Dynamic Scheduling for Work Agglomeration on Heterogeneous Clusters

Jonathan Lifflander, G. Carl Evans, Anshu Arya, and Laxmikant V. Kale
Dept. of Computer Science
University of Illinois
Urbana-Champaign, United States
Email: {jliffl2, gcevans, arya3, kale}@illinois.edu

***Abstract*—Dynamic scheduling and varying decomposition granularity are well-known techniques for achieving high performance in parallel computing. Heterogeneous clusters with highly data-parallel processors, such as GPUs, present unique problems for the application of these techniques. These systems reveal a dichotomy between grain sizes: decompositions ideal for the CPUs may yield insufficient data-parallelism for accelerators, and decompositions targeted at the GPU may decrease performance on the CPU. This problem is typically ameliorated by statically scheduling a fixed amount of work for agglomeration. However, determining the ideal amount of work to compose requires experimentation because it varies between architectures and problem configurations.**

This paper describes a novel methodology for dynamically agglomerating work units at runtime and scheduling them on accelerators. This approach is demonstrated in the context of two applications: an n-body particle simulation, which offloads particle interaction work; and a parallel dense LU solver, which relocates DGEMM kernels to the GPU. In both cases dynamic agglomeration yields comparable or better results over statically scheduling the work across a variety of system configurations.

***Keywords*-dynamic scheduling; accelerator; GPGPU; CUDA; agglomeration; adaptive runtime**

I. Introduction

Parallel applications often use techniques such as dynamic work scheduling [1]–[3] and varying decomposition granularity [4] to obtain high performance. A fine decomposition may yield higher performance when executed on a CPU because of increased communication and computation overlap or improved load balancing [5]. For applications with high memory pressure, it allows the program to incrementally free memory as appropriate. Some applications are naturally expressed with a fine decomposition, and it may be beneficial to preserve that decomposition if performance is not sacrificed.

Modern clusters are often augmented with highly data-parallel accelerators, such as GPGPUs. Prominent examples of supercomputers that use accelerators include Tianhe-1A, Nebulae, and TSUBAME 2.0, which are currently in the top 5 of the Top500 [6] rankings. The accelerators used in these computers are NVIDIA GPUs, which are similar to the accelerators used in this paper. These accelerators, which often execute thousands of threads, must have ample work to perform efficiently. To obtain high efficiency and utilization, the computational grain size must be sufficiently data-parallel and suitable for the accelerator model. In general, accelerators have low processing time due to their massive parallelism, but high overhead due to data copying and startup costs. In contrast, CPUs have relatively low overhead because of data locality and high processing time.

There is an intrinsic dichotomy between these two paradigms: work units sized for performance and expressibility on the CPU may yield insufficient data-parallelism for accelerators, and decompositions targeted at accelerators may inhibit precise load balancing and decrease the overlap of communication and computation on the CPU.

The description of our methodology uses the following terms. An *agglomeration* is a composition of distinct work units that may be independently dispatched or combined. *Static agglomeration* is a method in which a fixed number of work units are agglomerated. Conversely, *dynamic agglomeration* allows the number of work units agglomerated to vary during execution.

Many applications use static scheduling for the accelerator to ameliorate these problems. In static scheduling, a fixed amount of work is composed, and then offloaded to an accelerator. NAMD [7], a molecular dynamics simulation, statically agglomerates patch-pair calculations in a single compute object for the GPU. In NAMD, the grain sizes between CPU and GPU may vary by multiple orders of magnitude (sometimes more than 10,000 work units are agglomerated for each kernel launch). FLAME [8] on the GPU batches messages for work agglomeration to obtain high performance. ChaNGa [9] uses a work agglomeration scheme to combine work from single work requests. Husbands and Yelick [10] use data coalescing to obtain higher DGEMM performance. In all cases, the best agglomeration threshold is found empirically.

Static scheduling is deficient because it is difficult to determine the ideal amount of work to combine; it depends on the problem size, decomposition, and the system architecture of both of the host and the accelerator. In many cases it will even vary at runtime. Therefore, the application must be tuned for many different scenarios. Moreover, for very dynamic problems, where the amount of work varies unpredictably over time, a static schedule may be insufficient for obtaining optimal utilization of both the host and accelerator.

This paper describes a novel methodology for agglomerating work units dynamically at runtime and scheduling the agglomerated work units on accelerators. In contrast to static schemes,

our methodology is portable across architectures and adaptable between problem configurations while achieving comparable or higher performance. This technique is demonstrated in the context of two applications: an n-body particle simulation, which offloads particle interaction work; and a parallel dense LU solver, which relocates DGEMM kernels to the GPU.

II. Analysis of Agglomeration

The performance benefits of agglomeration arise from amortization of offloading overhead and higher utilization of the accelerator. The increase in performance is impeded by the cost of performing the agglomeration, composing the work in an efficient data structure for the accelerator, and executing the scheduler. Some of these benefits can also be achieved by overlapping accelerator work with CPU work. This type of overlap was not performed in this work to illustrate the benefits of agglomeration.

If, for a given application, the accelerator is sufficiently utilized and the overhead of offloading to the accelerator is negligible, overall throughput will not improve. In practice, a considerable amount of tuning effort must be applied to effectively utilize the accelerator.

Depending on the overhead of spawning an accelerator kernel, some types of work may not be suitable for offloading. For instance, if a small work unit is on the critical path and the overhead of instigating a kernel is high, the work may not be suitable for agglomerating or even offloading to an accelerator. Although agglomerating amortizes the overhead cost, it may delay the critical path if a significant amount of work is agglomerated before executing on an accelerator.

A significant portion of offloading overhead may originate from copying data to and from the accelerator. The benefit gained from the accelerator is often proportional to the ratio of work to data, a computation versus communication tradeoff. Thus the ratio can be used as an indication of the practicality of offloading. Although agglomeration decreases overhead and improves utilization, it does not affect this ratio. There are several application-specific optimizations that may be performed to reduce data copying when agglomeration is used, but in most cases the data-transfer per computation remains constant with agglomeration.

