Look Before You Leap: Using the Right Hardware Resources to Accelerate Applications

Jie Shen  
Delft University of Technology  
The Netherlands  
Email: j.shen@tudelft.nl

Ana Lucia Varbanescu  
University of Amsterdam  
The Netherlands  
Email: a.l.varbanescu@uva.nl

Henk Sips  
Delft University of Technology  
The Netherlands  
Email: h.j.sips@tudelft.nl

Abstract—GPUs are widely used to accelerate data-parallel applications. However, while the GPU processing capability is enhanced in each generation, the CPU computing power is also increased by adding more cores and widening vector units. Compared to the rapid development of GPUs and CPUs, the bandwidth of the data transfer between GPUs and the host CPU grows much slower, resulting in a data-transfer wall for using GPUs. In this situation, choosing the right mix of hardware resources—i.e., the right hardware configuration—is critically important for improving application performance, and the right choice is a function of the available hardware resources as well as the application and the dataset to be used. In this paper, we present a systematic approach to determine the hardware configuration that leads to the best performance for a given workload. Our approach captures the variation of hardware capabilities and data-transfer overhead for different applications and datasets, and uses modeling and prediction techniques to determine the best-performing hardware configuration. We have tested our approach on 7 applications with 6 datasets per application. The results show that our approach takes the correct decision in 38 out of 42 test cases, achieving up to 12.6×/6.6× performance improvement compared to an uninformed Only-CPU/Only-GPU decision.

Keywords—hardware selection; workload partitioning; GPUs; multi-cores CPUs; data transfer performance impact

I. INTRODUCTION

Promoted by many existing and emerging data-parallel applications and by the demand for high performance, general-purpose GPU computing is developing rapidly. People choose GPUs to accelerate data-parallel applications, as GPUs are tagged as massive-parallel and high-throughput machines.

However, GPUs are not always the best-performing solution on their own. One needs to take into account that GPUs are designed as accelerators and cannot work in stand-alone mode: a host, which is a CPU, is needed to manage the application execution on the GPU. Modern CPUs have multiple cores and wide vector (SIMD) units, offering thread- and data-level parallelism. Actually, the performance gap between GPUs and CPUs is not as large as it appears (e.g., on average 2.5× in [1]) after applying appropriate optimizations for both. More importantly, this gap is most likely to vary as applications and datasets vary [2], [3]. In other words, the host CPU is essentially a free hardware resource, and can be utilized, together with/instead of the GPU, to improve the overall performance of the workload [4]–[8]. Additionally, a data-transfer bottleneck appears as data to be consumed and produced by the GPU must be transferred from and to the host CPU (the GPU and the host CPU have separate memory spaces). This transfer is performed through a physical connection, typically a PCI Express bus. Because the GPU and CPU throughputs are increasing much faster than the bandwidth of the physical connection, the data transfer tends to become a performance killer, cancelling out any gain obtained on the GPU [9]. In such situations, using only the host CPU is more often than expected the best-performing option.

These observations indicate that, to obtain the best performance, we need to choose the right hardware configuration. This right configuration is a function of the application, the dataset, and the hardware resources. That is to say, to find the right hardware configuration, we need to understand how an (application, dataset) pair performs on each hardware resource. Therefore, it is difficult to devise the best hardware configuration without having the application implementation (for both the CPU and the GPU) and (a representative part of) the dataset at hand. We note that the implementation effort (even if finally only one of the two processors is chosen) is worthy, as the clearer understanding of the application, the more likely it is to get the optimal choice and the best performance for any dataset (keeping in mind that the configuration changes with both the hardware and the datasets).

In this paper, we propose a systematic approach to determine the best-performing hardware configuration. We start from the most widely used hardware setting, a CPU plus a GPU. We have three possible hardware configurations: Only-CPU (run the application on the CPU), Only-GPU (run the application on the GPU), CPU+GPU (run the application in parallel, on both the CPU and the GPU). To characterize the hardware platform in the presence of the workload (i.e., the application and its dataset), we propose two metrics: (1) the relative hardware capability—the ratio of GPU throughput to CPU throughput, and (2) the GPU computation to data transfer gap—the ratio of GPU throughput to data-transfer bandwidth.

Evaluating these metrics poses two major challenges: first, the diversity of hardware, applications and datasets add complexity to the evaluation of the two ratios; second, it is challenging to find the right hardware configuration that leads to the best performance gains in a limited amount of time and effort. Therefore, to estimate the two ratios, we use (partial) profiling for each processor. We find that the profiling provides a realistic estimation, sensitive to hardware and software changes. We assume that CPU+GPU is the right
choice, and we build a task partitioning model, based on the two ratios, to predict the optimal partitioning (which leads to the perfect execution overlap) between the CPU and the GPU. According to the prediction result, we finally determine the best hardware configuration (including the partitioning if CPU+GPU is the final decision).

