A Dynamic Self-Scheduling Scheme for Heterogeneous Multiprocessor Architectures

MEHMET E. BELVIRANLI, LAXMI N. BHUYAN, and RAJIV GUPTA, University of California, Riverside

Today's heterogeneous architectures bring together multiple general-purpose CPUs and multiple domain-specific GPUs and FPGAs to provide dramatic speedup for many applications. However, the challenge lies in utilizing these heterogeneous processors to optimize overall application performance by minimizing workload completion time. Operating system and application development for these systems is in their infancy.

In this article, we propose a new scheduling and workload balancing scheme, HDSS, for execution of loops having dependent or independent iterations on heterogeneous multiprocessor systems. The new algorithm dynamically learns the computational power of each processor during an adaptive phase and then schedules the remainder of the workload using a weighted self-scheduling scheme during the completion phase. Different from previous studies, our scheme uniquely considers the runtime effects of block sizes on the performance for heterogeneous multiprocessors. It finds the right trade-off between large and small block sizes to maintain balanced workload while keeping the accelerator utilization at maximum. Our algorithm does not require offline training or architecture-specific parameters.

We have evaluated our scheme on two different heterogeneous architectures: AMD 64-core Bulldozer system with nVidia Fermi C2050 GPU and Intel Xeon 32-core SGI Altix 4700 supercomputer with Xilinx Virtex 4 FPGAs. The experimental results show that our new scheduling algorithm can achieve performance improvements up to over 200% when compared to the closest existing load balancing scheme. Our algorithm also achieves full processor utilization with all processors completing at nearly the same time which is significantly better than alternative current approaches.

Categories and Subject Descriptors: C.1.4 [**Computer Systems Organization**]: Other Architecture Styles—Heterogeneous (hybrid) systems; D.1.3 [**Software**]: Concurrent Programming—Parallel programming; C.1.2 [**Computer Systems Organization**]: Multiple Data Stream Architectures–Single-instruction-stream, multiple-data-stream processors (SIMD)

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Dynamic self-scheduling, workload balancing, GP-GPUs, FPGAs

ACM Reference Format:
Belviranli, M. E., Bhuyan, L. N., and Gupta, R. 2013. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. ACM Trans. Architec. Code Optim. 9, 4, Article 57 (January 2013), 20 pages.
DOI = 10.1145/2400682.2400716 http://doi.acm.org/10.1145/2400682.2400716

1. INTRODUCTION

Future chip multiprocessors (CMPs) will feature hundreds of simple cores connected to a high-throughput on-chip network. The power consumption of running all cores concurrently will exceed the power budget of the chip. Moreover, for SIMD-based scientific

This work is supported by the National Science Foundation under grant CCF-0905509.
Authors' addresses: M. E. Belviranli (corresponding author), L. N. Bhuyan, R. Gupta, Computer Science and Engineering Department, University of California Riverside; email: ebelviranli@gmail.com.

DOI 10.1145/2400682.2400716 http://doi.acm.org/10.1145/2400682.2400716

loads, the energy per instruction (EPI) of processors is highly inefficient compared to the emerging hardware accelerator (HA) technologies, such as GPUs and FPGAs.

Heterogeneous multiprocessor systems featuring hardware accelerators have been used for a wide range of applications yielding significant speedups compared to CPU-only execution. Each type of hardware accelerator, such as FPGA or GPU, provides substantial performance improvement for the applications within its target domain. Image processing [De Ruijsscher et al. 2006], data mining [Baker and Prasanna 2005], and bioinformatics [Harris et al. 2007] are examples of applications with hardware implementations on FPGA and K-means [Che et al. 2008], AES encryption [Yamanouchi 2007], and network packet processing [Smith et al. 2009] are examples of GPU accelerated applications. Heterogeneous architectures like Cray XD1[Fahey et al. 2005] and SGI Altix 4700 [SGI 2008] employ FPGAs as accelerators, whereas nVidia Tesla [Lindholm et al. 2008] employs GPUs for acceleration.

Heterogeneous systems provide a hybrid execution environment that allows applications to run concurrently on accelerators and CPUs. However, programming heterogeneous architectures requires careful consideration of underlying computational capabilities and memory bandwidth between the host and the accelerators. To achieve maximum gain and resource utilization on these architectures, the computation workload in an application should be distributed across accelerators and CPUs according to their computational capabilities. Workload partitioning often involves both input data distribution and assignment of different application tasks.

Static partitioning techniques are ones where the developers estimate running times of tasks and/or time to process a range of data so that accelerator and CPU utilization can be maximized. Estimations are based on theoretical accelerator capabilities, compile-time parameters, or an offline training period over a fraction of the input dataset. Many applications [Galanis et al. 2005; Yeung et al. 2008; Scrofano et al. 2007; Tripp et al. 2005; Tsoi et al. 2011] have employed various compile-time data partitioning and task partitioning schemes. However, many of them deal with either reconfigurable or GPU architectures without considering simultaneous execution on the CPUs as well. One of the most notable research work on heterogeneous systems, Axel [Tsoi and Luk 2010], models communication and computation capabilities specific to each hardware configuration. For each processor, data transfer and execution times are estimated offline based on bandwidth speed as well as processor count and frequencies of the processor. The disadvantage is that the model must be updated for each application based on how it performs on that specific hardware. Also, theoretical capabilities of accelerators are very prone to misrepresenting actual runtime performance. Qilin [Luk et al. 2010], another important study on static workload partitioning, builds performance models per accelerator and CPU by measuring transfer and execution times during predefined training runs. These models are used to estimate the workload balance across processing units for the actual run. The models are again application specific and only represent the actual performance for a limited subset of the data. Therefore, static partitioning techniques fail to address runtime dynamics of the application and underlying architecture.

Dynamic partitioning, on the other hand, can more accurately balance task and data workload among heterogeneous processors [Augonnet et al. 2009; Hermann et al. 2010; Tse et al. 2010], and can better utilize all computational units during execution. It requires proper monitoring mechanisms for each accelerator type, identification of tasks and data as computational *block*s, and accurate performance estimation throughout the execution. Studies in Augonnet et al. [2009] and Hermann et al. [2010] use work stealing schemes which are also not suitable for accelerator-based systems due to possible redundant data transfers. The work to be stolen by CPU should have already been transferred to GPU, therefore the time spent on data transfer will be wasted. The work in Tse et al. [2010] uses two basic scheduling policies, exponential incremental and linear incremental. These schedulers increase task size exponentially (by a factor of m) or linearly (increasing by n) for each time a processor requests a block. In both cases, faster processing units reach bigger task sizes more quickly as execution progresses. However, exponentially growing task sizes increases the likelihood of load imbalance due to assignment of large blocks to slow processors towards the end of the execution. On the other hand, increasing linearly causes the use of excessive amount of blocks which decreases accelerator utilization.

An ideal dynamic load balancing scheme should be able to achieve minimum possible amount of execution time without any offline runs. For heterogeneous architectures, two factors are critical to achieve such an ideal balancing: (1) accurate measurement of computational weights per each processing unit and (2) the ideal size of computational task and data (i.e., *block size*) that is sent to an accelerator during each task assignment. If the block size is too small, data transfer to the device will take most of the time due to underutilization of DMA and the device initialization overhead will be excessive. On the other hand, large blocks will lead to better performance in accelerators; however, taking this to an extreme will result in workload imbalance. Finding the right block size will minimize underutilization and load imbalance, thus yielding the shortest execution time. Moreover, performance of an accelerator is significantly better when the block size is a multiple of its number of parallel threads. To best of our knowledge, none of the existing dynamic heterogeneous scheduling algorithms has considered effects of block sizes on accelerator performance.

In this article, we present HDSS, a *H*eterogeneous *D*ynamic *S*elf-*S*cheduler, a new dynamic load balancing scheme for loop iterations on heterogeneous architectures. HDSS uses *weighted self-scheduling* to fully utilize all available processing units in the system during the application lifetime. Our algorithm dynamically resizes blocks to prevent underutilization and load imbalance of GPUs or FPGAs due to small or large block sizes. Unlike existing approaches, HDSS does not require an offline training period or device- and application-specific performance calculations. Moreover, HDSS is also able to schedule loops with dependent iterations as well via wavefront parallelization.

