orchestration for Profiling and Execution

The way DySel orchestrates micro-profiling and execution at runtime can have significant impact on profiling overhead. We present two orchestration designs in this subsection.

Figure 4(a) shows the overall flow of synchronous DySel. The compiler deposits several code versions to the kernel pool in the executable binary file. Upon execution of the kernel, the runtime dispatches code versions from the pool and executes them (❷) in one of the productive modes described in the previous section, with a few work-groups (❶) assigned using safe point analysis [24] (Section 3.4). The runtime waits until all versions finish execution and compares their execution time to pick the best one. Then the rest of the execution runs with the selected kernel. The implementation is simple. However, it incurs latency penalty if there is large disparity between the best and the worst versions since the latency of the profiling phase is determined by the slowest execution (❸). Furthermore, the total number

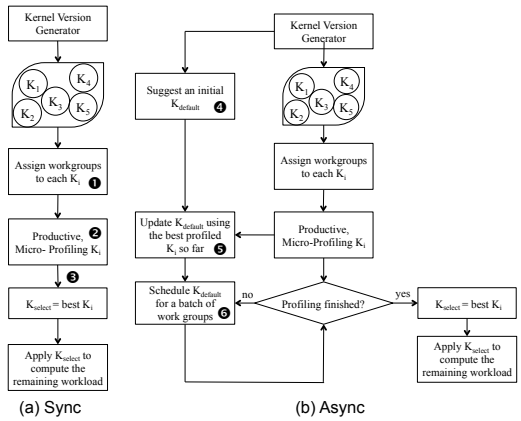

Figure 4. Flows of proposed synchronous and asynchronous DySel

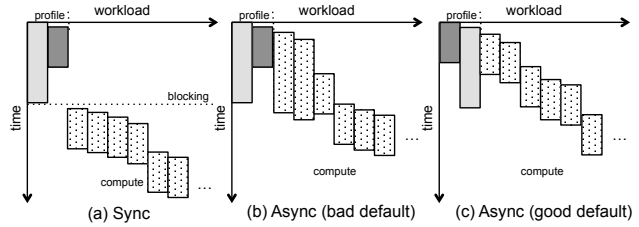

Figure 5. Illustration of timing difference between synchronous and asynchronous DySel

of work-groups used in micro-profiling may not be able to fully utilize the hardware resources.

Figure 4(b) shows the flow for asynchronous DySel. Unlike the synchronous design, the rest of the execution can begin even before profiling is complete, which we denote as eager execution. When the non-profiling execution begins, it runs with the best-performing version so far. To support eager execution, we require that the compiler or programmer suggest an initial version (❹). We will discuss more on the initial selection and its impact on performance later. As profiling progresses, the selection gets updated once a faster version is found (❺). The asynchronous flow must be able to switch to the best kernel found so far, therefore the eager execution is done via launching a series of chunks (❻), instead of a single batch. While it can better tolerate the latency of profiling, the implementation gets more complicated for two reasons. First, it requires careful workload management so that profiling can be done with a higher priority over the eager execution. Second, the eager execution is divided into many chunks, imposing associated kernel launch overhead.

Figure 5 illustrates the execution timing for both synchronous and asynchronous micro-profiling designs. The example assumes that there are four concurrent execution units and two kernels in the kernel pool. The darker gray kernel runs faster than the lighter one. With synchronous flow, the runtime waits for all kernels to finish profiling execution. It underutilizes the execution units while waiting for the slow kernel to complete its profiling execution. The asynchronous flow overcomes this problem by launching useful work on the vacant execution units with the initial version, which is shown Figure 5(b) and (c). However, the quality of the initial selection potentially can impact overall performance, as suboptimal code occupies execution units longer, as shown in Figure 5(b). In either case, the asynchronous flow yields better utilization and throughput compared to synchronous one.

Table 1. Summary of proposed productive profiling

Profiling method	Productive output in profiling	Extra space requirement	Async support
Fully-productive profiling	K	0	Yes
Hybrid-based partial-productive profiling	1	$\leq K-1$	Yes
Swap-based partial-productive profiling	1	$\leq K$	No

Table 1 summarizes throughput, extra space requirement, and support of asynchronous flows for the three proposed productive micro-profiling modes. Given K kernel variants in the kernel pool, all K profiled portions of the workload contribute to the final workload in fully-productive profiling, while only 1 profiled portion does so in the two partial-productive profiling modes. In terms of extra space requirement, fully-productive profiling directly writes results into the original output space and needs no extra space, while the two partial-productive profiling methods require at most $K-1$ or K copies of space for either sandboxes or private outputs, respectively. It is worth mentioning that extra space requirement can be further reduced if footprint of memory accesses during profiling can be predicted so that a subset of output is allocated for sandboxing. Last, both fully-productive profiling and hybrid-based partial-productive profiling support asynchronous flows, since profiling results are directly written into the distinct, final output space, while swap-based partial-productive profiling cannot support asynchronous flows, because the final output space is not determined until profiling is complete.

3. Implementation

In this section, we first introduce the DySel runtime interface, and then detailed implementations for both CPU and GPU. Lastly, we discuss the analyses to determine the ratios of workload among kernel variants, and the profiling mode.