Preliminary experimentation demonstrated that if the work-to-data transfer ratio is not sufficiently high, offloading to the accelerator is not beneficial due to data copying overheads. In practice, the accelerator is beneficial when the work grows at a rate that is higher than the amount of data. Otherwise, offloading to an accelerator may decrease overall throughput compared to executing exclusively on the CPU, regardless of the agglomeration strategy. Other strategies are required to observe speedups using accelerators in the $O(work) \leq O(data)$ regime.

The applications detailed in section V, have asymptotically more work than data (i.e. $O(work) > O(data)$), implying that offloading to an accelerator may improve performance. However, as the amount of work per data increases even further, the benefits of agglomeration diminish because the accelerator is used more efficiently. Despite the limited region of effectiveness, speedups are observed by agglomerating work.

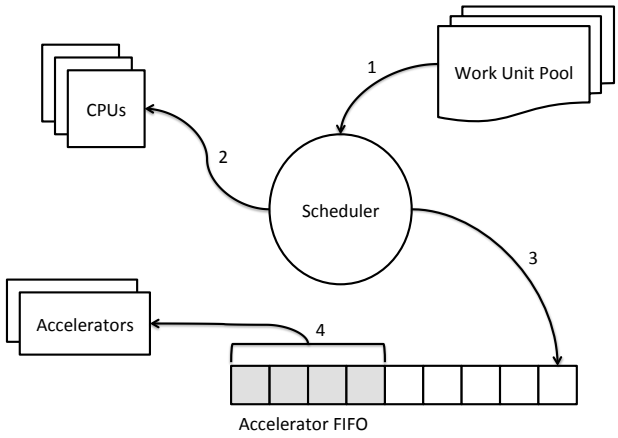

Fig. 1. High level system diagram. (1) Fetch highest priority work unit. (2) If work unit has no agglomerating kernel dispatch to CPU else (3) enqueue in accelerator FIFO. (4) If no high level work dispatch maximum size batch from accelerator FIFO to an accelerator.

III. Methodology

A. Approach

In the proposed system, depicted in figure 1, all work is decomposed into work units that are scheduled when processing time is available. A program's natural decomposition usually defines an appropiate work unit size. This technique is used in many systems such as TBB [3], Charm++ [11], and Concurrent Collections [12]. For each work unit that can be run on an accelerator, the programmer provides a separate agglomerating kernel that executes several work units of the same type in one kernel invocation.

Work units can have a priority that is specified by the programmer. The system maintains a scheduler that executes the next work unit from the pool of available work with respect to the defined priorities. Work that has an agglomerating kernel is placed in an accelerator FIFO rather than being executed immediately.

Priorities are partitioned into two classes: "high" and "low" priority. When the scheduler finds no high priority work and a non-empty accelerator FIFO, it will dispatch the maximal amount work from the FIFO that can be executed in a single agglomerating kernel using a CPU to manage the execution. All of these work units will need to use the same agglomerating kernel so that they can be executed in one invocation.

The division of work units into "high" and "low" classes is the programmer's responsibility. As a simple guideline, critical path work and methods that generate accelerator work should be high priority. The underlying scheduler may support more than two classes of priorities so the programmer may refine the priorities within these classes based on application specifics.

B. Reasoning

The intuitive idea is to maintain high CPU utilization while high priority work is available. When there is a reprieve in high priority work, the scheduler exploits idle time by agglomerating work units for the accelerator.

To efficiently utilize the accelerator and reduce overhead, the number of agglomerated work units, or the *agglomeration size*, varies depending on the rate that high priority work is generated. The agglomeration size becomes large with a high generation rate, and small with a low generation rate. This is appropriate because a high generation rate indicates a fine grain size, implying that more work units must be agglomerated to reduce accelerator overhead.

Moreover, a high generation rate suggests that the critical path is not immediately dependent on the accelerator work being generated. Therefore, waiting for more work versus scheduling immediately will likely be beneficial.

IV. Implementation

A. Charm++

The scheduler is written in Charm++ [11], [13], a parallel object-oriented extension to C++. Charm++ is a message-driven execution programming framework and runtime system that decomposes the application into objects and maps them to processors. This enables the runtime system to load balance work by moving objects and overlapping communication and computation.

As the problem is decomposed, it is divided into communicating objects instead of processors. The number of objects represents the amount of virtualization in the application. If the number of objects is less than or equal to the number of processors, there is no effective virtualization.

Every Charm++ object is a C++ object with methods. Some methods may be declared as *entry methods*, which have special semantics that allow remote asynchronous invocation by sending a message. When a message is received by the Charm++ runtime, it is placed in a processor queue and delivered with respect to message priorities. Delivery causes an entry method to execute.

B. Important Runtime Features

The following features in Charm++ made it straightforward to implement the scheduler efficiently.

Charm++ has a priority queue of messages on each CPU, which is used as the pool of work units described in section III. Hence, a message is a work unit. Moreover, to optimize the process of offloading to the accelerator, if a Charm++ method is executed locally on a processor, pointers may be sent in the message. This functionality enables us to limit the data copying performed by the scheduler.

C. Scheduler

The scheduler has the following methods:

- `scheduleWork(message)`: the external interface to add work that may be executed on an accelerator to the accelerator FIFO.

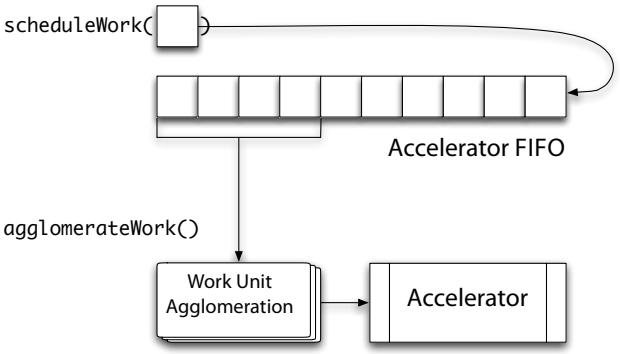

Fig. 2. Depicts the two scheduler functions: `scheduleWork()` and `agglomerateWork()`, and how they interact.

- `agglomerateWork()`: the internal interface used to agglomerate multiple work units and dispatch work to an accelerator.