The new algorithm has two phases. The adaptive phase determines computational weights of processors using the smallest possible amount of data. During this phase we start with small block sizes and increase them gradually while continuously monitoring the performance. Computational weights are readjusted after execution of each block until they become stable. In this phase, we use the *least squares estimation* technique for quick convergence to accurate weights by predicting right amount of next block size. The second phase is the completion phase where the remainder and majority of the computation is done based on the computed weights. This phase uses a modified version of Guided Self-Scheduling (GSS) [Polychronopoulos and Kuck 1987] algorithm to achieve near minimum number of blocks. Once computational weights are accurately determined in the previous phase, our modified version of the original guided self-scheduling (GSS) [Polychronopoulos and Kuck 1987] algorithm (MGSS) ensures that executional units (CPUs and accelerators) finish execution at nearly the same time in the completion phase.

We have evaluated our scheme on two different heterogeneous platforms: a CPU+GPU system featuring nVidia Fermi C2050 GPU and a CPU+FPGA system (SGI Altix 4700) employing Xilinx Virtex 4 FPGA. For CPU+GPU experiments, we have used four applications, Blackscholes, Histogram, BoxFilter, 2D Heat equation, and String matching. For CPU+FPGA experiments, we have developed FPGA versions of Blackscholes and Histogram applications. We have implemented and compared our load balancing scheme with the recent major studies on heterogeneous load balancing: Axel [Tsoi and Luk 2010], Qilin [Luk et al. 2010], and Monte-Carlo [Tse et al. 2010].

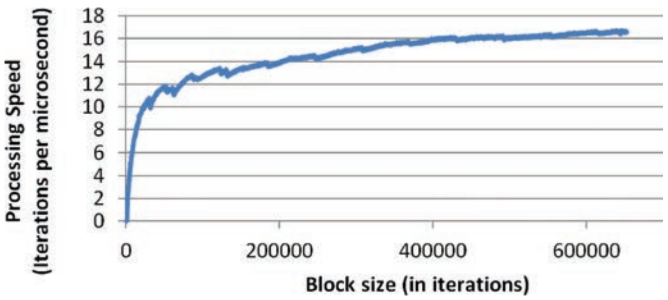


Fig. 1. Processing speed of GPU (iterations per microsecond) for varying number of block sizes.

The results show that HDSS provides performance improvements of up to 219% when compared to its closest load balancing scheme. Results also show that HDSS is able to minimize processor idle times throughout the execution. We present HDSS along with a generic C++ API for easy adaptation.

This article makes the following contributions.

—A new self-scheduling algorithm, HDSS, optimized for heterogeneous architectures is presented. The algorithm is block size aware and achieves near-optimal utilization for all processors without any need for offline training.
—The algorithm has been implemented and tested on CPU+GPU and CPU+FPGA platforms. Our algorithm is able to increase performance by up to 219% when compared to the best performing existing algorithm. Moreover, HDSS achieves running times that are very close to those achieved by ideal partitioning.
—An API has been developed for easy adaptation of the algorithm for new/existing parallel applications.

The remainder of this article is organized as follows: Section 2 presents motivations behind our scheme. Section 3 describes how HDSS works in detail. Section 4 presents our performance evaluation and comparison to previous approaches. Section 5 describes additional related work. The last section is conclusion and future work.

2. MOTIVATION

2.1. Effect of Block Size

A common approach in loop scheduling is to divide the entire data range into small groups of iterations, called *blocks*. Throughout the rest of this article, we will use the term *block* to refer to a chunk of data sent to a computational unit for processing. The term *block size* will be used to refer to the quantity of data (e.g., number of array indicies, loop iterations, bytes, etc.) in the block. Division of a large workload into smaller blocks reduces the idling of some processors due to waiting for other busy processors towards the end of the execution. For heterogeneous systems, the data is also sent in blocks to the accelerator memory before execution starts. We have observed that size of the blocks has much more impact on the performance of heterogeneous systems than on multicore systems.

To observe the effects of block sizes on accelerated computing, we conducted an experiment on our CPU + GPU system. In this experiment we ran the Blacksholes application on the nVidia C2050 GPU while linearly increasing the size of the blocks sent to GPU from 1024 to 67M. After execution of each block, the amount of time to process the block is divided by the size of the block and plotted as the processing rate. Figure 1 shows how accelerator performance, quantified in terms of “iterations executed per microsecond”,

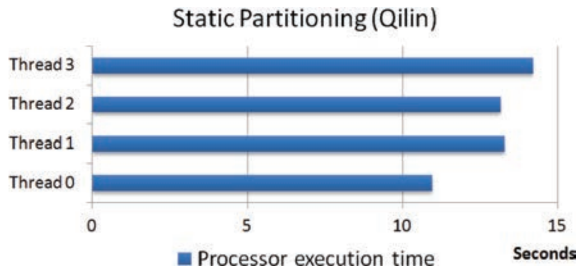


Fig. 2. Qilin [Luk et al. 2010] workload distribution with 1 GPU (thread 0) + 3 CPU threaded execution of Blackscholes application.

changes with the block size. Processing rate increases rapidly with the block size initially and then reaches stability. Several factors play a role in causing this type of behavior. A primary reason is due to the fact that accelerators usually overlap computation and communication so that input data is processed in a streaming fashion. When the amount of data to be sent to the GPU is small, it is underutilized and the optimization is less effective due to lack of data. Direct memory access (DMA) is also underutilized for small amount of data transfers. Another reason is the underutilization of possible pipelined or threaded parallelism implemented inside the hardware accelerator. If the size of the block is smaller than the pipeline steps, or the number of parallel execution units inside the accelerator, then the full computing power of the accelerators will not be exploited. The last reason is the constant initialization overhead associated with the use of accelerators. Small block sizes result in a greater number of blocks which increases the total time spent on initialization and hence an increases the execution time.

Thus, we can conclude that while small blocks provide better load balancing, large block sizes achieve higher utilization. An ideal scheduling algorithm should address this trade-off by using large enough blocks for better performance while keeping the workload balanced so that the threads finish their execution at nearly the same time.

2.2. Computational Rates (Weights)

Processors in a heterogeneous system have varying amount of computational powers. Hardware characteristics of the device and computational requirements of the application together determine the computational power of a processor. *Computational rate*, which represents the relative speed of each unit, is necessary for estimating the correct fraction of workload to be assigned to the processing unit.

Wrongly estimated compute rates will overload slow processors with excessive workload starting from the first assignment. This would cause faster accelerators to wait for the slow processors when there are no more iterations remaining to execute. Figure 2 shows how our implementation of a previous static partitioning scheme, Qilin [Luk et al. 2010], on the CPU+GPU system fails to balance the workload for the Blackscholes application. Computation power of GPU, running as thread 0, has been underestimated during the offline training and thus resulted in idle periods for the GPU. Thread 0, the GPU thread, has been assigned less workload than it can actually handle and it stays idle towards end of the execution. Offline training was not successful at producing correct compute ratios for GPU and CPUs.

2.3. Online Training

In this article we use online training that allows us to capture the compute rates (weights) on-the-fly without wasting any time on a prior offline run. The length of the online training should be long enough, since an overly short training period may

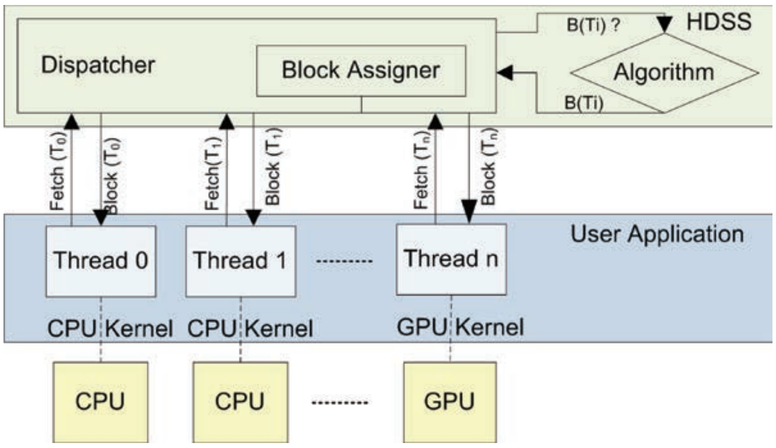


Fig. 3. Overall mechanism of HDSS.

yield inaccurate compute weights. Online training should carefully measure the vastly different computational powers of the executional units by assigning blocks of loop iterations to processors and tracking the time required to process them.

