Implementing Directed Acyclic Graphs with the Heterogeneous System Architecture

Sooraj Puthoor
AMD Research
Sooraj.Puthoor@amd.com

Ashwin M. Aji
AMD Research
Ashwin.Aji@amd.com

Shuai Che
AMD Research
Shuai.Che@amd.com

Mayank Daga
AMD Research
Mayank.Daga@amd.com

Wei Wu
The University of Tennessee, Knoxville, USA
wwu12@vols.utk.edu

Bradford M. Beckmann
AMD Research
Brad.Beckmann@amd.com

Gregory Rodgers
AMD Research
Gregory.Rodgers@amd.com

Abstract

Achieving optimal performance on heterogeneous computing systems requires a programming model that supports the execution of asynchronous, multi-stream, and out-of-order tasks in a shared memory environment. Asynchronous dependency-driven tasking is one such programming model that allows the computation to be expressed as a directed acyclic graph (DAG) and exposes fine-grain task management to the programmer. The use of DAGs to extract parallelism also enables runtimes to perform dynamic load-balancing, thereby achieving higher throughput when compared to the traditional bulk-synchronous execution. However, efficient DAG implementations require features such as user-level task dispatch, hardware signalling and local barriers to achieve low-overhead task dispatch and dependency resolution.

In this paper, we demonstrate that the Heterogeneous System Architecture (HSA) exposes the above capabilities, and we validate their benefits by implementing three well-referenced applications using fine-grain tasks: Cholesky factorization, Lower Upper Decomposition (LUD), and Needleman-Wunsch (NW). HSA's user-level task dispatch and signalling capability allow work to be launched and dependencies to be managed directly by the hardware, avoiding inefficient bulk-synchronization. Our results show the HSA task-based implementations of Cholesky, LUD, and NW are representative of this emerging class of workloads and using hardware-managed tasks achieve a speedup of 3.8x, 1.6x, and 1.5x, respectively, compared to bulk-synchronous implementations.

CCS Concepts

• **Computer systems organization~Heterogeneous (hybrid) systems** • *Computer systems organization~Single instruction, multiple data* • **Software and its engineering~Contextual software domains** • **Software and its engineering~Runtime environments** • *Software and its engineering~Concurrent programming languages* • Software and its engineering~Massively parallel systems • Computing methodologies~Parallel programming languages

GPGPU-9, March 12-16, 2016, Barcelona, Spain
© 2016 ACM. ISBN 978-1-4503-4195-0/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2884045.2884052

Keywords

APU, Directed Acyclic Graph (DAG), Heterogeneous System Architecture (HSA), Fine-grain Task Management

1. Introduction

Many segments of the computing industry are shifting towards massively throughput-oriented processing, such as the capabilities provided by Graphics Processing Units (GPUs), because these devices better leverage recent technology trends. GPUs use data-parallel execution to amortize front-end hardware across many threads and avoid power-hungry speculations and large caches that are often used by CPUs. As a result, GPUs better leverage the increasing transistor densities of Moore's law [25] and integrated heterogeneous (CPU+GPU) systems are gaining popularity in large-scale computing, which require high single-thread and multi-thread performance.

For applications with minimal inter-thread communication, using traditional bulk-synchronous execution models to evenly distribute and coordinate work across all threads is sufficient for good performance. However, for many applications that require extensive inter-thread communication, bulk-synchronous execution leaves heterogeneous resources underutilized. For example, certain threads have to wait for others at the global barrier (e.g., GPU kernel termination) before proceeding to the following execution.

For applications with inter-thread dependencies, effective extraction of parallelism can be achieved by structuring the computation as tasks represented by a Directed Acyclic Graph (DAG) [12]. Within a DAG, nodes represent tasks and directional edges represent data dependencies. By understanding the inherent data dependencies described by an application's DAG, the underlying execution environment can immediately execute independent tasks whenever they become available, leading to higher resource utilization.

In this paper, we demonstrate that the features provided by Heterogeneous System Architecture (HSA) facilitates the implementation of an asynchronous dependency-driven tasking model. HSA is empowered to capitalize on the application's task parallelism by the virtue of features like: (1) a low-level runtime that natively supports low-latency task dispatch, and (2) hardware support for fine-grain task scheduling and management. Additionally, HSA's user-level task dispatch and queuing capability allow applications to directly launch tasks onto hardware avoiding software management, such as a centralized task coordinator used by Ltaief *et al.* [22], and other overheads due to the traditional GPU software stack. HSA also incorporates a unified virtual address space and coherent shared

Table 1. Subroutines used by clMAGMA's Cholesky Factorization.