Messages that may be agglomerated are intercepted by the scheduler and placed in the accelerator FIFO. The scheduler collects work by being called from each CPU. When an object sends a message possibly destined for the accelerator, it calls the `scheduleWork(message)` entry method on the scheduler. This invocation copies data or saves pointers (depending on the data locality and whether the message data is reused) from the message into the scheduler buffer. Figure 2 depicts the control flow and interaction of the two scheduler functions.

Each time a message is enqueued, the scheduler asynchronously sends a message with medium priority (higher than the lower priority CPU work) to invoke the `agglomerateWork()` function. The `agglomerateWork()` function then agglomerates the first n messages in the accelerator FIFO and dispatches the agglomerated work to an accelerator, where n is constrained by hardware (e.g. memory) or application limits. Specific application-enforced limits are discussed on a per application basis in sections V-A and V-B.

By invoking `agglomerateWork()` with medium priority, work naturally accrues in the accelerator FIFO. If the size of the FIFO was more than n and there is still work in the queue, the `agglomerateWork()` function sends a message to call itself asynchronously with medium priority. The recursive asynchronous invocation ensures that queued work will always complete.

One scheduler is instantiated per CPU core so agglomeration is done on a per-CPU basis. Hence, every work unit on each CPU is intercepted by a local scheduler, where it deposits different types of work that may be offloaded. This design decreases the amount of agglomeration because work across processors is not combined; however, due to the locality of the scheduler, the overhead of intercepting work can be significantly reduced by maintaining pointers instead of copying data. Depending on the application's control flow, the scheduler may be required to copy the data if the application

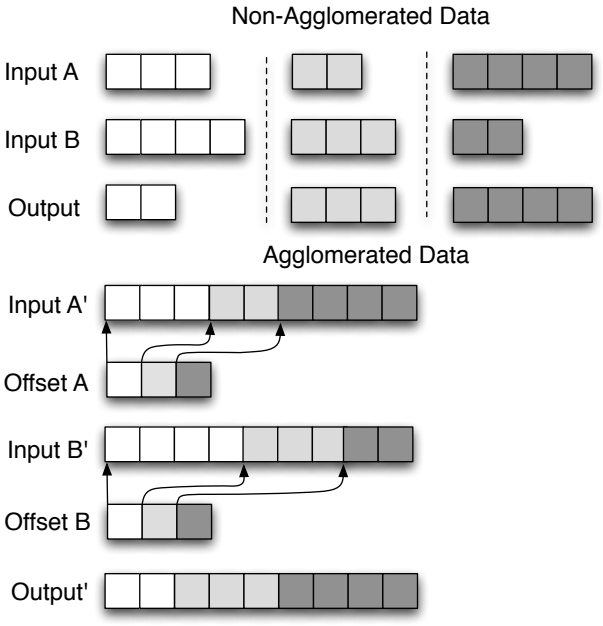

Fig. 3. The shaded regions represent segments of work within one message. The offset array stores the index where each segment starts, so the accelerator thread has efficient access to its segment. The segments are not constrained to be the same size.

changes it concurrently.

D. Accelerator

NVIDIA GPUs were used as the accelerators for the tested applications. The kernels for these accelerators were written in CUDA [14]. Similar languages, such as OpenCL [15], are available for other accelerators.

Given the current state of the art, programs must have kernels written to take advantage of accelerators. This methodology is no different in this respect. However, some extra structure is required in the kernel to handle the agglomeration.

In general, if the messages to be offloaded are data-parallel, control flow must be added to the kernel that directs each thread to its segment of the data. This offset information is stored in offset arrays, which must be copied to the GPU. Figure 3 shows how the offset arrays are used to map the threads to the correct segments of the agglomerated array. These extra arrays are not costly compared to size of the actual data because they are on the order of number of messages, not the size of the message.

The agglomeration step is optimized to copy the data from the messages directly to device pointers on the accelerator. The offset arrays are built by iterating through the messages and copying the data to the accelerator.

The most expensive GPU operation is allocating memory. Due to its high cost, memory is allocated for the maximum dispatch size during application startup. After the initial allocation, only the required data is copied to the GPU from the scheduler. The GPUs on the test platform did not have native support for double-precision floating point operations. Because this paper focuses on agglomeration, not accelerator performance and kernel implementation, all computation was implemented in single-precision.

For both applications studied, the kernels were hand-coded with and without the added loop that handles agglomeration. The kernels are very straightforward and not highly tuned to the architecture. Since absolute highest performance was not the goal of this work, further kernel optimization (or use of an optimized third-party library) could be possibly applied to both variants to increase performance.

V. Results

The test platform's specifications are as follows: each node has two dual-core 2.4 GHz AMD Opterons, 8 GB of memory, one NVIDIA Tesla S1070 with four GPUs, each with 4 GB of memory. All the runs were performed on one node of this cluster. Each CPU core was assigned exclusive access to a GPU.

A. Molecular2D n-Body Particle Simulation

Molecular2D is a two-dimensional n-body particle simulation written in Charm++, which was modified by applying the proposed dynamic agglomeration technique.

The application spatially decomposes the interaction space into $n \times m$ cells where increasing n or m corresponds to creating a finer decomposition. Each cell manages a set of particles and each particle interacts with all other particles within a cutoff radius.

Instead of interacting the particles directly and every cell receiving and sending particles, a set of interaction objects is created that manages all interactions. In this configuration, every cell sends its particles to nine interaction elements. When an interaction element receives two sets of particles, it computes force interactions, and sends messages with updated force information back to the appropriate cells. The cells then update the position information for their resident particles. Particles move to neighboring cells if their calculated position is outside their current cell's domain.

The interaction set is built dynamically and distributed to the processors with a round robin scheme at runtime.

As the grain size changes, this also increases or decreases the amount of work. Because the number of particles is constant, as the size of the cells increase (higher grain size), more cutoff distance checks are required between each pair of neighboring cells (more particles per cell), regardless of the number of particles that actually interact. Therefore, in general, virtualization should help performance up to the limit enforced by the cutoff radius.

1) Adding the Scheduler: To augment this program with the dynamic scheduler, the control-flow of the interaction objects was modified to enqueue the work in the scheduler (by calling `scheduleWork()`) instead of triggering the interactions directly.