The effects of block size on performance observed in Figure 1 implies that the training period should find the correct computational rates by testing with increasing number of block sizes until the weights do not change. However, once correct rates are found, large blocks should be avoided towards end of execution since that will increase the chance of load imbalance by keeping slow processors busy at the end.

To address these concerns, we developed a dynamic self-scheduling scheme with a two-phase algorithm: the first phase starts with small blocks and increases them gradually until it determines accurate compute rates (weights). Then we switch to the second phase which uses large blocks initially, then decreases block sizes as it reaches completion to ensure that all units complete execution at nearly the same time.

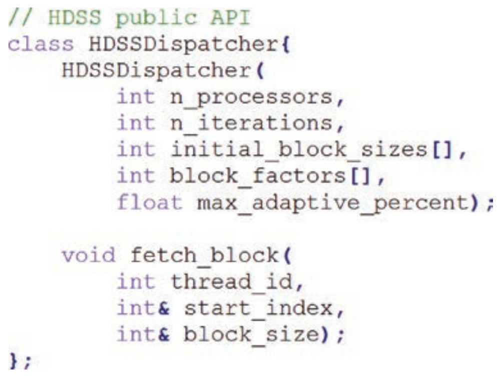
3. HDSS: HETEROGENEOUS DYNAMIC SELF-SCHEDULER

3.1. Overview

In a typical data-parallel application, parallel threads simultaneously process a specific portion of the shared input data in chunks (blocks). When all data is processed, parallel threads merge and the application either terminates or proceeds to the next phase of computation. Determining the amount of data for each thread during this parallel processing phase is the responsibility of the workload scheduler.

HDSS is a self-scheduling scheme which assigns blocks of loop iterations (i.e., input data) to each processing unit (i.e., CPUs, GPUs, FPGAs) when they become idle. The overall mechanism of HDSS is given in Figure 3. In HDSS, each processing unit P_i is represented by a dedicated parallel *thread* T_i (i.e., *representative thread*) which is responsible for fetching the next available block and executing device-specific binary for the retrieved block of iteration. For CPUs, representative threads are basically main threads which handle the execution, whereas they are responsible for transferring input/output data and launching the computation kernel for GPUs and FPGAs.

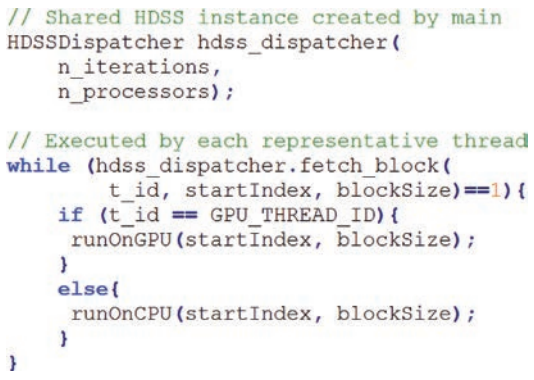
HDSS Dispatcher. Fetch request is satisfied by the *HDSSDispatcher* which internally calls the HDSS algorithm to find the right block size $B(T_i)$ for the requesting thread T_i. Once a block size $B(T_i)$ is determined, the next unprocessed iteration block, $Block(T_i)$, with size $B(T_i)$ is created by the *Block Assigner* and returned to the requesting thread.



```
// HDSS public API
class HDSSDispatcher{
    HDSSDispatcher(
        int n_processors,
        int n_iterations,
        int initial_block_sizes[],
        int block_factors[],
        float max_adaptive_percent);

    void fetch_block(
        int thread_id,
        int& start_index,
        int& block_size);
};
```

Fig. 4. The two public interfaces of HDSS API: The constructor and the *fetch_block* method.



```
// Shared HDSS instance created by main
HDSSDispatcher hdss_dispatcher(
    n_iterations,
    n_processors);

// Executed by each representative thread
while (hdss_dispatcher.fetch_block(
        t_id, startIndex, blockSize)==1){
    if (t_id == GPU_THREAD_ID){
     runOnGPU(startIndex, blockSize);
    }
    else{
     runOnCPU(startIndex, blockSize);
    }
}
```

Fig. 5. Example usage of HDSS via representative threads.

HDSS is a noncentralized scheduler, where the scheduling algorithm is run by the thread requesting the block. Scheduling overhead is distributed across representative threads, therefore increasing the scalability of our scheme considerably. Parallel runs of the HDSS algorithm need only to be synchronized when the global data pointer for the processed data needs to be advanced after a block assignment. However, no computation is needed during this synchronization and the overhead is very small and negligible.

API and Usage. HDSS provides a simple API, *HDSSDispatcher* class, to communicate with the parallel threads representing heterogeneous processors. As shown in Figure 4, *HDSSDispatcher* has a constructor that takes the total number of iterations, number of processors, initial block sizes for each processor, block size factors for each processor, and maximum length of the adaptive phase as parameters. Initial block sizes and block size factors have a default value of 128 and maximum length of the adaptive phase is taken as 20% of total number iterations by default. *fetch_block* method simply takes *thread id* as argument and returns *start* and *end* indicies of the iteration space that is assigned by HDSS to requesting thread.

Figure 5 shows the usage of HDSS. For each application, a shared instance of the *HDSSDispatcher* object is created. Whenever a representative thread for a processor becomes idle, *fetch_block* is called to get new workload. This method is blocking when there are no available blocks whose dependency constraints are satisfied. Once

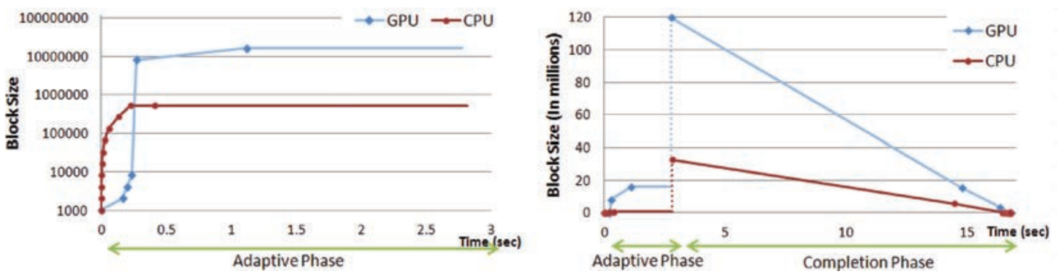


Fig. 6. An example showing block assignments by time for a two-threaded (1 GPU and 1 CPU thread) run of Histogram application on CPU+GPU system. Figure (a), on the left, illustrates the assignments dispatched during the adaptive phase. In this figure, block size axis is shown in logarithmic scale for a better separation of block assignments in the chart. Figure (b), on the right, shows the two phases of the algorithm separated with a clear jump in assigned block sizes just after computational weights are determined and stabilized.

a block size is retrieved, the programmer must call processor-specific implementations (kernels) of her application on the determined data block.

3.2. The Algorithm

Overview. The HDSS algorithm determines the ideal block size per processor using a two-phased approach. The initial phase, called the *adaptive phase*, is responsible for accurately finding the computational weights which reflect the relative processor speeds. This phase processes a relatively small amount of data whose upper bound is set as a fixed percentage of the total data. The final phase, called *completion phase*, processes the remainder of the data using the weights computed in the adaptive phase. This phase starts with the largest possible blocks to optimize the execution time by reducing the dispatching overhead. Block size decreases steadily as the computation works towards end so that all units complete execution at nearly the same time. As an example, the progression of block sizes during the adaptive phase is illustrated in Figure 6(a). The experiment is for Histogram application on a CPU+GPU system with 210M inputs. The overall process including completion phase is illustrated in Figure 6(b) for the same experiment. The sudden increase in the curve is the moment where HDSS assigns largest possible block for the very first block request of the second phase.