Name	Task	Library	Device
SYRK	Symmetric rank-k update	clBLAS	GPU
GEMM	Matrix-matrix multiplication	clBLAS	GPU
TRSM	Solve triangular system of equations	clBLAS	GPU
POTRF	Cholesky factorization	LAPACK	CPU

virtual memory across all the compute devices in the system. The unified virtual address space relieves the programmer from explicit data management across multiple devices and cache coherence ensures fast shared data access from different compute devices.

We show the task dependency management and resolution mechanisms of HSA by developing parallel task-based implementations of three applications – (a) Cholesky Factorization [24], (b) Lower Upper Decomposition (LUD) [11][1], and (c) Needleman-Wunsch (NW) [26]. Our results demonstrate that the task-based HSA implementations of Cholesky, LUD, and NW achieve speedups of 3.8x, 1.6x, and 1.5x, respectively, compared to their traditional bulk-synchronous implementations.

To summarize, the contributions from this paper are:

- We are the first to demonstrate an implementation of scalable tasking using the capabilities of HSA.
- We implement three applications, Cholesky factorization, LUD, and Needleman-Wunsch, with task dependencies on HSA without the schemes used by prior software implementations.
- Across the three evaluated applications, we demonstrate the HSA-based DAG implementations achieve up to 3.8x, 1.6x, and 1.5x speedups versus bulk-synchronous implementations, respectively.

The rest of the paper is organized as follows. Section 2. discusses related work and Section 3. introduces the three evaluated applications. Then Section 4. describes the bulk-synchronous and DAG implementations of the applications and Section 5. details the HSA implementation of fine-grain tasks. Section 6. evaluates the tasking capabilities of HSA. We discuss future work and conclude the paper in Section 7.

2. Related Work

We briefly consider related work in the area of tasking models and Directed Acyclic Graphs (DAG).

2.1 Tasking Models

There are many shared memory CPU tasking models including the widely-used *pthreads* model [28]. Most pthreads implementations rely on the operating system to schedule threads, resulting in relatively long dispatch times. Lower latency task dispatch can be achieved by dedicating persistent threads [32] to the task manager and execution threads. This reduces launch latency at the expense of dedicating resources. Subsystems that use pthreads in this manner include OpenMP [14], Cilk [16], Legion [23], HPX [21] and TBB [20]. When pthreads are used in a heterogeneous environment, a CPU thread is typically needed to manage the execution of GPU tasks. In contrast, our work does not rely on centralized task management.

Several existing programming models and software packages support task management in their runtime or allow developers to describe their applications with DAGs. The recent OpenMP 4.5 specification [27] allows for tasking with data dependencies specified as clauses on OpenMP pragmas. This new specification also allows for acceleration with target devices. In addition, software-based tasking models include PLASMA [2] and StarPU [6]. To our knowledge, there is no programming model that combines tasking with hardware-managed dependencies.

A number of efforts exploring exascale computing have indicated a strong need for advanced high-throughput tasking runtimes [3]. Many of these features can now be found in HSA such as task dependencies, shared memory between CPU and GPU, lightweight signalling and GPU pre-emption.

2.2 Directed Acyclic Graphs

Bosilca *et al*. [9] discussed the use of DAGs to model asynchronous tasking. This model is widely discussed as alternative to the bulk synchronous model, especially for high-performance computing applications. Implementations or prototypes of DAGs can be found in Beltran *et al*. [15], Ltaief *et al*. [22] and Tomov *et al*. [31].

One major hindrance to the widespread adoption of task-based programming has been the dependency management, the burden of which completely lies on software today. To highlight the problem of task dependency management, we refer to the work by Ltaief *et al*. [22] where the authors implemented a Cholesky factorization algorithm using a global progress table to track and manage task dependencies. Tracking dependencies with a software-managed and centralized (global) progress table is feasible only for applications with a limited number of tasks. As the application scale out and the number of tasks increases, this centralized structure limits performance. Even other runtimes that use distributed dependency management like work-stealing (e.g., Cilk [16]) use software threads for managing tasks, and thus encounter more overhead than a hardware solution. Bauer *et al*. [7] proposed warp specialization that uses GPU synchronization instructions to manage dependencies across sub-computations. In this paper, we show how HSA-capable hardware coupled with the low-level HSA runtime can be leveraged to manage task dependencies and scheduling.

```
//Blocked left looking Cholesky factorization algo-
rithm
for blocks 1 to n:
do
        // Symmetric rank-k update
        if(block > 1) SYRK();
        // Matrix-matrix multiplication
        if(block > 1 & block < n) GEMM();
        // Cholesky factorization
        POTRF();
        // Triangular solve
        if(block < n) TRSM();
end do
```