In the original program, when an interaction object receives particles, it buffers the first set, and when the second set arrives, it performs the interaction inline without buffering. However, because of the asynchronous nature of the dynamic

```
__global__ void interact(...) {
  int i = blockIdx.x * blockDim.x +
    threadIdx.x;

  // For loop added for agglomeration
  for(int j = start[i]; j < end[i]; j++)
    // interaction work
}
```

Fig. 4. Modifications to the Molecular2D CUDA interaction kernel for agglomeration.

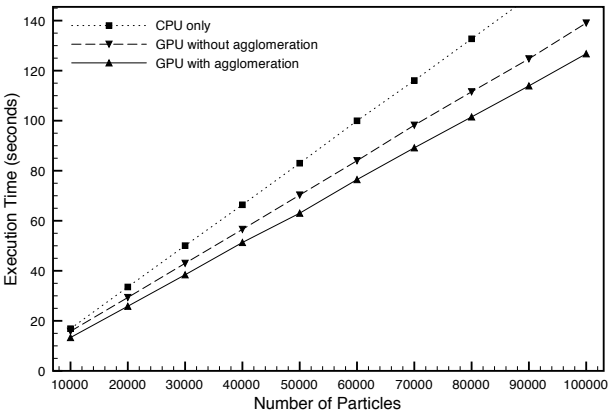

Fig. 5. Molecular2D running only on the CPU, on the GPU with no agglomeration, and on the GPU with agglomeration. Each point is the average of 5 runs. For each point, to establish a fair comparison, the grain size that yielded the best performance was plotted.

scheduler, all particles must be buffered. Because the interaction is no longer performed locally, the scheduler is responsible for forward progress.

The sets of particles are packed into two arrays such that the particles in the first array will interact with one or more sets of particles in the second array, and vice versa. Offset arrays are generated which hold the starting index of each particle set in each array so a GPU thread can access its segment of data efficiently.

The arrays of particles are copied to the GPU and two GPU kernel instances are invoked to interact the arrays with each other. The GPU kernel is the major custom code that must be written for an application to utilize the agglomerating scheduler. Figure 4 shows the necessary additions to the GPU kernel code.

2) Performance: For comparison purposes, a non-agglomerating version of the code was written that invokes a GPU kernel for each cell interaction without the scheduler. This methodology was compared to using a static agglomeration size in section V-A6.

3) Agglomeration and Problem Size: Figure 5 shows the execution time of 100 timesteps of Molecular2D versus problem size. With smaller problem sizes, agglomeration increases performance because the non-agglomerating kernel invocations do not yield enough parallelism to sufficiently utilize the GPU. However, as the problem size increases, the number

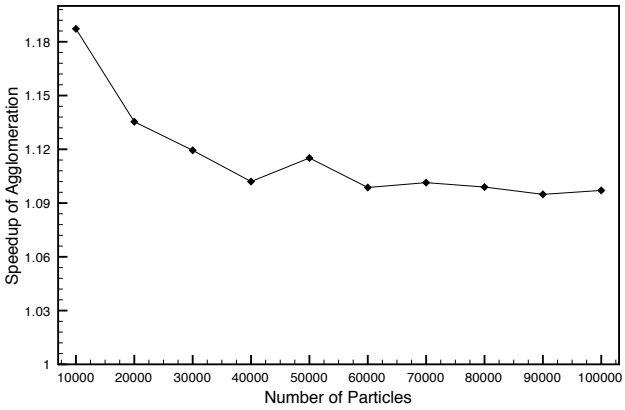

Fig. 6. Speedup with agglomeration relative to no agglomeration for Molecular2D as the problem size increases.

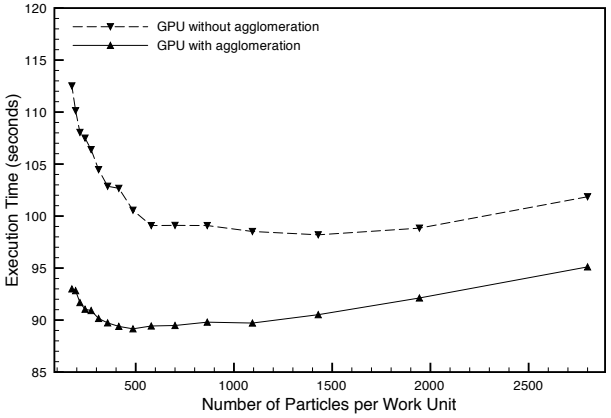

Fig. 7. Execution time versus grain size of Molecular2D for 70,000 particles and 100 timesteps. A lower number of particles per work unit corresponds to higher virtualization.

of threads spawned on the GPU by the non-agglomerating version increases proportionally, decreasing the performance gain that is observed by agglomerating. Performance is still improved because even though the accelerator is well utilized, there is still significant overhead from instigating the kernel. By agglomerating, the overhead is amortized, reducing the the overall time to perform the same amount of work.

Figure 6 is a graph of the speedup realized from agglomeration relative to running a single kernel as the problem size increases. As expected, as the problem size grows, the speedup diminishes because the accelerator is well utilized. Some speedup remains due to amortization of overhead.

4) Agglomeration and Virtualization: Agglomeration and virtualization adjust the grain-size of the computation in opposite directions. However, these seemingly disjoint techniques are very effective when used in conjunction: increased virtualization decreases grain size and increases computation and communication overlap on the CPU; the agglomerating scheduler increases the grain size on the GPU, amortizing the overhead and increasing utilization.

Figure 7 shows the effect of these techniques on the execution time of Molecular2D. At a low level of virtualization, the

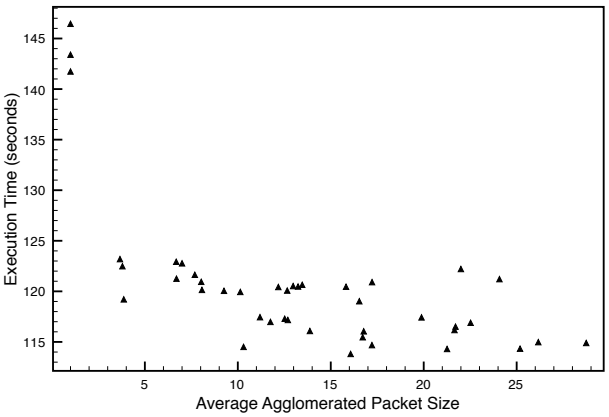

Fig. 8. Execution time versus average agglomeration size for Molecular2D running with 100,000 particles for 100 timesteps. This graph shows the execution time for different maximum agglomeration sizes in the dynamic scheduler.