A flow chart for our proposed self-scheduling algorithm, HDSS, is given in Figure 7. Our algorithm stores the remaining number of iterations and total computational weight as global variables. In addition, each processing element has the following attributes:

—the initial block size (user-defined);
—computational weight in iterations per microsecond;
—block size factor (user-defined);
—dynamic list of blocks that have been assigned to the processor before. Each entry in the list contains the size of the block as well as the execution start and end times of the block.

For the first block assignment, the initial block size parameter for the requesting processor is set as the block size. During subsequent block requests, HDSS decides whether to continue with the adaptive phase or move to the next phase based on dynamics of the execution. Once HDSS concludes that the adaptive phase should be terminated, all subsequent block assignments by all processors are calculated by the completion phase.

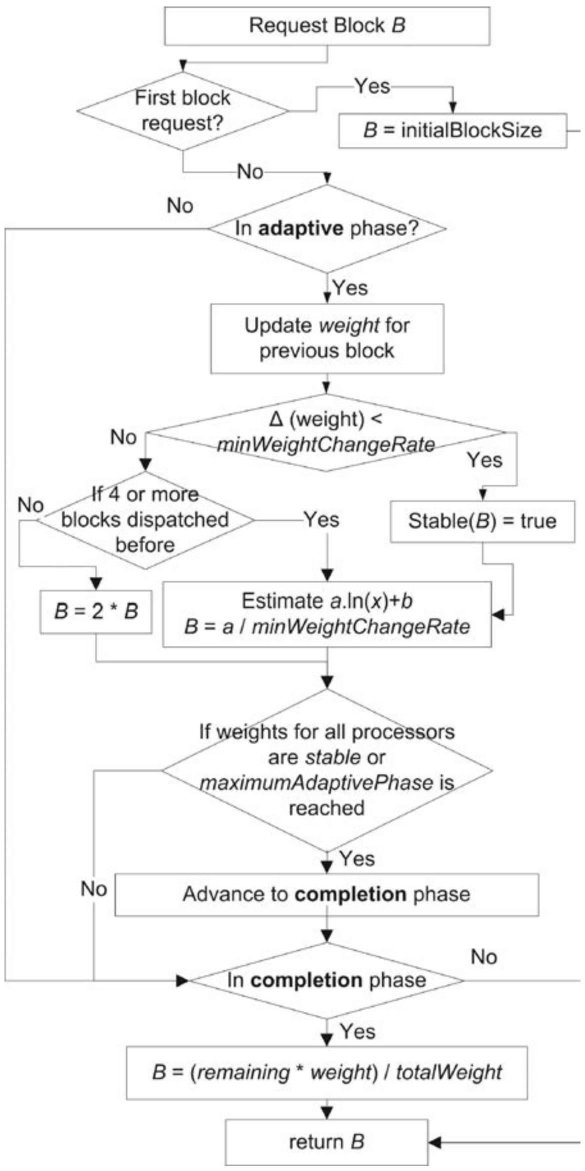

Request Block B
First block request?
Yes
B = initialBlockSize
No
No
In adaptive phase?
Yes
Update weight for previous block
Δ (weight) < minWeightChangeRate
No
Yes
If 4 or more blocks dispatched before
No
Yes
Stable(B) = true
B = 2 * B
Estimate a.ln(x)+b
B = a / minWeightChangeRate
If weights for all processors are stable or maximumAdaptivePhase is reached
Yes
No
Advance to completion phase
No
In completion phase
Yes
B = (remaining * weight) / totalWeight
return B

Fig. 7. Flow chart for the HDSS algortihm.

Computational rates (weights) are calculated based on the total time required for both data transfer (in/out) and execution time. This is essential for accurate load balancing as our results will demonstrate.

Adaptive phase. This phase finds the relative speed of processors resulting in weight factors to be used to assign workload in the completion phase. The main challenge is to keep the initial adaptive phase as short as possible so that the bulk of the data is left for the final completion phase.

To achieve this, we start with small block sizes initially and increase them gradually based on the observed performance until we are able to determine accurate computation

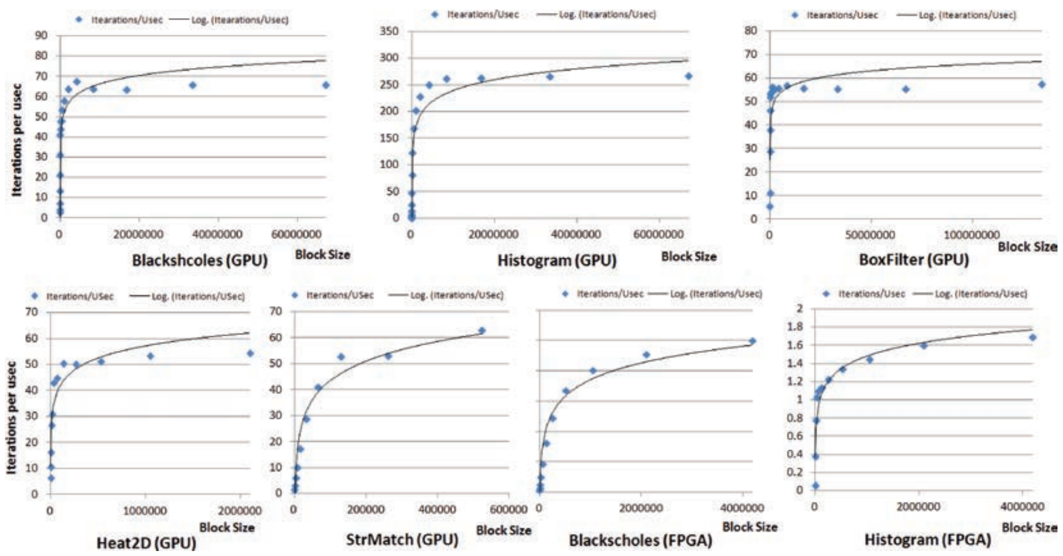


Fig. 8. Change of compute rates (iterations/usec) for varying number of blocks. Solid black line shows logarithmic curve fitting for each evaulated benchmark.

weights for each device. For an accelerator, ideal starting block size would be a multiple of the number of threads inside that accelerator.

The adaptive phase first measures the number of iterations completed in the previous step per unit time. This gives us the computational weight, a relative processing speed of that processor for the last step. If the change in weights between previous step and current step is less than MIN_CHANGE_RATE (which is set as 0.01), current weight for that processor is considered as stable. Otherwise, we need to increase block size for the next step.

In order to reach convergence in weights quickly, we estimate the block size using *least squares estimation* for logarithmic curves. Through our experiments, we have determined that logarithmic curve (Figure 8) is an appropriate fit for representing the change in computation weight for the applications we have used. After we have enough measurement points (which is set as 4), we estimate parameters a,b for the logarithmic equation $a\ ln(x)+b$. Once the parameters are obtained, we estimate a stable computational weight for that processor by finding the point where the slope of the fitted curve is equal to a / MIN_CHANGE_RATE.

Least squares estimation allows HDSS to dispatch fewer number of small blocks in the adaptive phase. Since small blocks underutilize accelerators, HDSS quickly skips them and proceeds to larger blocks which have higher utilization. After the estimated block size is dispatched to the processor, in the subsequent block request, HDSS checks whether the estimation is accurate by calculating the computational weight again. If the weight change is still not stable, the process of estimation continues.

Once a computational weight is found to be stable, HDSS keeps increasing block sizes for other unstable processors until either all compute weights become stable or a fixed percentage (20%) of all data is processed. The latter case is a forced upper bound to limit the length of the adaptive phase.

For applications with dynamically changing weights, *MIN_CHANGE_RATE* constant can be set to a value very close to zero and *max_adaptive_percent* can be set to 1.0. This will enable HDSS to monitor the change until the execution ends. However, our experiments show that setting *max_adaptive_percent* to 0.2 is enough to achieve stable compute rates.