Figure 1. clMAGMA's Cholesky factorization algorithm

3. Application Case Studies

3.1 Cholesky Factorization

Cholesky factorization is used for solving a system of linear equations (see Figure 1). The implementation of Cholesky factorization in the MAGMA library involves the factorization of a symmetric positive-definite matrix into a lower triangular matrix and its con-

```
// Tiled LU decomposition algorithm
for diagonal tiles 1 to n-1
do
        // kernel updating diagonal tile
        diagonal();
        // kernel updating perimeter row tiles
        perimeter_row();
        // kernel updating perimeter column tiles
        perimeter_col();
        // kernel updating internal tiles
        internal();
end do
// kernel updating last diagonal tile
diagonal();
```

Figure 2. LUD Algorithm

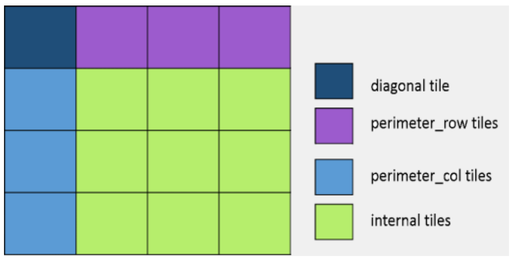

Figure 3. The major data structure of LUD. A 2-D matrix is decomposed to multiple tiles.

jugate transpose. These types of symmetric positive-definite matrices are common in many high performance applications modelling real-world physical phenomenon. In essence, Cholesky factorization represents a broader and important HPC application domain. Moreover, this algorithm has been well researched to explore and demonstrate the improved parallelism of task-based models [21]. The clMAGMA library [31] is the OpenCL™ implementation of MAGMA library. Cholesky factorization implemented in the clMAGMA library uses clBLAS [13] library calls and LAPACK [5] library calls. The clBLAS library is the OpenCL implementation of BLAS. Subroutines of clBLAS library run on the GPU whereas the LAPACK subroutines run on the CPU. Each loop iteration uses three subroutines from the clBLAS library and one subroutine from the LAPACK library, as shown in **Table 1**.

3.2 LUD

Lower upper decomposition (LUD) is a numerical analysis technique to decompose a matrix into a lower triangular and upper triangular matrix. Similar to Cholesky factorization, LUD is also a numerical analysis technique but unlike Cholesky that can only be used to factorize symmetric positive definite matrices, LUD doesn't impose any such restrictions. We use the LUD implementation from Rodinia benchmark suite [11] as our reference implementation. This LUD implementation from Rodinia uses LUD algorithm without pivoting. In addition, though the computation patterns of LUD and Cholesky are somewhat similar, because LUD needs to generate both upper and lower triangular matrices, this leads to different generated DAGs and GPU resource utilizations from Cholesky (see Section 4.   ).

Figure 2 shows the parallel patterns of the LUD implementation in Rodinia. Each loop iteration uses four subroutines. All the subroutines used in the algorithm are implemented using OpenCL and run on the GPU. Each iteration divides the matrix to be factorized into a diagonal tile, perimeter tiles (along vertical and horizontal directions) and internal tiles. Figure 3 shows the different types of

```
// Tiled Needleman-Wunsch (NW) algorithm
for diagonals 1 to n
do
        for all tiles in the diagonal
        do parallel
                // call NW kernel
                nw();
        end do parallel
end do
```

Figure 4. Needleman-Wunsch (NW) Algorithm

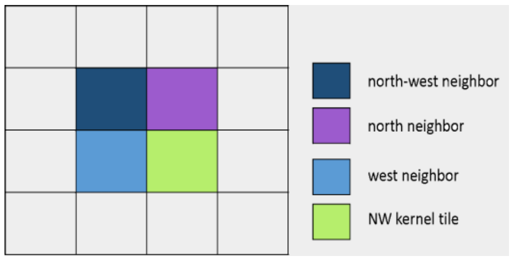

Figure 5. A 2-D matrix of Needleman-Wunsch tiles. The processing of a given tile will begin only when the processing of its north, west, and north-west neighbours are finished.