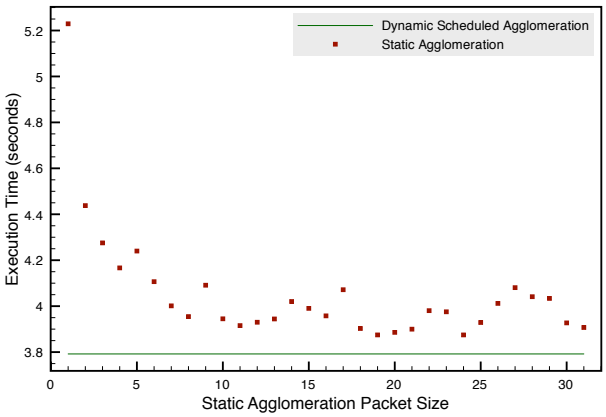

Fig. 9. Static versus dynamic agglomeration. Molecular2D running with 5000 particles, a 5x5 grid, and 50 time steps. Each static data point is the average of 5 runs. The dynamic line is an average of 5 runs also.

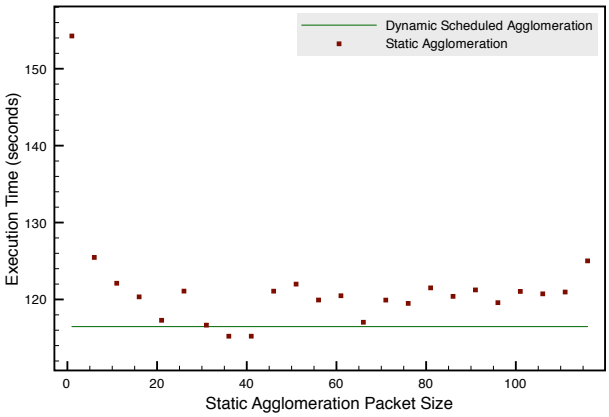

Fig. 10. Static versus dynamic agglomeration. Molecular2D running with 100000 particles, a 16x16 grid, and 100 time steps. Each static data point is the average of 5 runs. The dynamic line is an average of 5 runs also.

execution time is high because overlap is lacking and there is more work.

In both cases, performance increases with increasing virtualization until overheads counteract the benefit of communication and computation overlap and decreasing amount of work. The runs without agglomeration are dominated by GPU invocation overhead because the number of invocations increases proportionally with the number of cells. Runs with agglomeration significantly reduce GPU overhead by reducing the number of invocations. The gradual rise in the agglomeration curve is caused by the increasing overhead of virtualization.

The curves demonstrate different optimal levels of virtualization: 486 particles per work unit with agglomeration and 1429 particles per work unit without agglomeration. Runs with agglomeration achieve their best performance with more virtualization because GPU invocation overhead increases at a faster rate than virtualization overhead, thereby gaining a greater advantage from the decrease in work. This demonstrates that, in addition to amortizing GPU invocation overhead, agglomeration enables more effective use of virtualization.

5) Amount of Dynamic Agglomeration: Figure 8 displays the execution time versus the average agglomeration size for multiple runs of Molecular2D with different maximum agglomeration sizes. As long as some dynamic agglomeration is realized (not the first 3 points, which had a limit of 1) execution time decreases significantly. The number of agglomerated messages also varies due to the effects of dynamic agglomeration and system noise. The data shows that even low levels of agglomeration greatly improve performance; however, the optimal static amount of agglomeration is not clear. Hence, it is important that the agglomeration amount remain dynamic.

There is a large variation in the agglomeration size between runs that have similar execution times. This effect can be attributed to system noise and message order variations between execution instances. The dynamic scheduler adapts to these changes by changing the agglomeration level to attain higher performance.

6) Dynamic vs. Static Scheduling: To compare the performance of dynamic versus static scheduling, a static scheduler was implemented for agglomeration in Molecular2D. Because Molecular2D dynamically maps the interaction objects to processors (using a round robin scheme), there is no static method for determining which objects belong to which processor. In addition, an object migrates to a different CPU for load balancing, the mapping would dynamically change at runtime. Hence, a scheme was implemented where the number of objects assigned to each processor is counted when building the initial mapping. These counts are sent to the appropriate scheduler.

Once every scheduler knows the number of interaction objects it is responsible for, it can use this information to determine the total number of particle interaction messages it will receive for a given iteration. The edge case, when the static limit cannot be reached, can then be handled appropriately by the scheduler.

However, if load balancing were implemented in this application, the scheduler would have to be notified of every migration. Moreover, it would have to flush all the messages it received for that object that persist in the scheduling queue. These messages either need to be executed immediately and a message must be sent to the new scheduler that handles the agglomeration for that object, or the messages for that migrated object must be moved to the other scheduler. Both schemes require the messages to be tagged with the object index, and require extra messages to be sent during migration.

When migration occurs with dynamic agglomeration, the scheduler does not need to be notified. Instead, the program can perform the migration in the normal fashion and any new messages will be sent to the new scheduler. The old scheduler will offload its work at some point, and will then remotely send the resulting data to the newly migrated object.

The static scheduler uses the same basic infrastructure as the dynamic scheduler. They share the GPU offloading code that packs data into arrays and the specialized GPU kernel that handles agglomerated work.

Figures 9 and 10 show the performance of static versus dynamic scheduling. Each static point and the dynamic line are the average of five runs. These figures show that dynamic scheduling always performs close to optimal without any tuning. The figures present drastically different problem configurations, demonstrating the adaptability of this approach. Molecular2D is fairly resilient to non-optimal static scheduling, because if the work is not offloaded soon enough, the CPU still has sufficient work to perform.