3.1 Runtime Interface

Unlike traditional runtimes, DySel allows the compilers or programmers to deposit multiple implementations of the same kernel function signature. Figure 6(a) shows the DySel kernel implementation registration API. A DySel-specific

```
DySelAddKernel(
   string kernel_sig,               // kernel name
   func_ptr implementation,         // kernel implementation
   dim3 wa_factor,                  // work assignment factor
   vector<int> sandbox_index= []    // argument offsets for
                                    // sandboxes/private outputs
);
```

(a) Kernel Implementation Registration API

```
DySelLaunchKernel(
   string kernel_sig,      // kernel name
   bool profiling=true,    // profiling activation flag
   enum mode=fully_async   // profiling mode
 );
```

(b) Kernel Launch API

Figure 6. DySel runtime interface

requirement is to provide *work assignment factor*, which is the number of workload units packed into each work-group for accurate profiling. Figure 6(b) shows the DySel kernel launch API. It allows the caller to specify whether profiling is activated or not using a *profiling activation flag* along with *profiling mode*. Compiler analyses required are discussed in Section 3.4.

Work Assignment Factor. An important class of optimizations is designed to utilize resources such as registers or scratchpad memory for improved execution efficiency. Among them, coarsening [19] and tiling change the amount of work assigned to each thread and thus the work assignment per kernel invocation. The runtime needs to know the relative work assignment between variant kernels for fair comparison. Once such workload-changing optimization is done, the compiler needs to inform the runtime about the change. For those kernel variants generated by programmers, DySel requires programmers to specify the relative work assignment ratios.

Profiling Activation Flag. A class of applications, such as stencil operations in partial differential equation (PDE) solvers or sparse matrix-vector multiplication (spmv) in conjugate gradient (CG) iterative solvers, launch a kernel iteratively without changing workload or data shape between iterations. In this scenario, the kernel can be just profiled in the first iteration and the selected variant can be reused for the later iterations. The profiling activation flag allows the user to turn on profiling only for the first iteration.

Profiling Mode. As mentioned previously, different modes of profiling have different applicability, throughput, and cost. Different classes of optimizations require different productive profiling modes for efficient, fair profiling. Asynchronous profiling potentially can reduce overheads of profiling. The DySel interface allows the compilers and programmers to indicate their choices.

3.2 CPU Runtime Implementation

Work distribution and prioritized execution for profiling are two main requirements for the CPU implementation of DySel. We use Intel's TBB [20] that has strong support for both. TBB's work stealing feature provides load balancing over multiple cores while its concurrent task groups allow assigning higher scheduling priority to profiling execution. In the profiling task group, kernel invocation is wrapped by timer calls to measure execution time. Updating the current best variant is done via atomic operation when the execution time of a variant is found to be smaller than the current minimum. Non-profiling task group invokes the current best implementation upon launch. When profiling is activated, it first launches the profiling task group with priority, which is followed by launching the non-profiling task group. Synchronous mode puts a barrier to wait for the profiling task group to finish its execution between the two task group launches, while asynchronous mode schedules both task groups concurrently. When profiling is not activated, it launches the non-profiling task group only.

3.3 GPU Runtime Implementation

Measuring execution time with precision for a kernel with small workload on GPUs requires a high-resolution timer. It is often inaccurate to use wall clock timer or GPU driver event according to our experience. Alternatively, we use in-kernel performance counters to solve this problem. The in-kernel performance counter is CUDA-specific, thus DySel first translates OpenCL code to CUDA code when performing profiling on NVIDIA GPUs. On the other hand, performance comparison takes place on the host. Therefore, the GPU runtime for DySel entails a combined solution where kernel execution timing information is measured on the GPU and used by the DySel runtime code running on the host.

Kernel Code Transformations. Each kernel variant is further replicated into three extended versions: one for profiling, one for non-profiling batch execution, and one for eager execution in the asynchronous profiling mode. First the code for profiling is augmented by inserting performance measurement code, shown in Figure 7(b). It reads the CUDA clock register, as shown in Figure 7(a), at the beginning and the end of the code, and the difference is recorded to a dedicated memory location. The code will be executed over multiple thread blocks because GPU throughput is determined by both latency and parallelism. Atomic operations are used to mark the earliest and the latest cycles among the participating thread blocks. The profiling code is also augmented with a block index offset parameter to shift thread block id so that the kernel will be profiled based on a particular part of the workload. Second, the code for non-profiling batch execution has an additional parameter that indicates the number of thread blocks executed during profiling and a test that suppresses the execution of the thread blocks that have been executed during profiling. This code will also be used when profiling is not activated by supplying a 0 value for the parameter. Finally, the code for eager dispatch in asynchronous profiling mode has the same block id shifting code as the profiling code but does not have measurement code. This code is exclusively used in the asynchronous profiling mode.

```
__device__ unsigned int get_cycle() {
  unsigned int y;
  asm("{\n\t" // use braces for local scope
      " mov.u32 %0, %clock;\n\t"
      "}"
      : "=r"(y) );
  return y;
}
```

(a) CUDA in-kernel clock register

```
unsigned int local_start_stamp;
if((threadIdx.x)==0) {       //only one thread in a  thread block
   local_start_stamp = get_cycle();
   unsigned int old = atomicMin(global_start_stamp + kernel_id, local_start_stamp);
   local_start_stamp= old > local_start_stamp? local_start_stamp: old;
}

//Here is kernel code, between the two measurement codes

__syncthreads();
if((threadIdx.x)==0) {   //only one thread in a  thread block
   unsigned int old = atomicInc(global_count + kernel_id, gridDim.x);
   if(old ==  gridDim.x-1) {  //only last thread block of profiling
      unsigned int local_diff = get_cycle()-local_start_stamp;
      old = atomicMin(global_diff, local_diff );
      if(global_diff <old )
         atomicExch(global_final_selection,kernel_id);
   }
}
```

(b) GPU profiling code

Figure 7. Example of GPU profiling code for a kernel using 1D thread block

Host Code Generation. DySel uses CUDA streams to launch different kernels concurrently on a GPU. By using multiple concurrent streams, profiling of different kernels can be done in parallel, reducing the profiling overhead. In the synchronous mode, `cudaDeviceSynchronize` is called to wait for all streams to finish profiling. The host code then chooses the best one, followed by launching the code for non-profiling batch execution. In the asynchronous mode, DySel calls `cudaStreamQuery` to check the status of each stream for profiling and schedules eager dispatches. After all profiling streams complete, DySel launches the remaining computation in a batch.