To use our approach, one needs to have the code for both CPUs and GPUs, and translate single-device code to multi-device code to enable the partitioning when necessary. If the code is not available yet, we recommend using OpenCL [10] for parallelization, as it ensures code portability, and allows for code specialization in a parameter-tuning form (a switch between CPU-/GPU-friendly optimizations) to achieve more performance on different processor families [11]. Our approach also supports legacy code as we make no assumption about the programming models and the compilers that are used.

We evaluate our approach using a set of 7 applications with different problem sizes. The experimental results show that our approach correctly chooses the right hardware configuration in 38 out of 42 (application, dataset) test cases. For all the Only-GPU configuration, to the perfect execution overlap) between the CPU and the two ratios, to predict the optimal partitioning (which leads to the perfect execution overlap) between the CPU and the GPU. According to the prediction result, we finally determine the best hardware configuration (including the partitioning if CPU+GPU is the final decision).

We also show evidence that our approach can be immediately extended to the case of one CPU connected to multiple GPUs, and facilitate the hardware selection when different CPUs or GPUs are given.

In summary, our contributions in this work are:

- We propose a systematic approach to determine the best-performing hardware configuration.

- We develop a prediction-based partitioning model to assist and speed up the hardware configuration decision-making.

- We show that integrated CPU+GPU architectures are more efficient than discrete architectures for applications with a large GPU-computation-to-data-transfer gap.

- We find that data proximity becomes increasingly important for the performance of heterogeneous computing platforms.

The rest of the paper is organized as follows. In Section II, we present three case studies that illustrates the importance of choosing the right hardware configuration, leading to Section III detailing the systematic approach. Section IV presents an experimental evaluation of the approach, followed by a discussion in Section V. We study related work in Section VI. In Section VII, we draw conclusion and discuss future work.

1In this work, data proximity refers to the data being close to the computation unit.

II. CPU+GPU: THREE CASE STUDIES

To show how bad a wrong choice of hardware configuration can be, we present three case studies: Matrix Multiplication (SGEMM), Separable Convolution (SConv), and Dot Product (SDOT). The applications are extracted from the Nvidia OpenCL SDK.

We perform a similar experiment for each case study: we partition the application to run on the CPU and the GPU, and vary the size of the partitions from assigning 100% work to the GPU (Only-GPU) to assigning 100% work to the GPU (Only-CPU) with a granularity of 10%. We measure the execution time of each processor and the final time (the maximum time) after overlapping the executions of the two processors. For the GPU, as the data needs transferring, its execution time consists of two parts, the kernel-computation time and the data-transfer time, which are recorded separately. Figure 1 shows the execution time breakdown.

For SGEMM, as the GPU has higher throughput than the CPU (for 100% work, the GPU kernel-computation time is nearly 5× smaller than the CPU kernel-computation time), and as the data-transfer time takes a small proportion (around 20%) of the whole GPU execution time, the best performance (the minimum $T_{max}$) is achieved with the configuration of 80% work on the GPU and 20% work on the CPU. In other words, SGEMM favors the GPU.

For SConv, the GPU also has around 5× higher throughput, but its overall performance is diminished by the data-transfer overhead (the data transfer takes 11× more time than the GPU kernel computation). As a result, the best configuration is to make 70% work stay on the CPU.

For SDOT, the impact of the data transfer on the GPU is even more significant: the kernel-computation time is almost invisible compared to the data-transfer time (up to 46× difference). This large data transfer cancels out any benefit the GPU contributes, making Only-CPU the best configuration.

These three case studies demonstrate that choosing the right hardware configuration is of importance to the best performance, and the right configuration is influenced by (1) the capabilities of the hardware (the CPU and the GPU - captured as the ratio of GPU throughput to CPU throughput), and (2) the GPU computation to data transfer gap (captured as the ratio of GPU throughput to data-transfer bandwidth). Moreover, the two ratios vary with the changes of applications, datasets, and hardware platforms. For instance, SGEMM and SConv have similar relative hardware capabilities, but their GPU-computation-to-data-transfer gaps differ widely. SGEMM and SDOT are both linear algebra applications, but their two ratios are completely distinct from each other. Even for the same application with different datasets, the ratios vary (shown in Section IV). Thus, we need to develop a systematic approach that captures the variations of the two ratios, and determines the right hardware configuration accordingly.

III. A SYSTEMATIC APPROACH

Our systematic approach starts from the assumption that CPU+GPU is the right hardware configuration. Then, we follow the three main steps. First, modeling the partitioning: build a task partitioning model which represents the
execution of the partitioned workload on the CPU and the GPU. Next, predicting the optimal partitioning: calculate the optimal partitioning by solving the partitioning model. Finally, making the decision in practice: determine the right hardware configuration by convolving the predicted partitioning with practical consideration. Figure 2 depicts the whole approach and each subsection describes each step in detail.