Completion phase. Once all weights are stable, the algorithm moves into the completion phase. Block sizes from the adaptive phase are no longer used. Instead, HDSS uses the Modified Guided Self-Scheduling (MGSS) to find the correct block sizes. The completion phase integrates computational weights calculated in the previous step into MGSS. Eq. (1) is the original GSS formula, which simply divides the remaining number of iterations R to the total number of processors P. MGSS uses Eq. (2) which distributes R to processors proportional to their calculated weight w_i.

$$B = R/P \tag{1}$$

$$B = Rw_i / \sum_{j=1}^{P} w_j \tag{2}$$

Clearly, the number of iterations allocated to each processor reduces as processors finish execution of a block. MGSS is more suitable with heterogeneous computing because it starts with largest block size possible and accelerators are better utilized with larger block sizes (Figure 1). GSS was proved to finish executions of processors at the same time in a homogeneous SMP system. With accurately estimated computational weights, proof for the original GSS is valid for the MGSS. This is due to the fact that different block sizes calculated for different processors for the same remaining number of iterations will always take the same amount of time because computational weights cause proportional distribution of blocks. In other words, w_i / $\sum_{j=1}^{P} w_j$ will replace 1/P in the equation if the weights are accurately determined.

3.3. Dependency Resolution

Once a proper block size has been determined for the requesting processor, HDSS also determines the starting index (i.e., iteration number) for the iteration block. For loops with independent iterations (i.e., DOALL loops), the start index is set to the first position among the nonassigned iterations. This choice is made regardless of whether the computation for the previous indicies has completed or not.

On the other hand, if an iteration depends on a data value that is calculated in a previous iteration, HDSS automatically exploits wavefront parallelism by considering the dependency requirements between assigned blocks. To achieve parallelism in such loops, the 2D iteration space is divided into fixed width strides [Ciorba et al. 2006]. Elements in stride are further divided into blocks, each of which is assigned to different processors. Blocks that don't depend each other can be processed in parallel (inter-block parallelism). The computation of a block is carried in "*waves*" where elements lined over a diagonal of that block can be processed in parallel (intra-block parallelism). Whenever an idle processor requests a block for execution, HDSS first looks for a stride that contains iterations whose dependency constraints have been satisfied. Next HDSS determines a block size, as was the case for independent iterations, but limiting the remaining iteration space to those iterations that are ready for execution.

Dependencies are determined by dependency vectors per each application. HDSS runtime is responsible for tracking the dependency between blocks assigned to different processors. When there are no independent blocks to process, processors requesting for blocks are put on hold. Once a block is processed, HDSS lets the next idle processor know that there is a "ready to execute" block available and performs necessary assignments and inter-processor synchronization. Dependency vectors can easily be changed by users by modifying HDSS source related to runtime.

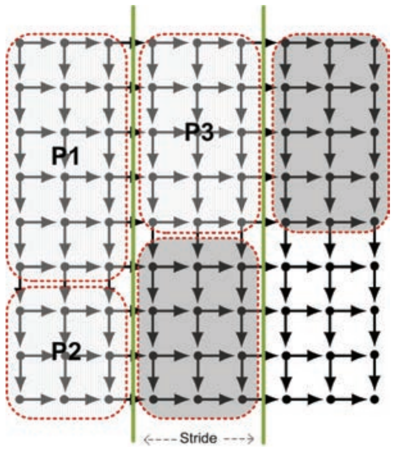


Fig. 9. Wavefront parallelism employed by HDSS: Arrows represent dependencies (target depends on source). Strides are separated by vertical dashed lines. Rectangular areas with light shade represent the blocks that are executed by the labeled processors. Dark shaded blocks represent the iterations whose dependencies are satisfied and ready to execute.

Figure 9 shows an example partitioning of a 2D iteration space divided by strides with size 2. Once the top-leftmost block is processed serially by P1, P2 and P3 execute corresponding blocks in parallel. Once P2 and P3 finish processing, HDSS runtime will pick proper block sizes based on the "ready-to-execute" iterations marked with shaded rectangles in Figure 9. If the block size determined by HDSS is larger than the number of independently executable iterations (data), then the block size is reduced to the available number iterations. As a side note, although HDSS currently supports two-dimensional wavefront parallelism, it can easily be extended to support multiple dimensions of dependences.

4. EVALUATION

4.1. System Configuration

In order to evaluate HDSS, we have used two different systems. The first system is a GPU+CPU architecture with four 16-core AMD Opteron 6200 series CPUs and an nVidia Fermi C2050 GPU. The second system is an SGI Altix 4700 connecting 16 dual-core Itanium Montecito CPUs to Xilinx Virtex 4 FPGA via fast interconnect named NUMALink4.

For the CPU+GPU platform, we have used CUDA SDK to run GP-GPU applications on the device. SGI Altix 4700 features the RASC library to transfer data to/from FPGA and execute applications on the device.

4.2. Applications

For CPU+GPU platform evaluation, we have integrated HDSS into *Blackscholes*, *Histogram*, and *BoxFilter* applications from nVidia CUDA Programming SDK. *Blackscholes* is a popular financial analysis algorithm for calculating prices for European style options. *Histogram* counts the the number of occurrences of 256-bit elements in a given input stream. The third application, *BoxFilter*, applies a filter of a predefined radius to a given image. Data sizes for all three benchmarks are chosen to be the maximum possible amount that a single GPU can store at once, which is around 750 megabytes. For a fair evaluation of our dynamic compute rate calculation, we have scaled the input with different data instead of simply replicating a subset of the input.

In addition to these three GPU applications, we have also implemented a CUDA version of 2D heat equation (*Heat2D*) and string matching (*StrMatch*) to provide examples for scheduling loops with dependent iterations. *Heat2D* simulates propagation of heat on a number of time steps. In each time step, heat propagation of each element depends the left&up elements in current step and right&bottom elements in the previous step. Therefore dependency on top-left elements should be resolved before proceeding. *StrMatch* is based on Smith-Waterman algorithm and it matches two given vectors using an internal 2D array of similarity values. Calculation of each value depends on top&left values.

For CPU+FPGA platform, we have developed FPGA versions of *Blackscholes* and *Histogram* applications. The amount of data to be sent to FPGA at once is limited around 250 megabytes by the RASC library.

Since both of our CPU+GPU and CPU+FPGA platforms are shared memory systems, we have modified each application to execute CPU- and device (GPU or FPGA)-specific implementations concurrently on different parts of the same shared input data.

4.3. Algorithms Compared

We have compared HDSS with the most recent major studies on heterogeneous load balancing: Axel [Tsoi and Luk 2010], Qilin [Luk et al. 2010], and Monte-Carlo [Tse et al. 2010]. Axel and Qilin use static load balancing whereas the Monte-Carlo method uses dynamic scheduling to balance the workload.

For Axel, we have developed communication and computation models for GPU and CPU. These models are determined based on the architectural specifications of processors. The inverse ratios of these values give relative computational weights for each processor in the system that we have used to distribute the workload.

Qilin requires an offline training period to develop a linear model of performance for increasing block sizes. We have performed two runs, the initial being 30% of the original, for each application. As suggested by the technique, during the first (training) run, we fitted a linear curve for the measured execution times for varying sizes of input. The slope of this curve gave the compute ratio for each processor. Later in the second (actual) run, the workload is partitioned statically based on these computed ratios. For a fair evaluation, we have excluded the training run while calculating the execution time for Qilin and comparing with our results.

The last study by Tse et al. [2010] based on the Monte-Carlo method offers the use of two basic dynamic scheduling schemes: *exponential* and *linear incremental* self-scheduling. *Exponential incremental* policy multiplies the block sizes by a factor of a constant m. Every time when a processor requests a new block, it gets m times more workload than a previous request. Similarly, *linear incremental* scheduling increases the block sizes with a constant n. In both cases, the increase in block sizes is more rapid for faster processors, since they request more blocks than a slow processor during the same amount of time. Although these two schemes are not unique to the study mentioned, authors claim that they offer good dynamic balancing for heterogeneous architectures. Hence, we also implemented them and obtained their results for comparison.

4.4. Results

Speedup on GPU. Table I gives reference execution times (in seconds) of single-threaded runs of applications on a single CPU core only and also on the device only (FPGA or GPU). During these runs, we have not applied any load balancing techniques. All iterations are sent to device at once without any executions by multiple CPUs.