tiles of the matrix in one iteration of LUD. The diagonal kernel operates on the diagonal tile of the matrix. Similarly, the perimeter kernels operate on the perimeter tiles and the internal kernel operates on the internal tiles. The data dependencies are expressed as follows. In the ith iteration the diagonal kernel only accesses the elements on the diagonal tile. The perimeter kernels read the elements from both diagonal and perimeter tiles and update the elements on the perimeter tiles. The internal kernels read the elements from both perimeter and internal tiles and updates the elements of the internal tiles. In the (i+1)th iteration, the diagonal tile will move from the (i, i) tile position to the (i + 1, i + 1) position. After processing the (i + 1, i + 1) diagonal tile, the associated perimeter and internal tiles will be processed. This will not include the tiles with indices (m, n) where $m < i$, $n < i$. Thus, the computation domain of LUD keeps shrinking across iterations while the last iteration only processes the last diagonal tile.

3.3 Needleman-Wunsch

Needleman-Wunsch (NW) is a global optimization method for DNA sequence alignment. The potential pairs of sequences are organized in a 2-D matrix. The NW algorithm in Rodinia fills the matrix with scores, which represent the value of the maximum weighted path ending at that cell. A trace-back process is used to search for the optimal alignment. A parallel Needleman-Wunsch algorithm processes the score matrix in diagonal strips of tiles from top-left to bottom-right.

Figure 4 shows the tiled NW algorithm used in the Rodinia benchmark suite. In that implementation, there are two GPU kernels used in the entire application, one for processing the top-left part of the matrix and the other for the down-right part of the matrix. The two similar kernels are implemented using OpenCL and operates on the tiles of two matrix regions. For both kernels, each iteration of the algorithm updates the elements of a strip of tiles (along the north-east to south-west diagonal) after reading the north, west and north-west neighbors of that tile. The NW kernel tile which depends on its north, west and north-west neighbors are shown in Figure 5. Each iteration of the algorithm moves to the next

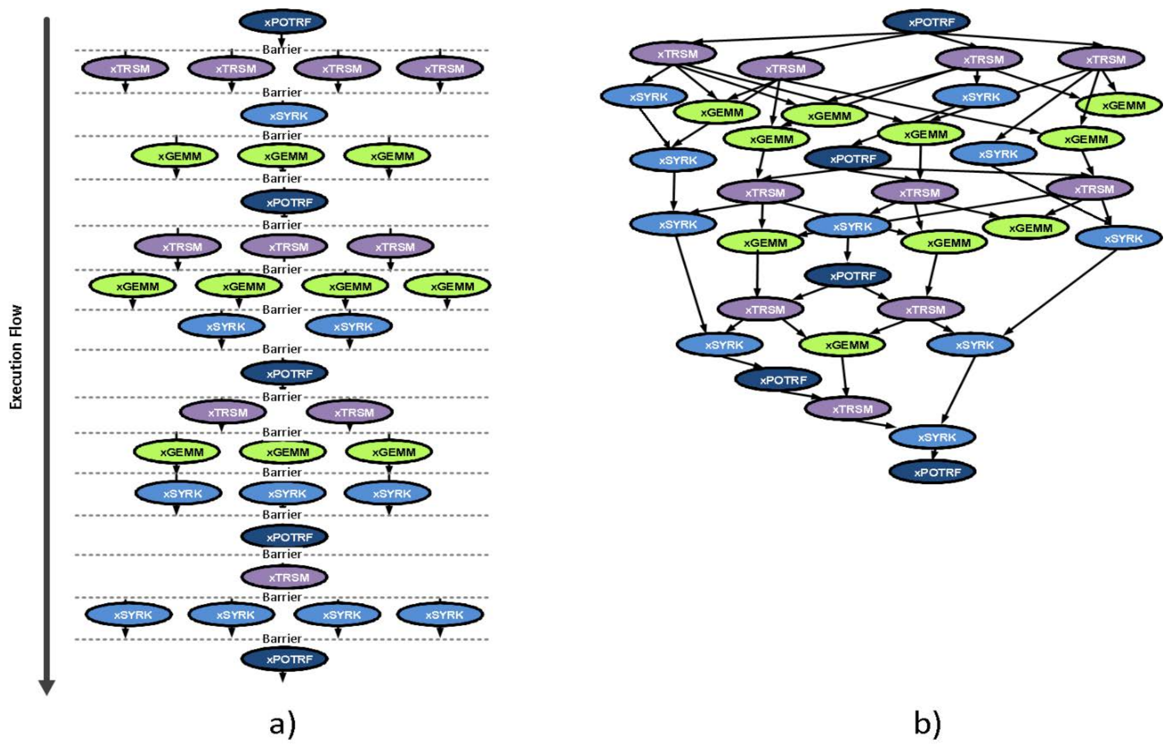

Figure 6. Cholesky Factorization Execution: a) Bulk-synchronous. b) DAG

strip of diagonal tiles after updating the current strip of diagonal tiles.