A. Modeling the Partitioning

To model the partitioning means to model the execution overlap between the CPU and the GPU. As also shown in the experiments in Section II, the optimal partitioning is the point where the perfect execution overlap is achieved: $T_G + T_D = T_C$. This equation defines the optimality, where $T_G$, $T_D$, and $T_C$ are the GPU kernel-computation time, the data-transfer time, and the CPU kernel-computation time, respectively. In this equation, we assume that the CPU manages the data transfer and processes its workload share in the same time, typically implemented by using different threads for data transfer and computation. To model $T_G$, $T_C$, and $T_D$, we introduce two pairs of metrics: the total workload size ($W$) and the processing throughput ($P$); the data-transfer size ($O$) and the data-transfer bandwidth ($Q$).

$W$ quantifies the total workload size of an application. Assuming the application problem size is $n$, and each data point $i$ in the problem space performs $w_i$ workload, we have the total workload size as $W = \sum_{i=1}^{n} w_i$ (i.e., we define $W$ as the sum of the workload of all the data points). To simplify the model description, we assume that each data point performs the same amount of work ($w = w_i$), thus we have $W = n \times w$. We further discuss how to make use of our approach for other types of data-parallel applications (e.g., applications with data dependency, with multiple phases, and with imbalanced workload per data point) in Section V. We note that $w$ can be measured with different metrics: compute-intensive applications can use the number of floating-point operations; memory-bound applications can use the number of memory accesses; applications which are iterative in nature (e.g., scientific applications) can use the number of iterations. Let $\beta$ be the fraction of data points assigned to the GPU ($\beta$ shows the partitioning point). The workload size for the GPU ($W_G = n \times \beta \times w$) and the CPU ($W_C = n \times (1-\beta) \times w$) can be expressed by $\beta$. Next, we use $P$, defined as amount of workload solved per second, evaluates the processor’s hardware capability. Therefore, we have $T_G = W_G / P_G$ and $T_C = W_C / P_C$.

Accordingly, we use $O$ (data-transfer size) and $Q$ (data-transfer bandwidth, measured in bytes per second) to quantify the data-transfer time: $T_D = O / Q^2$. We further note that there are three types of data-transfer scenarios: no data transfer (ND), partial data transfer (PD), and full data transfer (FD). For ND, it can be the case that the GPU computation is too intensive, thus the data transfer can be ignored; it can also be the case that the application kernel is a sandwich kernel between another two kernels, thus the data consumed and produced by this kernel does not need transferring. PD means that the data-transfer size ($O$) is proportional to the partitioning point ($\beta$), and therefore a larger GPU partition results in a larger data transfer. FD shows the case that $O$ is independent of $\beta$, thus no matter where the partitioning point is, the data-transfer size is fixed. The data-transfer type decides whether to add $T_D$ into the optimality equation and shows the relationship between $O$ and $\beta$.

Substituting the modeling of $T_G$, $T_C$, and $T_D$ into the optimality equation, and performing equation transformations, we have

$$\frac{W_G}{W_C} = \frac{P_G}{P_C} \times \frac{1}{1 + (O/W_C) \times (P_G/Q)} \quad (1)$$

Equation 1 is the partitioning model that indicates the optimal partitioning. If the data-transfer type is ND ($O = 0$), Equation 1 is degenerated to $W_G/W_C = P_G/P_C$. As $W_G$, $W_C$, and $O$ are expressed by $\beta$, $\beta$ is only determined by the ratio of GPU throughput to CPU throughput ($P_G/P_C$, the relative hardware capability) and the ratio of GPU throughput to data-transfer bandwidth ($P_G/Q$, the GPU computation to data transfer gap). Therefore, if we know the two ratios, we can determine $\beta$ from Equation 1.

B. Predicting the Optimal Partitioning

To predict $\beta$, we need to estimate $P_G/P_C$ and $P_G/Q$. The case studies in Section II already show that the two ratios are application- and dataset-dependent, thus we use profiling (i.e.,

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figures}
\caption{The execution time breakdown with different hardware configurations. The tuple (X,Y) denotes the configuration of X% work on the GPU and Y% work on the CPU. Thus, the tuple (100,0) represents using only the GPU; the tuple (0,100) represents using only the CPU; any tuple in between represents using a mix of both. $T_G$, $T_D$, $T_C$, and $T_{Max}$ denote the GPU kernel-computation time, the data-transfer time, the CPU kernel-computation time, and $\max(T_G + T_D, T_C)$, respectively.}
\end{figure}
we run the given application and dataset on the CPU and the GPU separately, and record their execution time profiles) to make a realistic estimation. We note that profiling is able to respond quicker and more efficiently to application, dataset, and hardware changes, because it models these three aspects together. Solutions that model them separately [12], [13] do not easily adapt to new hardware, different application types, and different datasets.