B. Linear System Solver using CharmLU

CharmLU [16] is a dense matrix LU decomposition application written in Charm++. This application decomposes an $n \times n$ square matrix into $b \times b$ square blocks. By default, the blocks use a block mapping scheme, but the mapping scheme is independent of decomposition in Charm++ and can be easily modified. For all presented data, the balanced snake mapping scheme from CharmLU was used.

Figure 11 shows the regions that are involved in one step of LU decomposition. At each step, the application calculates the LU factorization of $A_{1,1}$ and then broadcasts U to $A_{2,1}$ and L to $A_{1,2}$. The blocks of $A_{2,1}$ and $A_{1,2}$ then perform a triangular solve and multicast to their respective row or column of $A_{2,2}$ to perform matrix-matrix multiplies (DGEMMs) called "trailing updates." When the updates to the $A_{2,2}$ submatrix are complete, the next step of the algorithm may begin.

The prioritization of the trailing updates is complex. For each step of LU factorization there are trailing updates created for each block in the submatrix. Only the updates required in the next step (the next $A_{2,1}$ and $A_{1,2}$) need to be completed before the algorithm can proceed. This is enforced with priorities so work on the critical path will be completed before the next step.

After LU decomposition is complete, the application performs a distributed backward and forward solve to compute $Ax = b$. When the solving step is complete, the result is verified by calculating the residuals.

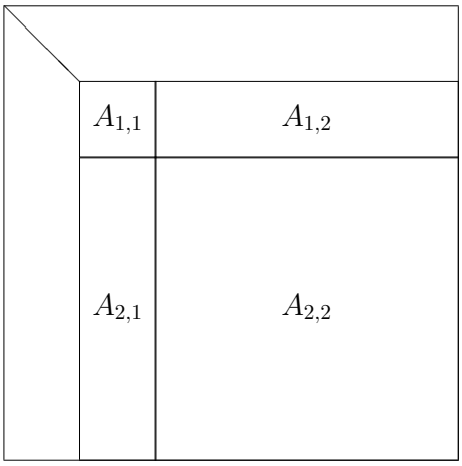

Fig. 11. Active regions of the original matrix at a given step of the computation.

1) Adding the Scheduler: There is a variety of work in CharmLU that fits the accelerator paradigm: much of the work is data-parallel. However, because of the amount of asynchrony in CharmLU and the complex priorities required to obtain high performance, the type of work to offload must be carefully selected. The local LU blocks are not a reasonable selection because they represent a relatively low amount of work and are directly on the critical path. Therefore, if virtualization is high, the block size will be comparatively small, and these cannot be agglomerated because only a single block is active at any given time.

The triangular solves (from the U and L broadcast) and trailing updates both perform $O(n^3)$ work on $O(n^2)$ data. Based on the work-to-data ratio, trailing updates and triangular solves are equally likely to perform well on the GPU. However, since the trailing updates represent much more work overall and triangular solves generate the trailing update work, trailing updates seem to be a better candidate for agglomeration. Hence, for this implementation, trailing updates were offloaded. It is possible that additional speedup would be observed by also offloading triangular solves (and possibly agglomerating them).

To add the scheduler, the previously synchronous control-flow was modified to asynchronously invoke the scheduler with the appropriate work to be performed. The trailing update is instigated when the L and U data is available. This data is passed to the method as two parameters. Pointers to the data in these messages are passed to the dynamic scheduler so it can perform the computation after agglomeration.

2) GPU Kernel: Once the scheduler runs, it invokes a GPU kernel that takes an agglomeration of trailing updates. A agglomerating matrix-matrix multiply was implemented on the GPU. It is much like a naive implementation with n^2 threads, each with $O(n)$ amount to work, with added offsets for indexing into the agglomerated vectors.

3) Performance: To compare the performance of the dynamic scheduler, two additional versions of the code were written to offload the trailing updates.

a) Single GPU Offload: The first version does not use the scheduler, but simply runs the computation on the GPU without agglomeration. With this version, the standard priorities of the work are respected, but the computation is just performed on the GPU instead of the CPU. The version is simple to implement, but does not yield the performance of the dynamic agglomeration.

b) Static Agglomeration: To compare to the dynamic scheduler, a static agglomeration scheme was implemented. In general, statically agglomerating work is not difficult once the basics are implemented, but sometimes ensuring progress and eliminating deadlock is difficult.

As mentioned previously, in the case of CharmLU, the trailing updates produced after each step are dynamically scheduled and can be postponed until they are on the critical path. They are on the critical path when the next step includes that block. When the GPU is not used, this is handled by creating messages for trailing updates, but setting the priority low so they do not execute until no higher priority work is present. By the time trailing updates on the critical path need to be performed, the higher priority work has been completed and the trailing updates have begun or are finished.

To implement a static scheduling scheme, it may seem that as soon as a trailing update becomes available it can be enqueued into the agglomeration buffer, and when a threshold is reached it can be offloaded. However, if cross-step trailing updates are agglomerated together it is difficult to calculate when the scheduler has reached an edge case and must offload immediately because it is inhibiting the critical path. This can deadlock when the remaining work is not evenly divisible by the threshold and a critical piece of work is waiting in the buffer. Note that this edge case can happen at every step.

Another way to statically schedule would be to agglomerate all the trailing updates in the next step. However, this greatly limits the amount of static agglomeration and is also difficult to implement because it must be aware of the distribution of the blocks.

To eliminate deadlock, only work for the current step is agglomerated. This is trivially correct and the edge cases are easy to handle. However, this limits the dynamism in scheduling and requires synchronization.

One of the major downsides of static agglomeration is the reduction of asynchrony in the application that this methodology may incur by adding new constraints. Moreover, dynamic agglomeration is beneficial because it obviates the need for handling complex edge cases that are often a consequence of static agglomeration.