3.4 Compiler Analyses

Delivering accurate information on profiling mode and work assignment factor to DySel is critical for both correctness and efficiency. The information can be specified by programmers. However, such an approach is not desirable because inconsistent information can be supplied by the programmer, which may result in unexpected behavior. Compiler-driven approach can supply safe information automatically, though it can be conservative. We outline three important analyses for a compiler to derive such information:

Safe Point Analysis. Relative work assignment among kernel variants can be normalized to the least common multiple (LCM) among all related assignments for a fair comparison. Therefore, the number of work-groups in each variant of a kernel can be defined as the LCM divided by each work assignment factor. For more details, we refer to [24]. In DySel, we further multiply the number returned from safe point analysis by a constant to make the total workload become a multiple of the number of CPU cores or GPU streaming multiprocessors (SMs) in order to fully utilize the hardware.

Uniform Workload Analysis. A large disparity of loop iteration counts among profiling execution may lead to unfair performance comparison. We employ uniform workload analysis to determine whether loop bounds are varying across work-groups. Uniform workload analysis also detects an early break of a loop or an early termination of a kernel. For more details, we refer to [14]. The result of uniform workload analysis determines which profiling mode is applied in DySel. It is worth noting that uniform workload analysis may be conservative for kernels with data-dependent loop bounds. For example, in spmv on a CSR matrix, the given matrix might have an uniform non-zeros per row, but our analysis will flag it as a non-uniform workload since the loop bound is data-dependent. DySel interface provides programmers with an option to select a specific profiling mode by overriding the compiler's decision.

Side Effect Analysis. Whether work-groups have overlapping and variable output ranges is critical for DySel to safely conduct productive profiling. We employ side effect analysis to identify whether output overlapping happens. Here, we either assume no data race in the original OpenCL source code [16] or that the original OpenCL programs are deterministic. Therefore, the current implementation of side effect analysis only detects global atomic operations. Upon detection of global atomic operations, the compiler restricts the micro-profiling to swap-based to ensure correctness. Similar to uniform workload analysis, side effect analysis is used to determine profiling mode, and is also conservative, since atomic operations do not directly imply that memory contention really happen across work-groups. Therefore, DySel also allows the programmers to override the compiler decision.

4. Evaluation

In this section, we evaluate our prototype DySel implementation on several OpenCL/CUDA benchmarks on both CPU and GPU architectures.

4.1 Setup

All experiments are done on a system with an Intel Core i7-3820 CPU with an NVIDIA K20c GPU connected via

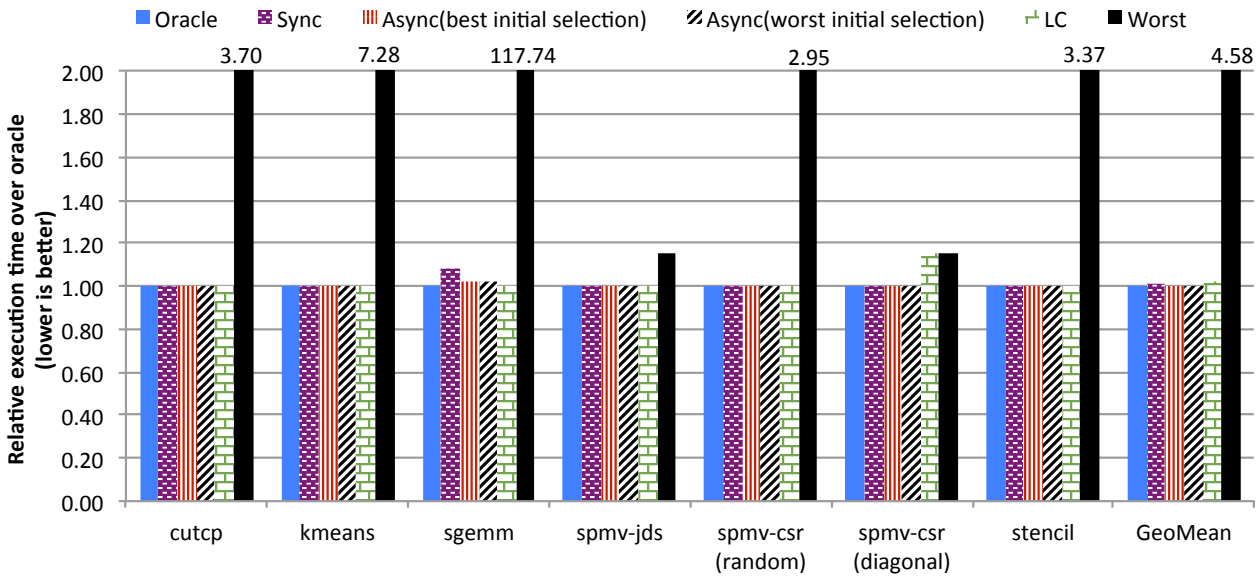

Figure 8. DySel performance results on locality-centric scheduling on CPU for OpenCL

PCI-E interface. For CPU experiments, our DySel starts with OpenCL code, performs all the code transformations described in this paper, produces C source code with calls to the TBB API, and uses the Intel's icc compiler version 15.0.2 with vectorization enabled to generate final executable code. For GPU experiments, DySel starts with OpenCL code, performs the code transformations described in this paper, produces CUDA source code, and uses nvcc version 7.0.27 with -O3 and –arch=sm_35 compile flag to generate executable. The CUDA driver version in use is 346.82. The system runs on 64-bit Ubuntu Server 14.04 LTS.