Figure 10 shows speedups for a CPU+GPU system with regards to single-threaded CPU execution. Results are obtained by running each algorithm for varying number of representative threads: 1 thread for GPU plus 1,3,7,15,31, and 63 CPU representative

Table I. Base Execution Times

	1-CPU Only	Device Only
Blackscholes (GPU)	169.8	4.7
Histogram (GPU)	133.4	1.82
BoxFilter (GPU)	346.1	4.25
Heat2D (GPU)	592.7	124.6
StrMatch (GPU)	447.5	91.4
Blackscholes (FPGA)	246.8	15.0
Histogram (FPGA)	155.2	25.1

Base Execution Times: Device and 1-CPU only execution times (in seconds) for each application.

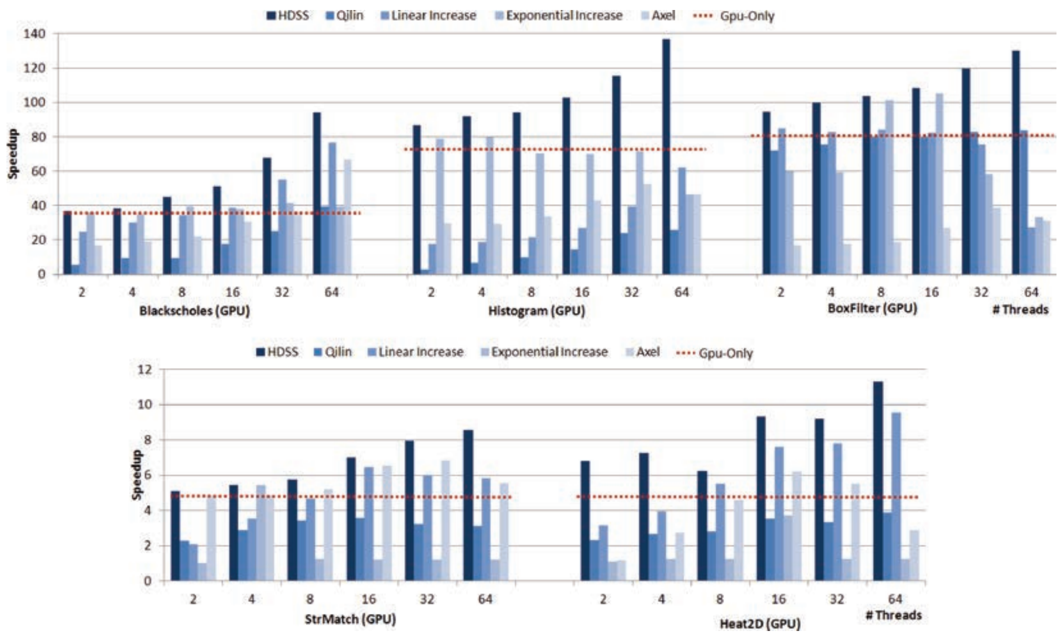


Fig. 10. Speedups obtained in CPU+GPU system when compared to single-threaded CPU-only execution. Varying number of threads 2, 4, 8, 16, 31 and 64 refers to 1 GPU + 1 CPU, 1 GPU + 3 CPU, 1 GPU + 7 CPU, 1 GPU + 15 CPU, 1 GPU + 31 CPU, and 1 GPU + 63 CPU representative threads respectively. GPU-only execution is also shown separately for each application.

threads for 2-,4-,8-,16-,32-, and 64-threaded executions respectively. All CPU and device threads of each run are configured to share the same input but process different parts of it. Speedup for GPU-only execution is also included for reference. Please note that, although we are using a single CPU-side representative thread to handle GPU execution operations, GPU still runs the application with all available hardware threads inside. In other words, by using more CPU threads in addition to the GPU representative thread, we are improving GPU-only results by including additional CPUs in the computation as well.

It may be observed that HDSS is the only algorithm that provides more speedup than GPU-only execution for all applications and thread counts. This is an essential goal of heterogeneous computing which tries to utilize all available processors in the system. Other algorithms fail to do better than even GPU-only execution for most cases because of high GPU idle periods. HDSS results in performance improvements of up to 219% (for 64-threaded execution of the Histgoram application).

Static scheduling schemes, Axel and Qilin, usually perform poorly due to under- (or over) estimation of weights. Qilin's linear adaptation results in inaccurate weights which fail to represent actual computation power of GPUs. Qilin starts from an initial block size and increases it gradually to build a linear-time function based on varying block sizes. However, if the initial block size and subsequent measurements are small enough to underutilize the accelerator (as described in Section 2.1), the linear adaptation will end up in a misbalanced workload ratio. On the other hand, HDSS follows a logarithmic curve fitting approach, which characterizes the relationship between block size and execution time much better than a linear curve.

The other static scheduling scheme, Axel, achieves speedups close to GPU-only execution, since theoretical calculations formulate GPU faster due to its massive number of cores (240 in our case). Modeling CPUs, on the other hand, is not straight-forward due to nonquantifiable capabilities (such as instruction-level parallelism, hardware threads, speculation, etc.) of these architectures. As a result, CPUs are left idle most of the time because of the inaccuracy of static performance estimation. Moreover, this method requires manual adaptation of communication and computation models for each benchmark, which makes the technique very difficult to apply to a range of applications.

Linear and exponential incremental scheduling are the two dynamic schemes that we have compared with HDSS. Exponential incremental does not provide more speedup than GPU-only execution in most cases because growing blocks become too large towards end of the execution. The last large block assigned to CPUs takes more time to process than the one assigned to GPU, hence causing the accelerator to sit idle until CPU finishes execution.

On the other hand, linear incremental scheduling seems to perform surprisingly better than others. Since block sizes are linearly increased, they do not grow up very quickly. This behavior of linear incremental scheduling decreases the chance of accelerators being idle at the end, due to quick processing of small blocks by the CPU cores. However, since linear incremental scheduling slowly increases block sizes, it assigns too many small blocks during the initial phases of the execution. As described earlier, GPUs fail to unleash their full power for small block sizes, even though they are always kept busy all the time. In our experiments, Heat2D and StrMatch are the algorithms employing dependent loop iterations and inter-block synchronization. As the results in Figure 10 show, although HDSS shows better performance than others, it fails to exploit major speedup for this application. This is due to the long idling periods of processors where they wait for other parts of the data to be computed. HDSS performance on dependent loop iterations can be improved by employing priority distribution queues where faster processors are preferred for larger chunks of "ready-to-execute" iterations, however, this is left as a future work.

Speedup on FPGA. Figure 11 shows speedups for SGI Altix CPU+FPGA system when compared to single-threaded CPU execution. Results are very similar to the ones obtained on the GPU system. The logarithmic behavior of block sizes shown in Figure 1 versus execution time is also observable for this platform as well. FPGA is also utilized better when the amount of task executed at once gets bigger. This is mainly due to overlapping of execution time with the communication time in the SGI Altix architecture.

Initial block assignments. For a better exploration of how HDSS is able to use less number of small sized blocks in the beginning, when compared to linear and exponential incremental schemes, Figure 12 is given to illustrate initial block assignments for the three dynamic scheduling algorithms. The experiments are derived from two-threaded execution of Histogram on CPU+GPU platforms given in Figure 10.

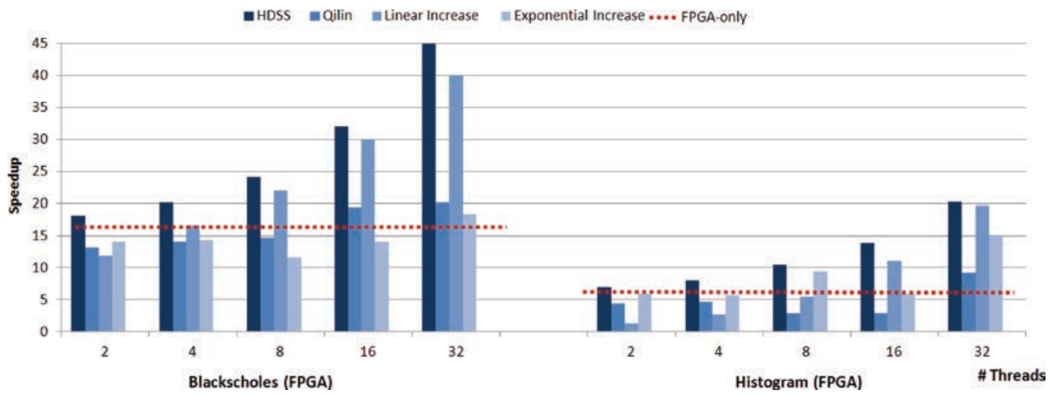


Fig. 11. Speedups obtained in SGI Altix system when compared to single-threaded CPU-only execution. Varying number of threads 2, 4, 8, 16, and 32 refers to 1 FPGA + 1 CPU, 1 FPGA + 3 CPU, 1 FPGA + 7 CPU, 1 FPGA + 15 CPU, and 1 FPGA + 31 CPU threads respectively. FPGA-only execution is also shown separately for each application.