Observing that profiling for applications with a large number of iterations is usually time-consuming and that most of the execution time is spent on loops, we use partial profiling to save profiling cost. In such case, we limit the number of iterations to \( v (v_{\text{min}} \leq v \leq w) \), and use this partial execution to estimate the application’s global behavior. The choice of \( v \) is a tradeoff between profiling accuracy and cost. Lowering \( v \) below a minimum threshold (which is platform dependent), the estimation quality will not be acceptable.

Assuming the profiled workload is \( W^v \) (\( W^v = n \times v \)), for full profiling \( v = w \), and the profiled data-transfer size is \( O^v \) (the full data-transfer size when we run all the computation on the GPU), we can estimate \( P_G/P_C \) and \( P_G/Q \) as:

\[
\frac{P_G}{P_C} \approx \frac{W^v/T_G^v}{W^v/T_C^v} = R_{GC}, \quad \frac{P_G}{Q} \approx \frac{W^v/T_G^v}{O^v/T_D^v} = \frac{W^v}{O^v} \times R_{GD} \quad (2)
\]

Substituting Equation 2 into Equation 1, we get three solvers of \( \beta \) corresponding to three data-transfer types:

\[
\beta = \frac{R_{GC}}{1 + R_{GC}} \quad (0 < \beta < 1) \quad \text{if } O = O^v = 0 \quad (3a)
\]

\[
\beta = \frac{R_{GC}}{1 + R_{GC} + \frac{n}{w} \times R_{GD}} \quad (0 < \beta < 1) \quad \text{if } O = O^v \times \beta \quad (3b)
\]

\[
\beta = \frac{R_{GC} - \frac{n}{w} \times R_{GD}}{1 + R_{GC}} \quad (\beta < 1) \quad \text{if } O = O^v \neq 0 \quad (3c)
\]

By measuring the time ratios \( T_G^v/T_C^v \) (noted as \( R_{GC} \)) and \( T_D^v/T_G^v \) (noted as \( R_{GD} \)), we can predict \( \beta \). We note that it is necessary to know the ratio of \( v \) to \( w \) only when we enable partial profiling; if full profiling is used, this ratio will be 1, and there is no need to examine the code and get the values of \( v \) and \( w \). Comparing Equations 3b and 3c to 3a, we see that the term \( \frac{n}{w} \times R_{GD} \) actually quantifies the impact of data transfer, which makes \( \beta \) smaller than that without data transfer.

C. Making the decision in Practice

After obtaining the value of \( \beta \), we need to convert it into an actual hardware configuration. To prevent unnecessary waste of hardware computing power, we set lower bound of the GPU partition size and the CPU partition size to half of the total GPU cores, the total CPU cores, respectively. This is a non-aggressive estimation of the usage of hardware computing power. We compare the calculated GPU partition size (\( N_G = n \times \beta \)) and CPU partition size (\( N_C = n \times (1 - \beta) \)) to their lower bounds. If \( N_G \) is smaller than its lower bound, GPU is considered to be underutilized, thus the decision is to use only the CPU; for the same reason, if \( N_C \) is smaller than its lower bound, we will choose only the GPU; if none of these condition is met, the decision is to use the mix of both. As the smallest thread scheduling unit on GPUs is warp/wavefront, once we use the GPU, we will further round up \( N_G \) to multiple of the warp/wavefront size. This is to improve the utilization of the GPU cores, as recommended by [14], [15]. Accordingly, the final \( N_C \) is calculated \((n - N_G)\).

We also note that for the FD data-transfer type, the value of \( \beta \) can be smaller than zero, case in which the data-transfer impact is so large that it cancels out any GPU speedup. In such case, we directly decide to use the CPU.

As the main cost to use this approach is the runtime profiling, we adopt an incremental learning method based on historical data to avoid unnecessary profiling. Assuming the application and hardware platform are given, for every tested problem size \((n)\), its corresponding profiling result and hardware configuration are stored in a repository. This information is used to learn the \( n \)-range for each type of hardware configuration. Once a new problem size is input, we check if it falls into the range of the Only-CPU or Only-GPU configuration, and, if so, directly use the prior decision without going through the profiling, prediction, and decision-making process. Only when the new input falls into the range of the CPU+GPU configuration, the whole process is triggered and completed, and the new profiling result and hardware configuration are added into the repository, further refining the \( n \)-range. If an input size is already tested, we directly fetch its configuration from the repository for reuse.
D. Summary

Our systematic approach uses modeling and prediction techniques to determine a theoretical optimal partitioning for a given application, dataset, and hardware platform, and adjusts it to the right hardware configuration with practical consideration. We implement our approach into the Glinda framework, which was originally developed to find the best workload partitioning for a specific type of applications (imbalance applications) [16], [17]. In this paper, we extend the use of Glinda to generic applications and their hardware configuration deployments.