Figure 12 shows the difference in performance between dynamic and static agglomeration. The static scheduler performs much worse because of the added synchronization to the application. As the static agglomeration size increases, more work is agglomerated, which combines trailing updates on the critical path with work not on the critical path. This decreases

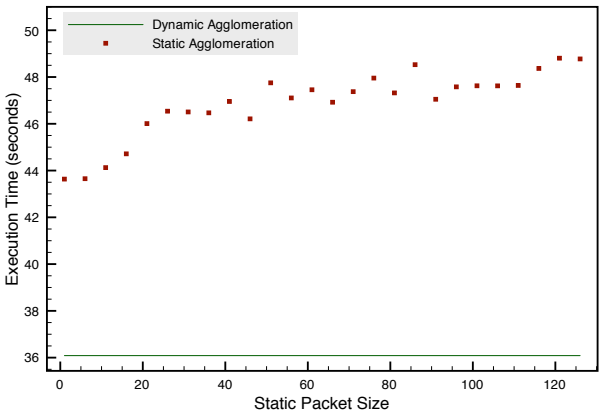

Fig. 12. CharmLU with static versus dynamic agglomeration with a 10240x10240 matrix and a block size of 256x256.

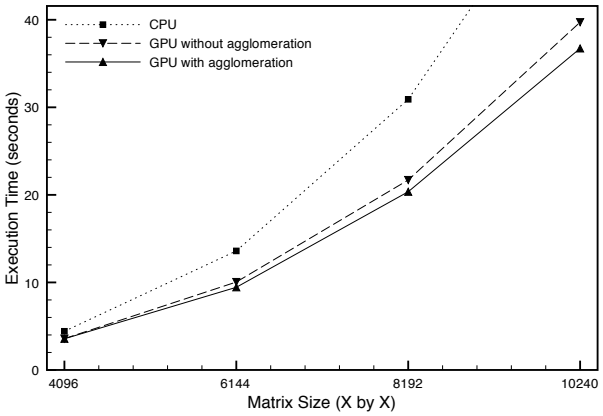

Fig. 13. CharmLU execution time versus problem size.

performance because the CPU is idle while it waits for more work to complete on the GPU.

4) Agglomeration and Problem Size: Figure 13 shows the execution time of CharmLU as the problem size (matrix size) is varied. For each run the best block size (the one that yielded the best performance) was selected. For the CPU runs, the best block size of all matrix sizes was 128×128. For the dynamic agglomeration curve, the best block size for all matrix sizes was 256×256. For the single offloading version, the best was 256×256 except for the first point (matrix size 4096) where the best was 128×128.

Because the best virtualization is constant for the dynamic agglomeration curve, the amount of overhead increases as the matrix size grows. As the problem scales, more virtualization is needed to obtain high performance, but this has the competing effect of increasing overhead. By agglomerating, the overhead is reduced compared to the single GPU offloading curve.

For the trailing updates there is $O(n^3)$ work for $O(n^2)$ data. The work-to-data ratio is $O(n)$, indicating that this operation is a reasonable candidate for agglomeration. Although, this ratio indicates that offloading may be beneficial in general, it does not necessarily predict the utility of agglomeration. For reasonable block sizes, the maximum number of threads is in-

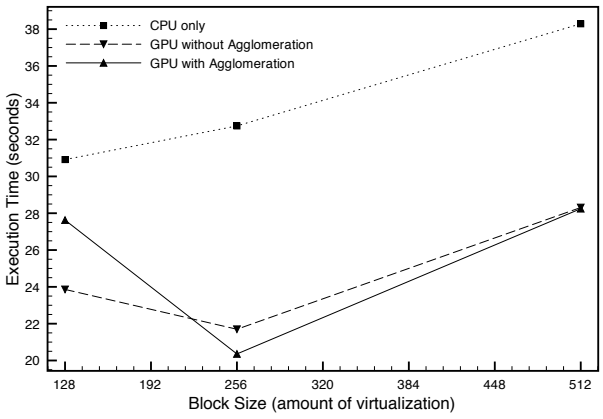

Fig. 14. CharmLU execution time versus blocks size (smaller blocks increase virtualization) for a 8192x8192 matrix.

voked ($O(n^2)$). Therefore, the major benefits of agglomeration in this problem are not due to better utilization of the GPU; they arise from the reduction of overhead.

With a 10240×10240 matrix, dynamically agglomerating work yields a 7.5% reduction in execution time, over offloading a single trailing update at a time to the GPU.

5) Agglomeration and Virtualization: Figure 14 shows the change in performance as the virtualization of CharmLU is varied with a single matrix size. It is expected that dynamic agglomeration will perform better than single GPU offloading as virtualization increases. Because the accelerator is sufficiently utilized, dynamic agglomeration is not as effective.

This reduction in performance is due to limitations in this implementation. In the version with the GPU and no agglomeration, priorities are respected for the trailing updates. In the agglomeration version, priorities are not respected since they are overloaded as part of the scheduler. In some cases this will cause low priority work to be done when there is higher priority work available. This problem could be resolved if the scheduler was part of the runtime system where it could directly examine the message queues. When the level of virtualization is ideal the agglomerating system still performs best.

C. Application Comparison

The performance of both applications improves over single GPU offloading when dynamic agglomeration is used. However, in the case of Molecular2D greater speedups are observed because the GPU is used less efficiently in the case of single work unit offloading. This is a fundamental difference in the suitability of the work being offloaded; one application requires substantially more work than what is provided to perform efficiently.

Molecular2D offloads particle interactions on a per iteration basis, because the application is synchronized every iteration in its natural form. However, the trailing updates in CharmLU happen dynamically between steps. This property makes static agglomeration much more difficult to implement in CharmLU and the added synchronization used for static agglomeration decreases performance. These same problems would be present in Molecular2D if the offloads were not synchronized at each iteration. Also, if measurement-based load balancing were used at runtime, static scheduling would become more complex.

From a software engineering standpoint, dynamic scheduling for work agglomeration is much simpler to implement and use in an application, and it requires less domain-specific information from the application to be effective. Therefore, it can be effectively implemented in a runtime system or library that generically handles dynamic agglomeration.

D. Predictable Performance

Results from both applications showed lower variance in execution time for dynamically scheduled runs because the scheduler can compensate for system variations. OS noise and varied message orderings can both cause variations in the execution time.

By greedily agglomerating work units together, arrival order of work units is less important, because any set of work units of the same type can be dynamically agglomerated, instead of an exact set of specific work units, which is required in static agglomeration. Hence, variations in message order impact the performance less with a dynamic schedule.