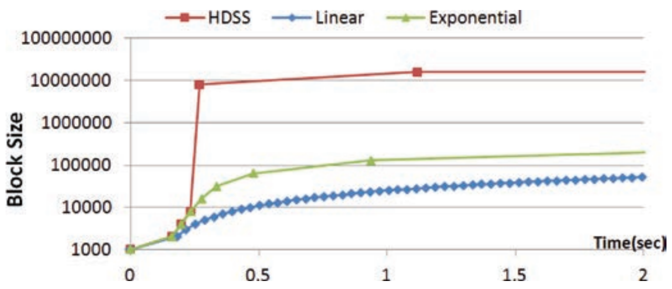


Fig. 12. Illustration of initial block sizes for HDSS, linear and exponential increase algorithms for the 2-threaded execution of Histogram on CPU+GPU platform.

The initial block sizes are set as 1024 and only the assignments during the first two seconds are shown. Stable compute ratios are achieved when the block size is around 2.7M, so blocks having sizes smaller than this number are considered as “small”.

Exponential increase doubles the block sizes consecutively until end of the execution. Similarly linear increase adds 1024 to the previous block size on every assignment. Both of these two algorithms use a considerable amount of small blocks until they reach 16.7M. On the other hand, the HDSS algorithm runs the least squares estimator after the first four block assignments. Based on the resulting logarithmic curve, the block sizes are increased from 8192 to 8M and all potential small blocks that exponential and linear schemes have assigned are skipped. HDSS achieves the stable block size (16.7M) on the fifth block just after one second, whereas exponential increase reaches stability in 3 seconds (not shown) and linear increase never reaches that large block size. In other words, all of the blocks assigned by the linear increase method are considered as “small” blocks which cause underutilization of the accelerators. On the other hand, HDSS is able to quickly converge to the stable large size without spending time on small sized blocks.

Load balancing. For a better understanding of the load balancing efficiency of the algorithms, we have also plotted the time difference between the earliest and latest finishing threads for each run in the CPU+GPU system (Figure 13). The results for HDSS and linear incremental are indistinguishable because they line up in the x-axis along the zero difference line. HDSS and linear have the smallest amount of difference

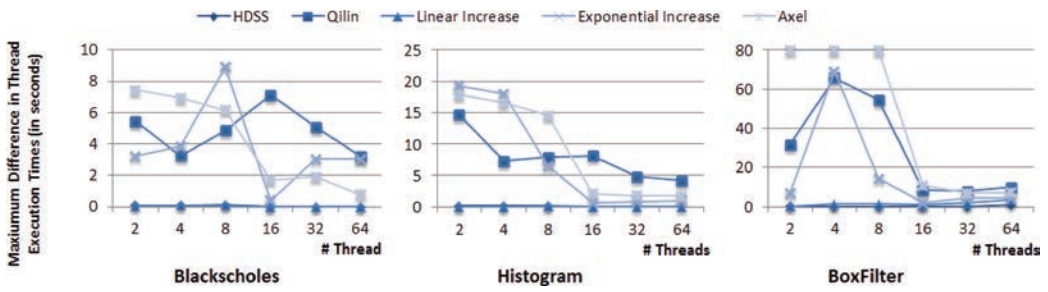


Fig. 13. Time differences between the earliest and latest finishing threads for each run per application in CPU+GPU system.

among all the algorithms implying perfect load balancing of processors. Other algorithms show considerable differences mostly because of the idle GPU at the end of execution. The load is not properly balanced to address the heavy computational weight of the GPU compared to a CPU. Even though the linear incremental policy shows very minimal difference, accelerators are underutilized due to excessive usage of small sized blocks, as previously shown in Figures 10 and 11.

Please also note that Heat2D and StrMatch are excluded from this experiment since idle periods in the lifetime of a thread are distributed over the entire execution due to unsatisfied dependencies, therefore it is not possible to quantify load balancing for these two applications using this kind of experiment.

Workload partitioning. In order to further demonstrate how HDSS achieves close to ideal workload balancing, we have compared the percentage of the blocks assigned to the accelerator (GPU and FPGA) with the percentage that ideal partitioning would assign to that device. In this example, the ideal partitioning is determined in the following way. First the applications are executed with the same inputs separately on a single-core CPU and the accelerator. After obtaining the execution times for both cases, compute ratios (iterations per unit time) are calculated for the accelerator and CPU. The CPU processing rate is scaled by the number of CPU cores (1 to 63) used in the system. Based on the ideal&scaled compute ratios, the ideal percentage of accelerator workload is calculated and indicated by connected points in the chart. Actual workload balancing between the accelerator and CPU produced by HDSS is drawn as bars in Figure 14, where the dark bar represents the accelerator percentage. The results clearly show that HDSS is performing very close to the ideal partitioning, without requiring an offline run or configuration. A notable observation is that, since scalability of two dependent-loop applications, Heat2D and StrMatch, are lower compared to others, HDSS distributed more workload to GPU than the ideal partitioning curve. This is an expected result for dependent-loop applications.

5. ADDITIONAL RELATED WORK

Self-scheduling is a simple form of dynamic scheduling, where there is no central scheduling unit. Instead, each processor decides how much load to grab from a shared pool of iterations. Self-scheduling is effective for executing independent iterations of a loop across multiple processors in parallel.

Most notable self-scheduling schemes are described in Ciorba et al. [2006]. These schemes try to find the ideal block sizes so that processors finish their execution at nearly the same time while using the least possible number of blocks.

For load balancing in distributed systems, a common solution is to assign blocks according to a weight factor representing the processing speed of each processor. Early approaches used fixed weight factors determined at compile time with limited success

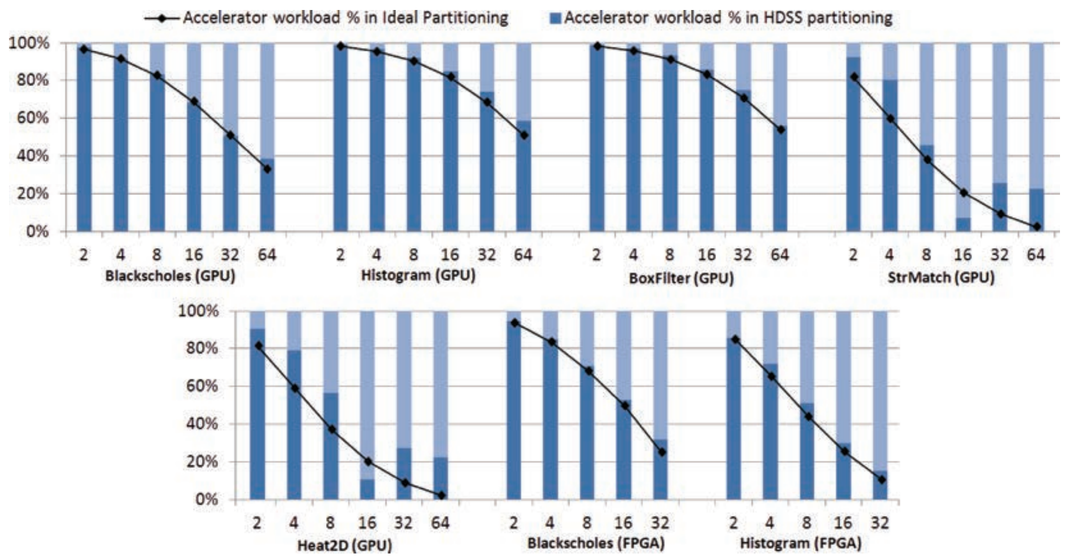


Fig. 14. Comparison of accelerator workloads in ideal and HDSS partitioning. Ideal values are scaled from single-core CPU and accelerator-alone executions.