The experiments are based on benchmarks from Parboil [28], Rodinia [6] and SHOC [9] benchmark suites. Different experiments subscribe to slightly different sets of benchmarks according to their own purposes, which are individually described for each.

The evaluation focuses on the impact of DySel on kernel execution time. To this end, we measure execution time that includes all profiling time, profiling kernel launch overheads, and compute time of the remaining workload for both CPU and GPU architectures. For GPUs, time spent on data transfer among architectures before and after kernel execution is not included since it is the same across variants and profiling modes, but time spent on data transfer required by DySel profiling and variant selection is included.

The oracle results in the following is defined as the single *pure* versions that delivers the shortest runtime for the corresponding benchmarks. Note a *mixed* version that applies different pure versions on different partitions of computation could potentially outperform the "oracle". In this paper, we only compare DySel with the best pure version. For the mixed version, we consider it as the future work.

4.2 Case Study I & II: Single Compile-time Optimization

Performance modeling at compile time can yield the highest accuracy when it is used in isolation from other transformations. We compare DySel's performance to two high-quality compile time optimizations for CPU and GPU in order to compare accuracy and benefit. For CPU, locality-centric scheduling of OpenCL work-items execution [17] is used. For GPU, data placement [7, 15] is chosen.

The purpose of this experiment is to test whether DySel can adaptively and efficiently select the right variant from candidate kernel implementations. When a single standalone optimizer works well, DySel can confirm the version with an ignorable overhead. On the other hand, when the static optimizer makes a wrong decision, DySel can insure that an alternative is chosen to avoid performance loss.

Locality-Centric Scheduling on CPU for OpenCL. We evaluate DySel on the locality-centric scheduling of work-item executions for CPUs using four benchmarks, `sgemm`, `spmv` (denoted as `spmv-jds`), `stencil` and `cutcp`, from Parboil [28], one benchmark, `kmeans`, from Rodinia [6], and one benchmark, `spmv` with scalar dot products on a CSR format matrix without padding (denoted as `spmv-csr`), from SHOC [9]. All of the chosen versions in Parboil are base versions. These benchmarks are selected because their CPU performance is sensitive to the scheduling policy used. The inputs for Parboil and Rodinia benchmarks are all default, while the inputs for `spmv-csr` include a $16k$-by-$16k$ random sparse matrix with 1% probability of non-zeros (the default input in SHOC, denoted as `random`) and a $2M$-by-$2M$ diagonal matrix (denoted as `diagonal`). In `spmv-jds` and `spmv-csr`, DySel applies hybrid-based partial-productive profiling due to irregular workload, while

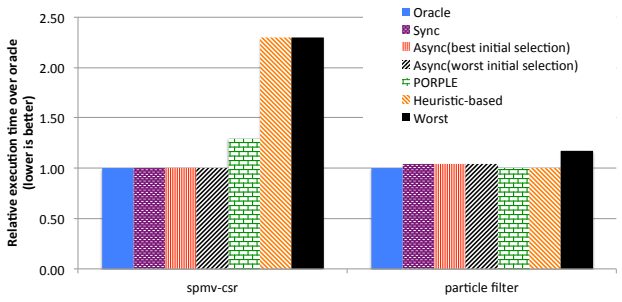

Figure 9. DySel performance results on data placement on GPU

in the rest, DySel applies fully-productive profiling. Since `kmeans`, `spmv-jds`, `spmv-csr` and `stencil` are iterative, DySel profiles only the first iteration.

We use the OpenCL implementation of Kim et al. [17], which is denoted as LC, to generate code from possible combinations in scheduling of work-items execution. LC employs a heuristic that minimizes overall memory access strides in work-items execution in the presence of loops. Possible numbers of different schedules are 60, 3, 6, 2, 2 and 6 for `cutcp`, `kmeans`, `sgemm`, `spmv`, `spmv-csr` and `stencil`, respectively, as schedules are permutation of work-item loops and kernel loops to form loop nesting.

Figure 8 compares performance of DySel and compile-time heuristic over oracle. DySel achieves close to optimal results for all benchmarks with negligible overhead. The results also demonstrate the importance of a data-locality-friendly schedule as the gap between the oracle and the worst is significant. A heuristic-based static selection could have caused a large performance loss with a suboptimal decision. However, DySel correctly selects the optimal schedule for all given benchmarks. In the case of `spmv-csr` with the diagonal input, the LC static heuristic selected incorrectly but DySel avoided the mistake. When no or little performance variation due to input distribution is expected, DySel chooses the optimal from profiling, witnessed by `cutcp`, `kmeans`, `sgemm`, and `stencil`. In a situation where input distribution has a high impact to data locality, such as `spmv-csr`, the static approach works well with a certain input distribution but it cannot cope with all possible cases with equal efficiency. DySel, on the other hand, adaptively chooses between two schedules, yielding close to optimal performance for both cases. Additionally, it also shows that asynchronous mode hides profiling latency better than synchronous mode. The impact of initial version selection in asynchronous mode is marginal in all cases. More discussion about the impact of synchronous mode and the initial version selection in asynchronous mode is shown in Section 5.