To use Glinda, one needs to provide parallel implementations (for both CPUs and GPUs) of the application, and enable the profiling and the partitioning (workload distribution and output gathering) in the code if the decision is to use both processors. The resulting code can be reused with little effort for different datasets and hardware platforms. Thus, by using our approach, one obtains a reusable single-device and multi-device code of the application, a hardware configuration per dataset and platform, improved application performance, and more efficient hardware utilization.

IV. EXPERIMENTAL EVALUATION

In this section, we present a detailed evaluation of our systematic approach.

A. Experimental Setup

We use seven applications, summarized in Table I. Vector Addition (SAXPY), Dot Product (SDOT), and Matrix Multiplication (SGEMM) are linear algebra applications typical in libraries like BLAS. Black-Scholes (BS) is a financial model to calculate European option prices. Mersenne Twister (MT) is a pseudo-random number generator. Separable Convolution (SConv) uses 2D separable filters to process images. NBody is a scientific simulation of interactions of individual objects. These applications are widely used in different domains, and have different workload characteristics. For example, SGEMM and NBody are compute intensive as both applications perform a large number of floating-point operations, while the others tend to be memory bound.

For each application, we use six different problem sizes (n1–n6), ranging from within the CPU L1 and GPU L1 cache sizes to within the GPU main memory size, to study how the problem size influences the choice of hardware configuration. This is done by varying the vector or array size in SAXPY, SDOT, and SGEMM, the option count in BS, the number of random number generators in MT, the image size in SConv; and the number of bodies for simulation in NBody. Therefore, we have a total of 42 test cases.

The implementations of the seven applications are provided by Nvidia OpenCL SDK, thus the CPU and the GPU share the same code structure and parallelization approach. We further make the following modifications per application: for the kernel code (on the device side), we apply architecture-friendly optimizations (e.g., using row-major memory accesses on the CPU and column-major memory accesses on the GPU, using zero copy on the CPU and real copy on the GPU, etc.) [11], [18] to improve each processor’s performance; for the host code, we enable the profiling and the partitioning to use both processors when necessary.

The evaluation is performed on a hardware platform integrating an Intel Xeon E5-2620 CPU (hyper-threading enabled) and an Nvidia Tesla K20 GPU. Table II lists the hardware information. The device-side compiler is Intel OpenCL SDK 2013 and NVIDIA OpenCL SDK 4.2, respectively. The host-side compiler is GCC 4.4.6 with -O3 option.

<table>
<thead>
<tr>
<th>Application</th>
<th>Domain</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Addition</td>
<td>Linear algebra</td>
<td>Memory-bound</td>
</tr>
<tr>
<td>Dot Product</td>
<td>Linear algebra</td>
<td>Memory-bound</td>
</tr>
<tr>
<td>Black-Scholes</td>
<td>Finance</td>
<td>Memory-bound</td>
</tr>
<tr>
<td>Mersenne Twister</td>
<td>Random number generator</td>
<td>Memory-bound</td>
</tr>
<tr>
<td>Separable Convolution</td>
<td>Image processing</td>
<td>Memory-bound</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>Linear algebra</td>
<td>Compute-intensive</td>
</tr>
<tr>
<td>NBody</td>
<td>Scientific simulation</td>
<td>Compute-intensive</td>
</tr>
</tbody>
</table>

| Problem size: n1 < CPU L1 & GPU L1, n2 < CPU L2 & GPU L2, n3 < CPU L3, n4 < 100 MB, n5 < 1 GB, n6 < GPU Memory. |

The evaluation is performed on a hardware platform integrating an Intel Xeon E5-2620 CPU (hyper-threading enabled) and an Nvidia Tesla K20 GPU. Table II lists the hardware information. The device-side compiler is Intel OpenCL SDK 2013 and NVIDIA OpenCL SDK 4.2, respectively. The host-side compiler is GCC 4.4.6 with -O3 option.