If OS noise interrupts the flow of work into the scheduler, the message that invokes `agglomerateWork()` has a greater chance of being delivered later, that is, with a higher level of agglomeration. Therefore, the scheduler automatically counteracts noise by offloading coarser-grained work on noisy systems, whereas the static scheduler would be forced to perform multiple offloads.

The end result is counterintuitive: an application has more consistent and predictable performance when dynamically scheduled. For the 100,000 particle run of Molecular2D, the standard deviation for dynamic agglomeration was 1.465 seconds versus static agglomeration runs, which yielded an average standard deviation of 2.429 seconds.

VI. Related Work

While agglomeration is regularly used, it has not been studied explicitly. Most work on accelerators has focused on achieving performance on specific tasks. In these cases tuning the amount of work to offload has been examined in a static manner. Our approach differs in two key respects. First, we examine the benefits of agglomeration and attempt to characterize its gains. Second, rather than attempting to find statically ideal agglomeration sizes, we provide a method for dynamically tracking the available agglomeration to optimize performance.

The area of work scheduling on accelerators has been examined with the StarPU system [17]. In addition, Wang, et al. [18] study task scheduling on the CPU and GPU. In these cases, they are interested in the heterogeneous nature of scheduling. We do allow for heterogeneous scheduling but since we only select the best candidates for offloading we do not make the same choices. We focus on the question

of agglomeration rather than location, though we believe that their methods could be combined with ours to yield even better results.

In the Charm++ Offload API [19], a framework is described in Charm++ for offloading entry methods to accelerators. However, this work does not include any notion of work agglomeration.

Schneidert, et al. [20] describe a method for streaming aggregation to GPUs, which focuses on methods for making streaming aggregation more efficient, but does not describe dynamic agglomeration techniques. In addition, Blagojevic, et al. [21] describe methods for dynamically aggregating multiple grains on Cell processors, but their techniques focus on scheduling between PPE and SPE tasks on the accelerator. They also perform a low-level dynamic "malleable loop-level parallelism," which allows them to change the grain size of parallel work by modifying how loops are executed.

Luk, et al. [22] partition work for the GPU and CPU at runtime based on training runs, history of previous runs, and the input size. However, they do not dynamically repartition work based on runtime conditions.

VII. Conclusion and Future Work

To maximize the effectiveness of our approach, a dynamic scheduler should be tightly integrated with a runtime system. A message-passing runtime system has queues to handle message delivery, which can be modified to allow tagging of accelerator work. Accelerator messages can be prioritized like CPU messages, and appropriately prioritized messages can be agglomerated together. The runtime system would call a user-defined method that handles the actual agglomeration of the work. After this, the runtime system would pass the work off to the appropriate scheduler. Such a mechanism would also allow the system to handle trade-offs between the CPU or accelerator executing the message, like the StarPU system [17].

Implementing this mechanism into a runtime system has the advantage of reducing overhead because a built-in scheduler does not have to enqueue extra messages to create the agglomeration effect.

In conclusion, we analyzed work agglomeration, a technique for achieving high performance on accelerators, and showed that it substantially improved performance in the tested cases. Moreover, we presented a novel methodology for dynamically scheduling work units for work agglomeration. Our methodology rivals static scheduling, because it out-performs it in many cases, is much easier to implement for application developers, and is adaptable across problem configurations and architectures.

References

[1] L. Kale, B. Ramkumar, V. Saletore, and A. B. Sinha, "Prioritization in parallel symbolic computing," in *Lecture Notes in Computer Science*, T. Ito and R. Halstead, Eds., vol. 748. Springer-Verlag, 1993, pp. 12–41.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, Santa Barbara, California, Jul. 1995, pp. 207–216, mIT.

[3] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[4] L. V. Kalé, "Performance and productivity in parallel programming via processor virtualization," in *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[5] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[6] "Top500 supercomputing sites," http://top500.org.

[7] J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a message-driven parallel application to GPU-accelerated clusters," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–9.

[8] P. Richmond, D. Walker, S. Coakley, and D. Romano, "High performance cellular level agent-based simulation with flame for the gpu," *Briefings in Bioinformatics*, vol. 11, no. 3, pp. 334–347, 2010. [Online]. Available: http://bib.oxfordjournals.org/content/11/3/334.abstract

[9] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn, "Scaling Hierarchical N-body Simulations on GPU Clusters," in *Proceedings of the ACM/IEEE Supercomputing Conference 2010*, 2010.

[10] P. Husbands and K. Yelick, "Multi-threading and one-sided communication in parallel LU factorization," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–10.

[11] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[12] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[13] L. V. Kale and S. Krishnan, "Charm++: Parallel Programming with Message-Driven Objects," in *Parallel Programming using C++*, G. V. Wilson and P. Lu, Eds. MIT Press, 1996, pp. 175–213.

[14] NVIDIA, *CUDA 2.0 Programming Guide*. NVIDIA Corporation, Santa Clara, CA, USA, June 2008.

[15] *The OpenCL Specification Version: 1.0*. Khronos OpenCL Working Group, February 2009. [Online]. Available: http://www.khronos.org/opencl/

[16] I. Dooley, C. Mei, J. Lifflander, and L. Kale, "A study of memory-aware scheduling in message driven parallel programs," in *Proceedings of 17th Annual International Conference on High Performance Computing*, 2010.

[17] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Euro-Par 2009 best papers issue*, 2010, accepted for publication, to appear.

[18] L. Wang, Y. zhong Huang, X. Chen, and C. yan Zhang, "Task scheduling of parallel processing in cpu-gpu collaborative environment," *Computer Science and Information Technology, International Conference on*, vol. 0, pp. 228–232, 2008.

[19] D. M. Kunzman and L. V. Kalé, "Towards a framework for abstracting accelerators in parallel applications: experience with cell," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–12.

[20] S. Schneidert, H. Andrade, B. Gedik, K.-L. Wu, and D. S. Nikolopoulos, "Evaluation of streaming aggregation on parallel hardware architectures," in *DEBS '10: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA: ACM, 2010, pp. 248–257.

[21] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos, "Dynamic multigrain parallelization on the cell broadband engine," in *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2007, pp. 90–100.

[22] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, dec. 2009, pp. 45 –55.