Data Placement on GPU. We evaluate DySel on the data placement optimization on GPU using two benchmarks, `spmv-csr` from SHOC [9] and particle filter from Rodinia [6]. For `spmv-csr`, we choose three data placement policies listed in the PORPLE paper [7] for the recent three generations of GPUs, and one data placement policy from a heuristic-based method [15] as the candidates. For particle filter, we choose two data placement policies from the PORPLE paper, one from the heuristic-based method, and one originally from Rodinia as the candidates. The input for `spmv-csr` is the random matrix, which is also used in micro simulation of PORPLE, and the input size for particle filter is 32,000. Both benchmarks use hybrid-based partial-productive profiling due to irregular workload, and `spmv-csr` is profiled for only the first iteration.

Figure 9 demonstrates that DySel can efficiently choose the optimal data placement policies for both benchmarks on a Kepler K20c GPU. On `spmv-csr`, DySel successfully selects the optimal version with a completely negligible overhead, while both PORPLE and the heuristic-based method generate suboptimal versions, falling short of the best achievable performance by a factor of 1.29x and 2.29x, respectively. Interestingly, the heuristic-based method yields the worst performance. Also, the optimal data placement for `spmv-csr` on Kepler is actually generated by PORPLE but with the target on Fermi architectures. On particle filter, DySel successfully selects the optimal version with at most 4% overhead, while both PORPLE and the heuristic-based method generate the optimal version. The original policy from Rodinia delivers the worst performance of a 1.17x slowdown compared to the best performance.

In this evaluation, both benchmarks expose different levels of challenge for static data placement due to irregular workload. PORPLE tends to perform much better than the heuristic-based method, because PORPLE uses more complicated memory and cache models. However, due to lack of runtime information, PORPLE might still make a wrong decision. With DySel, the performance loss from these mistake can be minimized.

4.3 Case Study III: Mixed Compile-time Optimizations

We evaluate multiple mixed and complex optimizations using four Parboil benchmarks: `cutcp`, `sgemm`, `spmv-jds`, and `stencil`. We choose all versions listed in these Parboil benchmarks as the candidates. There are two candidates for `sgemm`, three for `stencil`, four for `spmv-jds` on GPU, two for `spmv-jds` on CPU, and two for `cutcp`. All inputs are the default large datasets in Parboil. The profiling modes are the same as the previous experiment.

The purpose of this experiment is to test DySel's adaptive selection capability when static heuristics cannot easily foresee the impact of combined optimizations for decision making. The relevant optimizations for `cutcp`, `sgemm` and `stencil` are tiling, coarsening, data placement, loop unrolling and prefetching. Particularly, `cutcp` has tiling, coarsening, and data placement using scratchpad; `sgemm` has tiling, coarsening, loop unrolling, and data placement using scratchpad; `stencil` has tiling, coarsening, and data

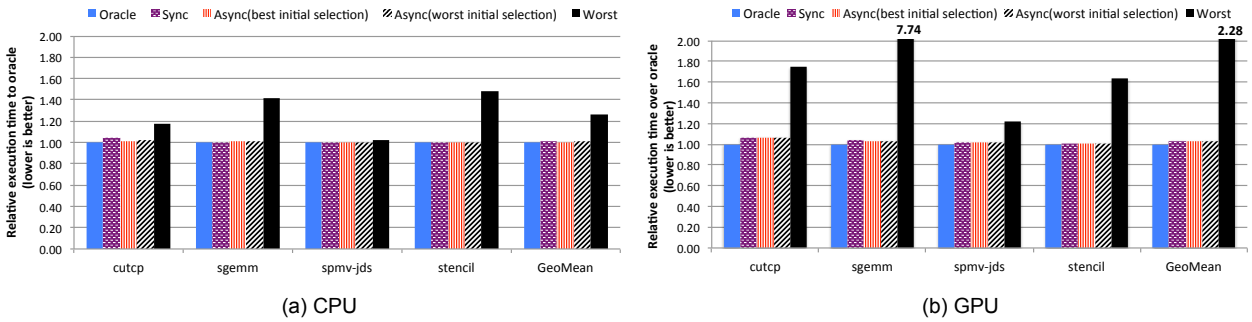

Figure 10. DySel performance results on mixed compile-time optimizations

placement using scratchpad; `spmv-jds` has tiling, coarsening, loop unrolling, prefetching, and data placement using texture memory, which is only for GPUs. Work assignment factors are 4x and 16x for `cutcp` and `sgemm`, respectively, while `stencil` comes with three versions and work assignment factors relative to the base version are 64x and 128x.

CPU. Figure 10(a) compares the performance of DySel when differently optimized kernels are provided. Again, LC from the OpenCL implementation of Kim et al. [17] is used to compile the kernels. DySel achieves near optimal results for all benchmarks, with only 2% overhead on average compared to oracle. While not shown in Figure 10(a), we note that for all benchmarks, the best variant for CPU execution are naive base versions from the selected benchmark suites, as they allow the greatest flexibility for the compiler in planning how to serialize execution of work-items. GPU-specific optimizations such as data placement and data prefetching using scoreboarding make no difference for CPU. Tiling using scratchpad memory typically leads to negative results on CPUs because there is no latency gain using them after they are lowered to CPU's uniform memory space, resulting in a 1.23x slowdown on average compared to oracle. This result again confirms that the help from runtime can mitigate the stress having to develop complicated models at compile time.