### TABLE I. THE APPLICATION, ITS DOMAIN, TYPE AND PROBLEM SIZES.

<table>
<thead>
<tr>
<th>Application</th>
<th>Domain</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Addition</td>
<td>Linear algebra</td>
<td>Memory-bound</td>
</tr>
<tr>
<td>Dot Product</td>
<td>Linear algebra</td>
<td>Memory-bound</td>
</tr>
<tr>
<td>Black-Scholes</td>
<td>Finance</td>
<td>Memory-bound</td>
</tr>
<tr>
<td>Mersenne Twister</td>
<td>Random number generator</td>
<td>Memory-bound</td>
</tr>
<tr>
<td>Separable Convolution</td>
<td>Image processing</td>
<td>Memory-bound</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>Linear algebra</td>
<td>Compute-intensive</td>
</tr>
<tr>
<td>NBody</td>
<td>Scientific simulation</td>
<td>Compute-intensive</td>
</tr>
</tbody>
</table>

| Problem size: n1 < CPU L1 & GPU L1, n2 < CPU L2 & GPU L2, n3 < CPU L3, n4 < 100 MB, n5 < 1 GB, n6 < GPU Memory. |

### TABLE II. THE HARDWARE PLATFORM DETAILS.

<table>
<thead>
<tr>
<th>Processor</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (GHz)</td>
<td>Intel Xeon E5-2620</td>
<td>Nvidia K20m</td>
</tr>
<tr>
<td>#Cores</td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>L1 cache size (KB)</td>
<td>32 × 6</td>
<td>16 × 14</td>
</tr>
<tr>
<td>L2 cache size (KB)</td>
<td>256 × 6</td>
<td>128</td>
</tr>
<tr>
<td>Memory capacity (GB)</td>
<td>64</td>
<td>3</td>
</tr>
<tr>
<td>Peak Memory Bandwidth (GB/s)</td>
<td>42.6</td>
<td>208.0</td>
</tr>
</tbody>
</table>

B. The Validation

First, we validate the quality of our approach, i.e., we check if our approach selects the right hardware configuration and predicts the optimal partitioning. We follow the steps specified in Sections III-B and III-C: estimate R_{GC} and R_{GD} by profiling, calculate the partitioning point (\beta) through the solvers, and decide the final hardware configuration taking the lower bounds of the partition sizes into account. Table III shows our experimental results.

We see that R_{GC} and R_{GD} vary for different applications and/or problem sizes, proving that using profiling to estimate the relative hardware capability and the GPU computation to data transfer gap is necessary to correlate the hardware processors’ sensitivity to the application and the problem size.

We use an auto-tuning approach (based on a binary search) to validate our approach. We chose the auto-tuning as the performance baseline because it provides the optimal hardware configuration and, if needed, the optimal partitioning. The auto-tuning starts from \beta = 0.5, tunes \beta until the condition T_G + T_D = T_C is met, and returns the configuration of CPU+GPU with the partition sizes (N_G and N_C). Once N_G or N_C is smaller than its lower bound, the auto-tuning stops and returns the configuration of Only-CPU or Only-GPU. Compared to the results of the auto-tuning approach, our approach takes the right hardware configuration decision in 38 out of 42 test cases. For the other 4 cases marked

3The Nvidia CUDA compiler is v5.5, but the OpenCL compiler is not updated since v4.2.
with (*), the correct decision is Only-CPU. Our approach chooses CPU+GPU because we set a non-aggressive lower bound for $N_G$. Actually, the predicted $\beta$ is very close to 0 in these 4 cases, thus the performance of using the mix is also very close to that of using only the CPU (within 0.5 ms).

We further compare the performance of the predicted partitioning and the auto-tuned partitioning only for the cases that use the CPU+GPU configuration to see if the predicted $\beta$ is close to the auto-tuned $\beta$ in terms of performance. Figure 3 presents the comparison result. We see that the prediction and the auto-tuning obtain similar partitioning points, thus similar performance. For SAXPY, SDOT, BS, MT, SConv, and SGEMM, the relative performance difference is within 12%. For NBody, the performance difference is 31%. We find that NBody has higher prediction error because the prediction underestimates the CPU performance. The causes of the underestimation are under investigation.

To validate the use of partial profiling, we use SGEMM-n5, 6, and halve the width of the matrix A and the height of the matrix B (thus, $\beta = 1/4$). The partitioning we get is very close to that with full profiling, leading to less than 5% performance difference and 50% time saved in profiling.

In summary, our approach successfully selects the right hardware configuration and predicts the close-to-optimal partitioning (if CPU+GPU is selected) in most test cases, leading to close-to-optimal performance for a given workload.

### C. Performance Analysis

Next, we study how much performance is gained through our approach. We compare the performance of the hardware configuration obtained through our approach to that of the Only-GPU and Only-CPU configuration in Figure 4.