[Hummel et al. 1996]. Variations in processor speed lead to adaptive schemes which measure weights and change the workload during runtime. Adaptive weighted factoring [Banicescu and Velusamy 2001] cumulatively updates computational weights per each processor after each block assignment. Similarly, existing self-scheduling algorithms have also been modified to adjust the block sizes dynamically based on measured weight factors [Yang et al. 2005, 2008; Chronopoulos et al. 2001, 2006].

On the other hand, heterogeneous multicore architectures bring CPU cores with different sizes and speeds together to provide a hybrid execution environment. These systems dynamically determine which core is most suitable for a given task and make scheduling decisions accordingly. As examples, Li et al. [2010] provide operating-system-level scheduling mechanisms for assigning tasks to heterogeneous cores whereas Lee et al. [2010] provide frameworks and algorithms for workload balancing in such systems. Very few studies present compiler-assisted approaches for better load balancing in heterogeneous systems. Among those, Yeung et al. [2008] employ a map-reduce-based scheduler embedded into the application during compile time. However, limited applicability of the map-reduce pattern on scientific applications and increased programming effort makes such an approach infeasible.

6. CONCLUSION AND FUTURE WORK

In this study we have proposed a new scheduling and workload balancing algorithm, HDSS, for loops with independent or dependent iterations on heterogeneous multiprocessor systems. The algorithm runs in two phases. The first phase is dedicated for learning accurate computational weights for each processor and the second phase distributes the remaining workload using computed weights. Our algorithm uniquely considers the impact of block sizes on performance on heterogeneous multiprocessor systems without an offline run or a prior knowledge of application or architecture specifications.

We have evaluated and tested our algorithm on two different heterogeneous systems, nVidia FERMI (CPU+GPU) and SGI Altix (CPU+FPGA), for 7 benchmarks in total. We have also compared our results with recent major studies on static and dynamic scheduling of heterogeneous multiprocessors. Results showed that our algorithm gives best performance in all cases and provides improvements up to 219% when compared to its closest load balancing scheme on certain applications. In our results, HDSS algorithm caused processors to finish their execution near to each other and the workload partitioning was very close to the ideal partitioning. Our results also showed that, although some dynamic schemes (e.g., linear incremental) are able to balance workload properly, they were not giving the same speedups as HDSS due to underutilization of accelerators with small block sizes. This important observation showed that keeping processors busy all the time is not enough by itself for the ideal workload balancing. Processor capabilities should also be fully utilized by giving the right amount of work.

We are planning to extend our experiments with 2 or more GPUs and also on more recent GPUs and FPGAs like nVidia Kepler and Virtex 6. We will also improve support for dependent loops and add more applications that employ such loops.

REFERENCES

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P. 2009. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *Proceedings of the International European Conference on Parallel Processing (Euro-Par'09).* 863–874.

Barker, Z. and Prasanna, V. 2005. Efficient hardware data mining with the apriori algorithm on fpgas. http://gridsec.usc.edu/files/TR/zbakerUSCfccm05.pdf.

Banicescu, I. and Velusamy, V. 2001. Performance of scheduling scientific applications with adaptive weighted factoring. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*. IEEE, 791–801.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., and Skadron, K. 2008. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput. 68*, 10, 1370–1380.

Chronopoulos, A., Benche, M., Grosu, D., and Andonie, R. 2001. A class of loop self-scheduling for heterogeneous clusters. In *Proceedings of the Internatioanl Conference on Cluster Computing*. IEEE Computer Society, 282.

Chronopoulos, A., Penmatsa, S., Xu, J., and Ali, S. 2006. Distributed loop-scheduling schemes for heterogeneous computer systems. *Concur. Comput. Pract. Exper. 18*, 7, 771–785.

Ciorba, F., Andronikos, T., Riakiotakis, I., Chronopoulos, A., and Papakonstantinou, G. 2006. Dynamic multi phase scheduling for heterogeneous clusters. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS'06).* IEEE.

de Ruijsscher, B., Gaydadjiev, G., Lichtenauer, J., and Hendriks, E. 2006. FPGA accelerator for real-time skin segmentation. In *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*. 93–97.

Fahey, M., Alam, S., Dunigan, T., Vetter, J., and Worley, P. 2005. Early evaluation of the cray xd1. In *Proceedings of the Cray User Group Meeting*. 12.

Galanis, M., Milidonis, A., Theordoridis, G., Soudris, D., and Goutis, C. 2005. A partitioning methodology for accelerating applications in hybrid reconfigurable platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'05).* Vol. 3. IEEE Computer Society, 247–252.

Harris, B., Jacob, A., Lancaster, J., Buhler, J., and Chamberlain, R. 2007. A banded Smith-Waterman fpga accelerator for mercury BLASTP. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'07).* IEEE, 765–769.

Hermann, E., Raffin, B., Faure, F., Gautier, T., and Allard, J. 2010. Multi-GPU and multi-cpu parallelization for interactive physics simulations. In *Proceedings of the International European Conference on Parallel Processing (Euro-Par'10).* 235–246.

Hummel, S., Schmidt, J., Uma, R., and Wein, J. 1996. Load-Sharing in heterogeneous systems via weighted factoring. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM Press, New York, 318–328.

Lee, J., Lee, J., Seo, S., Kim, J., Kim, S., and Sura, Z. 2010. Comic++: A software svm system for heterogeneous multicore accelerator clusters. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA'10).* IEEE, 1–12.

Li, T., Brett, P., Knauerhase, R., Koufaty, D., Reddy, D., and Hahn, S. 2010. Operating system support for overlapping-isa heterogeneous multi-core architectures. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA'10).* IEEE.

LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro 28*, 2, 39–55.

LUK, C., HONG, S., AND KIM, H. 2010. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. IEEE, 45–55.

POLYCHRONOPOULOS, C. AND KUCK, D. 1987. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput. 100*, 12, 1425–1439.

SCROFANO, R., GOKHALE, M., TROUW, F., AND PRASANNA, V. 2007. Accelerating molecular dynamics simulations with reconfigurable computers. *IEEE Trans. Parallel Distrib. Syst.*, 764–778.

SGI. 2008. Sgi altix 4700. Delivering new levels of performance and flexibility. http://www.sgi.com/products/servers/altix/4000/index.html

SMITH, R., GOYAL, N., ORMONT, J., SANKARALINGAM, K., AND ESTAN, C. 2009. Evaluating gpus for network packet signature matching. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*. IEEE, 175–184.

TRIPP, J., HANSON, A., GOKHALE, M., AND MORTVEIT, H. 2005. Partitioning hardware and software for reconfigurable supercomputing applications: A case study. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 27.

TSE, A., THOMAS, D., TSOI, K., AND LUK, W. 2010. Dynamic scheduling monte-carlo framework for multi-accelerator heterogeneous clusters. In *Proceedings of the International Conference on Field Programmable Technology (FPT'10)*. IEEE, 233–240.

TSOI, K. AND LUK, W. 2010. Axel: A heterogeneous cluster with fpgas and gpus. In *Proceedings of the 18th Annual ACM SIGDA International Symposium on Field Programmable Gate Arrays*. ACM Press, New York, 115–124.

TSOI, K., TSE, A., PIETZUCH, P., AND LUK, W. 2011. Programming framework for clusters with heterogeneous accelerators. *ACM SIGARCH Comput. Archit. News 38*, 4, 53–59.

YAMANOUCHI, T. 2007. AES encryption and decryption on the gpu. *GPU Gems 3*, 785–803.

YANG, C., CHENG, K., AND LI, K. 2005. An enhanced parallel loop self-scheduling scheme for cluster environments. *The J. Supercomput. 34*, 3, 315–335.

YANG, C., SHIH, W., AND TSENG, S. 2008. Dynamic partitioning of loop iterations on heterogeneous pc clusters. *The J. Supercomput. 44*, 1, 1–23.

YEUNG, J., TSANG, C., TSOI, K., KWAN, B., CHEUNG, C., CHAN, A., AND LEONG, P. 2008. Map-Reduce as a programming model for custom computing machines. In *Annual Symposium on Field Programmable Custom Computing Machines (FCCM'08)*. IEEE.

Received June 2012; revised November 2012; accepted November 2012