GPU. Figure 10(b) compares the performance of DySel when differently optimized kernels are provided for a Kepler K20c GPU. DySel always selects optimal version among candidates, except `spmv-jds`. In `spmv-jds`, DySel chooses the second best version (which applies loop unrolling, prefetching, and texture memory), which has only 0.8% performance degradation from the best one (texture memory only). We observe that optimizations of loop unrolling and prefetching applied in `spmv-jds` on a Kepler architecture are redundant when texture memory is applied, but they do improve performance when texture memory is not used. The similar situation is also observed in `stencil`, where scratchpad tiling and coarsening along the x-dimension does not improve performance on top of coarsening along the z-dimension on Kepler. These mismatched optimizations may come from a mistake of the experts who write the benchmarks or from non-transferable optimizations across different generations of GPUs. In both cases, it confirms performance prediction for the combined set of optimizations is challenging, and DySel can help resolve the challenge.

4.4 Case Study IV: Input-dependent Optimization

We evaluate an input-dependent version selection scenario using `spmv-csr` from SHOC benchmark suite [9]. We choose two `spmv-csr` versions, one using scalar dot product (denoted as `scalar`) and the other using vector dot product (denoted as `vector`), from SHOC. The optimal version of `spmv` on a CSR-format matrix is highly dependent on sparsity of a matrix [4], which is typically unknown at compile time. We test with two matrices, the random sparse matrix and the diagonal matrix, described in the previous experimental setup. The profiling modes are also the same as the previous experiment.

The purpose of this experiment is to demonstrate DySel's adaptive selection capability when the compiler simply cannot predict the performance, due to lack of critical information, which is sparsity of the actual matrix in this experiment.

CPU. Figure 11(a) shows performance of DySel compared to that of a scalar kernel and a vector kernel for all possible combinations of work-item scheduling for them. DySel correctly selects the optimal one for both inputs, yielding near oracle performance. The selection here is particularly complicated by the dimensions of schedule, kernel version and input data distribution. Here, LC chooses to iterate in-kernel loops first (denoted as `DFO`) for both scalar and vector implementations and uses it unconditionally. However, it does not cope well with unfavorable input distribution from the diagonal matrix, where work-item-loop-first (denoted as `BFO`) schedule is desired. As for version selection among scalar and vector, scalar performs better when `DFO` is chosen mainly because of less overhead having to deal with control divergence. For CPU execution, the vector version is inferior because it uses local memory which incurs the copy cost without any benefit in a CPU. This is a reason why the `BFO` schedule for the vector kernel performs similar to `DFO` in spite of favorable data locality.

GPU. Figure 11(b) demonstrates that DySel can adaptively select the optimal implementations for different matrices on a Kepler GPU. On the random matrix, DySel se-

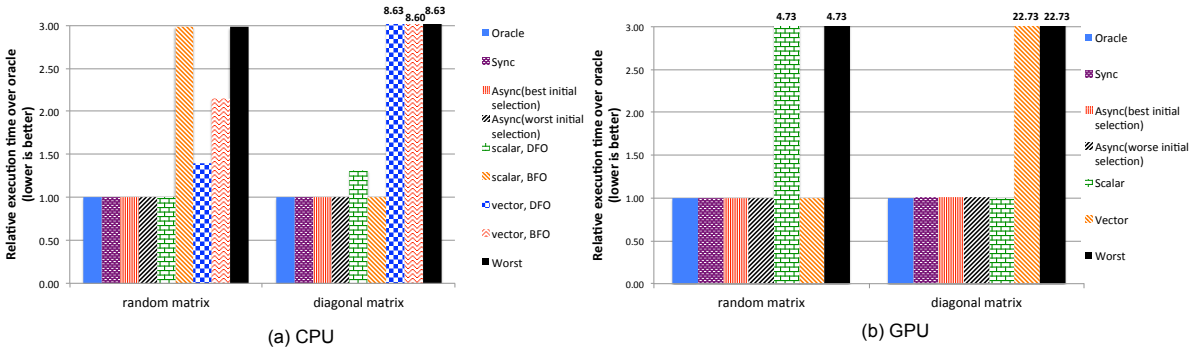

Figure 11. DySel performance results on input-dependent optimization

lects the vector version with at most 0.4% overhead, while the scalar version delivers a 4.73x showdown, due to non-coalescing memory accesses. On the other hand, for the diagonal matrix, DySel selects the scalar version with at most 0.8% overhead, while the vector version delivers a significant 22.73x showdown, due to underutilization of vectors (warps on NVIDIA GPUs).

5. Discussion

5.1 Performance Impact between Sync and Async DySel

Long-running profiling execution that lasts to the end of the profiling phase increases profiling overhead, which is further amplified by the number of vacant execution units waiting for workload. For example, `sgemm` with data locality scheduling shown in Figure 8 is regarded as a pathological case due to its huge variation of 117x between the oracle and the worst. The synchronous profiling overhead is 8%. Nevertheless, asynchronous mode can scatter the overhead, suppressing it down to less than 5%.

The asynchronous DySel GPU implementation is based on CUDA concurrent execution using CUDA streams, in which different kernel invocations are enqueued by the host CPU. It requires the host to query the status of the streams. At the same time, DySel employs micro-profiling, which typically takes extremely small profiling time. Combining these two facts, querying the status often takes a longer latency than profiling time. Therefore, it can only have few or even zero eager dispatches. In the end, synchronous and asynchronous DySel have only marginal difference for GPUs.