The average performance speedup of our configuration to the Only-GPU configuration for SAXPY, SDOT, BS, MT, SConv, SGEMM, NBody is 9.7×, 12.6×, 4.7×, 3.2×, 3.8×, 1.5×, 1.4×, respectively. For SAXPY and SDOT, the performance improvement is significant because the data transfer severely diminishes the performance of Only-GPU. Our approach detects that $R_{GD}$ is very large, and assigns above 90% of the work on the CPU accordingly. For BS, its $R_{GD}$ is also very large, but its $R_{GC}$ is increased by 2.4× compared to that of SAXPY and SDOT, thus the performance improvement is 60% off, as more work is shifted to the GPU. For MT and SConv, both $R_{GC}$ and $R_{GD}$ are moderate, leading to a moderate performance improvement. SGEMM and NBody have the smallest performance speedup. As $R_{GD}$ is very small, more than 80% of the work is actually assigned to the GPU.

We further notice that the hardware configuration in all the test cases is either Only-CPU or CPU+GPU (see Table III). Thus, if we would take the GPU for granted, up to 96% performance would be lost for applications with large $R_{GD}$ and small $R_{GC}$ (like SAXPY and SDOT).

The performance speedup between our configuration and the Only-CPU configuration is smaller for SAXPY, SDOT, BS,
MT, and SConv (less than 1.4×). This is because the CPU takes most of the work after the partitioning. For SAXPY-n2,3, SDOT-n3, BS-n2, the cases marked with (*), it seems that the Only-CPU configuration largely outperforms ours, but the actual difference is within 0.5 ms (see Section IV-B). For SGEMM and NBody, the performance speedup is around 4.7×-6.6×, as the right partitioning is in favor of the GPU.

In summary, we cannot make a blind or biased hardware choice when accelerating applications. Our approach provides a way to properly understand the application and the hardware resources, and derive the right hardware configuration leading to the best performance gain.

D. The Impact of Data Transfer

Finally, as the data transfer plays an important role in the hardware configuration, we quantify its impact on performance. We assume that the data transfer overhead is removed, imitating an integrated CPU+GPU architecture, and use Equation 3a to predict the new partitioning point.

We see that the new β is shifted towards 1, making most test cases choose the CPU+GPU configuration (only SConv-n1-3 and SGEMM-n1-2 stay with Only-CPU). The impact of data transfer varies. For SAXPY, SDOT, and BS (the ones with higher $R_{GD}$), up to 80% of the work is moved from the CPU to the GPU due to the absence of data transfer, leading to up to 7.6× performance improvement compared to the original partitioning with data transfer. For MT and SConv (the ones with moderate $R_{GD}$), the GPU gets around 50% more work, and the final performance is improved by 2.6× and 4.7×, respectively. For SGEMM and NBody, the change of β is small. As the original data-transfer overhead is already very small, the performance change is within 1.2×.

The applications with higher $R_{GD}$ get more performance impact from the data transfer. In such cases, an integrated CPU+GPU architecture (like AMD APU [19] and Intel Sandybridge [20]) can be a better hardware solution than the discrete CPU+GPU architectures, as it largely eliminates the data-transfer overhead. A software solution is to hide the data transfer by overlapping it with the computation [21], but the implementation may not be easy for every application. Recently, Nvidia released CUDA 6 [22], which supports “Unified Memory” between the CPU and the GPU, to simplify programming by removing the manual data copy. However, the CUDA runtime and driver have to migrate data between the two processors, thus the data-transfer overhead still exists.

V. DISCUSSION

This discussion focuses on the features, limitations, and possible extensions of our hardware selection approach.

First, the experiments show that the impact of data transfer can be as significant as the kernel computation on the hardware configuration and the overall performance. The problem is, the data-transfer bandwidth grows much slower than the GPU computation throughput, resulting in a data-transfer wall in heterogeneous computing. The larger the gap between the computation throughput and the data-transfer bandwidth, the more likely it is to suffer from the data-transfer penalty, and the more necessary it is to pay attention to the data transfer. Thus, data proximity becomes an increasingly important factor when deciding the right hardware configuration, i.e., computation should follow the data, not vice versa.

Our systematic approach considers both the kernel computation and the data transfer to determine the hardware configuration. We find that the approach can be immediately extended
to the hardware combination of one CPU and multiple identical GPUs. Assuming there are \( m \) GPUs connected to the host CPU, and their ratios (noted as \( R_{GD}^{On} \) and \( R_{GC}^{On} \)) are identical to each other, first we regard all the GPUs as an aggregated GPU, and we have,

\[
\begin{align*}
R_{GC} &= m \times R_{GC}^{On} \\
R_{GD} &= R_{GD}^{On} & \text{if } O = O^V \times \beta \\
R_{GD} &= m \times R_{GD}^{On} & \text{if } O = O^V \neq 0
\end{align*}
\]

The aggregated GPU’s \( R_{GC} \) is increased \( m \) times as the GPU computing power is aggregated. For \( R_{GD} \), depending on the data-transfer type, the calculation varies. For the PD type, \( R_{GD} \) is the same as that of each GPU, because the data-transfer work is equally shared by all the GPUs. For the FD type, \( R_{GD} \) is increased \( m \) times as the same amount of data has to be copied to all the GPUs. By substituting Equation 4 into Equation 3a-3c, we calculate the aggregated partitioning point (\( \beta_m \)), and each GPU gets \( 1/m \times \beta_m \times n \) work. Thus, by measuring the ratios of one GPU, we are able to get the partitioning for multi-GPUs. We validate this using SGEMM-n5,6 on a platform with one CPU and two GPUs. The performance is improved by 44% compared to that with one CPU and one GPU.