4. BSP vs. Asynchronous Task-based Execution

Depending on the problem, developers often need to choose how to synchronize multi-threaded execution. One option is bulk synchronization often using barriers. This model allows programmers to easily reason about the program's synchronization and communication. Another option is asynchronous task-based execution where data dependencies are explicitly identified. This requires the exploration of more fine-grain control to manage data dependencies and may yield better performance.

Valiant introduced the Bulk-Synchronous Parallel (BSP) programming model for mapping high-level programs to hardware without losing the efficiency of the program [33]. In the BSP model, the application splits into a sequence of computation and communication stages (stages are called super-steps), with each computation stage parallelized into multiple threads. The communication stage acts as a synchronization point between successive computation stages. Most current GPU execution models follow the bulk-synchronous model where the computation is defined by kernels and implicit synchronization happens across kernel boundaries.

The asynchronous task-based model represents an algorithm as a graph of inter-dependent tasks. Tasks are represented by the nodes of DAG and the dependencies are represented by the graph edges. A task is ready to be executed when all its dependent tasks are completed. Independent tasks of the DAG can be executed in parallel and synchronization across tasks is only needed for dependent tasks. The BSP execution is actually a subset of task parallel execution that does not fully exploit the fine grained synchronization and parallelization potential of task parallel execution.

In the succeeding sub-sections, we demonstrate the difference between the bulk-synchronous and asynchronous task-based implementations using the three benchmarks Cholesky factorization, LUD and NW. Our bulk-synchronous implementations always run only one type of kernel at a time. Additionally, the kernels are launched as large monolithic kernel as opposed to tiled versions used by the tasked-based implementation. This reduces the number of kernels launched thus bringing down the kernel launch overhead. For example, in Figure 6(a), even though the execution flow shows four xTRSM instances launched (second row of the figure), the xTRSM bulk-synchronous kernel is actually one large monolithic kernel that operates on four tiles. The same kernel launch semantics is applicable to all other figures showing bulk-synchronous execution.

4.1 Cholesky Factorization

4.1.1 Bulk-synchronous Design

Figure 6(a) shows the bulk-synchronous implementation of Cholesky Factorization and highlights the inefficiency of the execution. The algorithm uses bulk-synchronization across all threads between the subroutine calls and thus only one subroutine can be executed at a time. The SYRK subroutine is dependent on the results from TRSM and SYRK subroutines of previous iterations. The GEMM subroutine is not dependent on SYRK subroutine but dependent on GEMM and TRSM subroutines of previous iterations. Since GEMM and SYRK subroutines within a loop iteration are independent, both of these subroutines can be executed in parallel but the bulk-synchronous execution only allows the parallel execution of one kernel type at a time. Similarly, the POTRF subroutine only needs the diagonal blocks from the SYRK subroutine and can be executed in parallel with GEMM subroutine. This parallelization opportunity is also not exploited by the bulk-synchronous model.