One possible solution to further improve asynchronous DySel is to apply CUDA Dynamic Parallelism to avoid the long latency from host queries. However, current CUDA Dynamic Parallelism also suffers from a huge kernel launching overhead on the device [32, 34], which is even larger than the original DySel overhead.

5.2 Profiling Overhead

Among all experimental benchmarks, DySel exposes 8% overhead in the worst case compared to the oracle results. However, since a few of the experimental benchmarks are iterative, their overhead is amortized across iterations and cannot be easily quantized in the above figures.

Here, we further investigate those benchmarks, which are `spmv-jds`, `stencil`, `spmv-csr`, and `kmeans`, by enabling profiling in every iteration. In this experiment, we observe increased overhead.

For CPU, most of the above benchmarks show less than 6% overhead, except `spmv-csr` for the random matrix with observed overhead of 88% to the oracle. Performance profiling using Intel's vtune reveals that the TBB dispatch loop incurs substantial overhead from spin and associated cost, which might have been caused by managing huge number of extremely tiny tasks. A tradeoff is to implement bigger task granularity by serializing several kernel invocations at the expense of load imbalance. Also, profiling accuracy can be a problem when the unit of workload is small, which is the case where system noise can affect the evaluation, particularly for CPUs. In `spmv-csr`, for example, the dynamic selection accuracy is 95%. By increasing the number of executions per kernel during profiling, this can be resolved at the expense of additional profiling overhead. The exact tradeoff and an empirical solution are to be further studied.

Similarly, on GPUs, `spmv-csr` in Case study II has 14.7% overhead; `spmv-jds` in Case study III has 46.5%, while stencil has only 3.7% overhead; `spmv-csr` in Case study IV has 9.9% to 30.5% overhead for different input datasets. We recognize most of the high overhead is from the `spmv` benchmarks (`spmv-csr` and `spmv-jds`). Unlike the other benchmarks, each iteration of `spmv` spends several to several hundred microseconds in non-profiling execution on the GPU, while profiling time takes around a few to 10s of microseconds, which is very close to the GPU kernel launch overhead. In this sense, profiling overheads are completely exposed despite the small portions of the workload DySel profiles. Fortunately, most of these data-parallel kernels with insignificant execution time are either iterative or not performance critical. Also, this overhead should be much less for larger sparse matrices in real applications.

5.3 Performance Advantage over Heuristic Approaches

As mentioned previously, quality and precision of heuristics dictate performance. In this subsection, we highlight performance insurance that DySel provides against decisions made by state-of-the-art heuristic implementations.

From Case study I & II, DySel can provide 1.15x more performance in `spmv-csr` when the diagonal matrix is given to the CPU (in Figure 8). PORPLE [7] and the heuristic-based method [15] making suboptimal data placement policies for GPU, DySel achieves 1.29x and 2.29x improvements over the model-based counterparts (in Figure 9).

In Case study III & IV, it is difficult to judge which version the compiler should generate since the kernels are written by experts [9, 28]. Assuming suboptimal decisions are made, DySel can outperform 1.26x and 2.28x on average for CPU and GPU, respectively (in Figure 10). In Case study IV (in Figure 11), the amount of performance improvement is input-dependent. DySel recovers 2.98x and 8.63x speedups over the worst possible choice for CPU for the random and the diagonal input matrices, respectively. Similarly, it achieves 4.73x and 22.73x speedups on GPU for the same input matrices.

6. Related Work

Various works have covered performance modeling for overall kernel performance or specific optimizations of data-parallel programming.

Dynamic optimization has been studied extensively for performance, which exploits information only available at runtime. Dynamo [3] and Mojo [8] monitor execution traces to capture a sequence of frequently executed basic blocks and replace the trace with a highly optimized instruction sequence. However, their performance suffers in many cases due to high overhead of execution monitoring and runtime compilation. Similarly, modern managed languages typically are supported by runtime optimization [1] for instruction throughput. While aforementioned approaches are focused on finding a better instruction sequence at fine-grained scope such as basic blocks, our work differs as multiple differently implemented kernels are used, which brings a greater impact to performance with low cost profiling overhead. Reactive tiling [24] alters tiling size on-the-fly for working set size optimization, which can also be properly implemented using our approach. ADAPT [31] sharing a similar idea with our work, however, the optimization rules are not performance portable to other architectures. Unlike this work, our approach does not require coupling with a runtime optimizer, making it easier to deploy.

Performance models for GPUs have been studied to provide performance feedbacks for optimization benefits [2, 23]. Several compile-time optimizations for data-parallel programming based on **heuristics** or **modeling** have been done on CPUs [12, 14, 17, 21, 27] and GPUs [7, 11, 15, 22, 35]. DySel relieves the pressure for developing an optimal and costly version as it allows them to generate multiple versions.

Adaptive runtime using **profiling** has been an area of great interest. Charm++ [30] employs a message-passing-driven adaptive runtime for matching an execution unit for a given task at a node level by continuously monitoring the performance over different types of execution units. Unlike Charm++, DySel applies runtime profiling for selecting the optimal versions of kernels on a CPU or GPU. SASSI [25] provided by NVIDIA can deliver flexible **software profiling** for GPU architectures. DySel applies software profiling instructions on CPU and GPU architectures for selecting the optimal versions at runtime.