We note that this extension is especially useful when a single GPU cannot hold all the data assigned to it due to the memory capacity limitation. A solution is to make the CPU share more work, which leads to potential performance loss. An alternative solution is to use multiple GPUs without sacrificing the performance. By using partial profiling (because of the memory limitation, we cannot use full profiling), we can obtain the ratios for one GPU, and eventually derive the right decomposition to multi-GPUs.

In fact, our approach can be extended to a different mix of hardware accelerators, connected to the host with either a shared memory or a physical connection (e.g., CPUs, GPUs, FPGAs, Xeon Phi), as we do not model the accelerator architecture in detail.

In this paper, the applications we used are all data-parallel applications with similar and independent workload per data point. However, our approach can be also applied to applications in which several (adjacent or random) data points are needed to calculate one result (e.g., stencils) by replicating some/all data on both the CPU and the GPU, and applications with multi-phases (kernels) and inter-phase synchronizations by using our approach phase-by-phase. In addition, we have already applied and validated our approach for applications with imbalanced workload per data point in [16], [17].

We further note that our approach can assist hardware selection. For example, if the CPU is fixed, and one GPU must be selected from a set of options, we can compare the predicted \( \beta \) in each combination. If \( \beta_i > \beta_j \), GPU \( i \) makes the CPU process less work than GPU \( j \), leading to lower overall execution time. Thus, the GPU which has the minimum \( \beta \) is the best choice, performance-wise.

VI. RELATED WORK

Several studies [1], [9], [23] have examined the GPU vs. CPU performance issues, showing that CPUs are good candidates for data-parallel applications, and the data-transfer overhead can be significant when using GPUs. Other studies have proved that CPU+GPU heterogeneous computing leads to improved application performance and hardware utilization [4]–[8]. Our approach is an essential improvement to these studies, as it recommends the best hardware configuration to use.

To obtain the best performance on a given hardware platform, performance modeling techniques [12], [13] have been developed. However, these techniques model the application and the hardware separately, and therefore they are not adaptive to new platforms, different application types (e.g., memory-bound or compute-intensive), and different datasets: a lot of effort is spent on benchmarking hardware, modeling applications, matching models, and calibration. Compared to these solutions, our approach models the application performance together with the dataset and the platform by using profiling, thus it reacts to application, dataset, and hardware changes more easily and more efficiently.

To determine the best-performing partitioning between the CPU and the GPU, the work in [24], [25] proposed prediction models that relate program features to the optimal partitioning by using machine learning. In this approach, a lot of training data needs to be collected and used to build the model. This phase is costly and must be repeated for platform changes. Qilin [26] is an adaptive partitioning approach. Our work relates to Qilin as both approaches use profiling to tackle application and hardware changes. However, Qilin requires repetitive profiling for curve fitting, while our approach uses partial profiling and incremental learning to mitigate this overhead. Moreover, Qilin does not differentiate the data-transfer time from the kernel-computation time, thus the partitioning quality is lower than ours.

In summary, our approach has its advantages in terms of cost-effectiveness for applying it, adaptivity to the application, dataset, and hardware diversities, and can be extended easily to different mixes of hardware accelerators.

VII. CONCLUSION

Choosing the right hardware configuration (among Only-CPU, Only-GPU, and CPU+GPU) is essential for improving application performance. A wrong hardware choice or computation decomposition may lead to significant performance loss. In this paper, we propose a systematic approach to determine the best-performing hardware configuration, taking both the hardware capabilities and the data-transfer overhead into consideration. Experiments with different applications and datasets show that our approach, through modeling the partitioning between the CPU and the GPU, predicting the optimal partitioning, and making the hardware choice in practice, correctly determines the right hardware configuration to use, leading to as much as 12.6× and 6.6× performance improvement compared to the Only-CPU and Only-GPU configurations, respectively.

In the future, we plan to investigate in more detail the use of the integrated CPU+GPU architecture as it is a promising solution to eliminate large data-transfer overhead. We further want to investigate more classes of applications, characterize application workloads, and extract representative workloads for hardware profiling, so that profiling data can be reused for multiple applications.
REFERENCES