Model-driven adaptive runtime has been widely studied for both CPUs and GPUs. Reactive tiling [24] switches between kernels of different tile sizes at runtime, from a decision made by subscribing **online** profiling results into a model-based (through curve-fitting) heuristic. Dollinger et al. [10] proposed a model-based adaptive runtime selection for GPUs using offline profiling. Elastic computing [33] dynamically selects the optimal version from multiple implementations of a function based on a cost model built at install time via profiling them. PEPPHER [5] provides offline profiling [18] functionality to help select variants of components and construct a heterogeneous program. In contrast, DySel provides a lightweight **online productive** profiling for multiple variants selection on data-parallel programming without specific models involved.

7. Conclusion

We have discussed and showed the drawback of solely relying on heuristics or performance models to drive transformations on diverse and complicated heterogeneous architectures. We leverage the property of performance similarity across the entire workload processing and introduce micro profiling technique at runtime for a fraction of workload to derive performance characteristics for different kernel implementations. Our proposed approach, DySel, not only overcomes the drawback of model-based approaches but also achieves near-oracle performance with profiling overhead under 8% in the worst observed case from our evaluation of Parboil, Rodinia, and SHOC benchmarks on CPU and GPU. As precise modeling becomes harder to come by for modern architectures, our proposed approach can be a simple yet effective alternative for a wide range of systems and applications.

Acknowledgments

This work is supported by the Starnet Center for Future Architecture Research (C-FAR), the DoE Vancouver Project (DE-FC0210ER26004/DE-SC0005515), and the NVIDIA GCoE at UIUC.

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapẽno JVM. In *ACM SIGPLAN Notices*, volume 35, pages 47–65. ACM, 2000.

[2] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. An adaptive performance modeling tool for GPU architectures. In *ACM SIGPLAN Notices*, volume 45, pages 105–114, 2010.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

[4] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 18:1–18:11, 2009. ISBN 978-1-60558-744-8.

[5] S. Benkner, S. Pllana, J. L. Träf, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov. PEPPHER: Efficient and productive usage of hybrid computing systems. *IEEE Micro*, 31(5):28–41, 2011.

[6] 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*, pages 44–54, 2009.

[7] G. Chen, B. Wu, D. Li, and X. Shen. PORPLE: An extensible optimizer for portable data placement on GPU. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 88–100, 2014.

[8] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, pages 81–90, 2000.

[9] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74, 2010.

[10] J.-F. Dollinger and V. Loechner. Adaptive runtime selection for GPU. In *Parallel Processing, 2013 42nd International Conference on*, pages 70–79, 2013.

[11] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju. Auto-tuning of fast Fourier transform on graphics processors. In *ACM SIGPLAN Notices*, volume 46, pages 257–266, 2011.

[12] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng. Twin Peaks: A Software Platform for Heterogeneous Computing on General-purpose and Graphics Processors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 205–216, 2010.

[13] Intel. Vectorizer knobs. `https://software.intel.com/en-us/node/540483`.

[14] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg. pocl: A performance-portable OpenCL implementation, 2014.

[15] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Trans. Parallel Distrib. Syst.*, 22 (1):105–118, 2011.

[16] Khronos OpenCL Working Group and others. The OpenCL Specification. *A. Munshi, Ed*, 2008.

[17] H.-S. Kim, I. El Hajj, J. Stratton, S. Lumetta, and W.-M. Hwu. Locality-centric thread scheduling for bulk-synchronous programming models on CPU architectures. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 257–268, 2015.

[18] L. Li, U. Dastgeer, and C. Kessler. Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems. In *High Performance Computing for Computational Science-VECPAR 2012*, pages 329–345. 2013.

[19] A. Magni, C. Dubach, and M. F. O'Boyle. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 11, 2013.

[20] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[21] N. Rotem. Intel OpenCL Implicit Vectorization Module, 2011.

[22] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, 2008.

[23] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *ACM SIGPLAN Notices*, volume 47, pages 11–22, 2012.

[24] J. Srinivas, W. Ding, and M. Kandemir. Reactive tiling. In *Code Generation and Optimization, 2015 IEEE/ACM International Symposium on*, pages 91–102, 2015.

[25] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler. Flexible software profiling of gpu architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 185–197. ACM, 2015.

[26] J. Stratton, N. Anssari, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, G. D. Liu, and W.-m. Hwu. Optimization and architecture effects on GPU computing workload performance. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, 2012.

[27] J. A. Stratton, S. S. Stone, and W. W. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *Languages and Compilers for Parallel Computing*, pages 16–30. 2008.

[28] J. A. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *IMPACT Technical Report*, 2012.

[29] J. A. Stratton, C. Rodrigues, I.-J. Sung, L.-W. Chang, N. Anssari, G. Liu, W.-m. W. Hwu, and N. Obeid. Algo-

rithm and data optimization techniques for scaling to massively threaded systems. *Computer*, 45(8):0026–32, 2012.

[30] R. Vasudevan, S. S. Vadhiyar, and L. V. Kalé. G-Charm: an adaptive runtime system for message-driven parallel applications on hybrid systems. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 349–358, 2013.

[31] M. J. Voss and R. Eigemann. High-level adaptive program optimization with ADAPT. In *ACM SIGPLAN Notices*, volume 36, pages 93–102. ACM, 2001.

[32] J. Wang and S. Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured GPU applications. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 51–60, 2014.

[33] J. R. Wernsing and G. Stitt. Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. In *ACM SIGPLAN Notices*, volume 45, pages 115–124, 2010.

[34] Y. Yang and H. Zhou. CUDA-NP: Realizing nested thread-level parallelism in GPGPU applications. In *ACM SIGPLAN Notices*, volume 49, pages 93–106, 2014.

[35] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *ACM SIGPLAN Notices*, volume 45, pages 86–97, 2010.