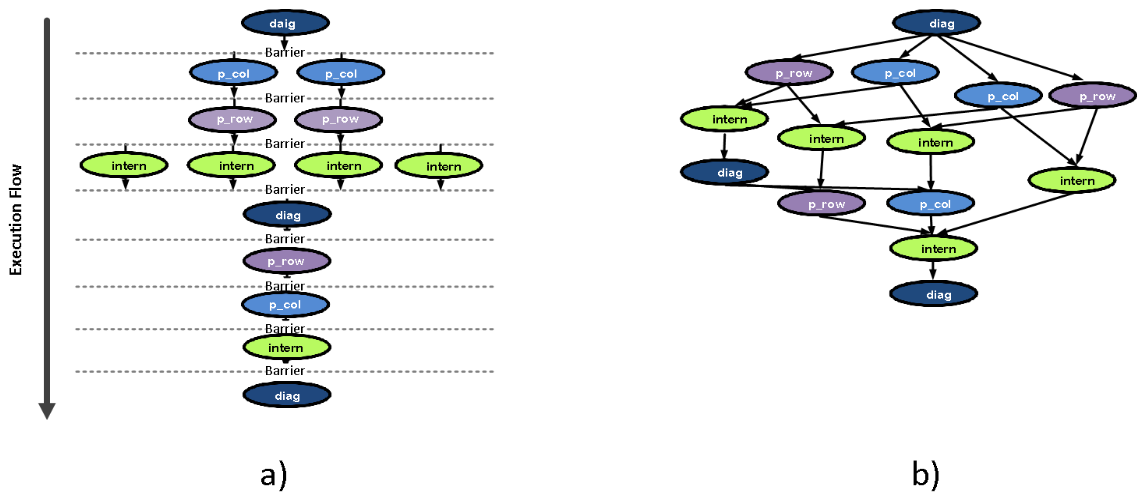

Figure 7. LUD Execution: a) Bulk-synchronous. b) DAG

The bulk-synchronous model unnecessarily stalls parallel resources waiting for the last thread to complete, leading to underutilization.

To summarize, while the bulk-synchronous model is easy to implement, it does not fully utilize the available parallelism inherent in the application and requires long running tasks to achieve good performance. As an alternative, the dataflow programming models that support fine-grain tasks are emerging for heterogeneous systems. The next section describes one such dataflow model in detail.

4.1.2 Asynchronous Task-based Design

We use the parallel task-based Cholesky algorithm proposed by Ltaief *et al*. [22] for our asynchronous tasked-based implementation. As compared to the bulk-synchronous Cholesky factorization algorithm, the task-based Cholesky algorithm presents several additional opportunities for parallel execution. The asynchronous task-based design identifies several individual tasks that are independent and can be executed in parallel. For example, all instances of TRSM, GEMM, POTRF and SYRK kernels that are not interconnected can execute in parallel and in any order. As a result, the application can take better advantage of the available heterogeneous resources.

The task-based Cholesky algorithm differs from the bulk-synchronous Cholesky algorithm, as described in Section 4.1.1 in several important ways. Both use the exact same kernel functions; however the asynchronous task-based algorithm splits the data matrix into a 5x5 grid of tiles. Tiling exposes the possibility of running computations on different data tiles in parallel. Finer tiling decreases the granularity of tasks, which leads to an increase in parallelizing opportunity depending on the overhead for launching tasks. We extract the OpenCL kernels from clBLAS library and use them directly as our task implementations. Hence, the TRSM, SYRK and GEMM tasks run the same code implemented by clBLAS library on the GPU. On the other hand, the POTRF task uses the LAPACK POTRF subroutine and executes it on the CPU.

4.2 LUD

4.2.1 Bulk-synchronous Design

Figure 7(a) shows the bulk-synchronous implementation of LUD where each kernel execution represents a computation stage. There is an implicit global synchronization after each kernel execution that involves sending a completion notification to the host CPU before the host launches the next kernel. Although there is no dependency across p_col and p_row, they are serialised because only one type of kernel is executed in our bulk-synchronous implementation. Thus the bulk-synchronous implementation utilizes the thread-level parallelism within each kernel, but serializes execution between kernels.

4.2.2 Asynchronous Task-based Design

In addition to the intra-kernel parallelism utilized by the bulk-synchronous implementation, the asynchronous task-based implementation exposes inter-kernel parallelism between tasks by explicitly identifying data dependencies. The bulk-synchronous implementation can be converted into an asynchronous task-based implementation by identifying the producer-consumer relationship between tiles. In particular, one iteration of the perimeter tile calculations depend on data from the diagonal and perimeter tiles. The elements in the diagonal tiles are last updated by the diagonal task of the current iteration whereas the elements of the perimeter tiles are updated by the internal kernels from the previous iteration. In the task graph, this is represented as dependencies between the perimeter task and the diagonal and internal tasks. Similarly, an internal kernel tasks depends on data from both perimeter and internal tasks. In particular, an internal task is dependent on the perimeter tasks from the current iteration and the internal task from the previous iteration. In addition, the internal task depends on the perimeter tasks that updated the tiles on its same row and column. Meanwhile a diagonal task only accesses elements from a diagonal tile, thus a diagonal task only depends on the internal task from the previous iteration.

Figure 7(b) illustrates the task graph for a 3x3 tiled LUD execution. The task graph uses the same kernel functions from the bulk-synchronous implementation, but the asynchronous task-based kernels act as tasks that are operating on a single tile of the matrix. It can be seen from the task graph that the perimeter_row and perimeter_col tasks are only dependent on the diagonal tasks and can be executed in parallel. Additionally, the task graph shows several opportunities for diagonal, internal and perimeter tasks to execute in parallel.

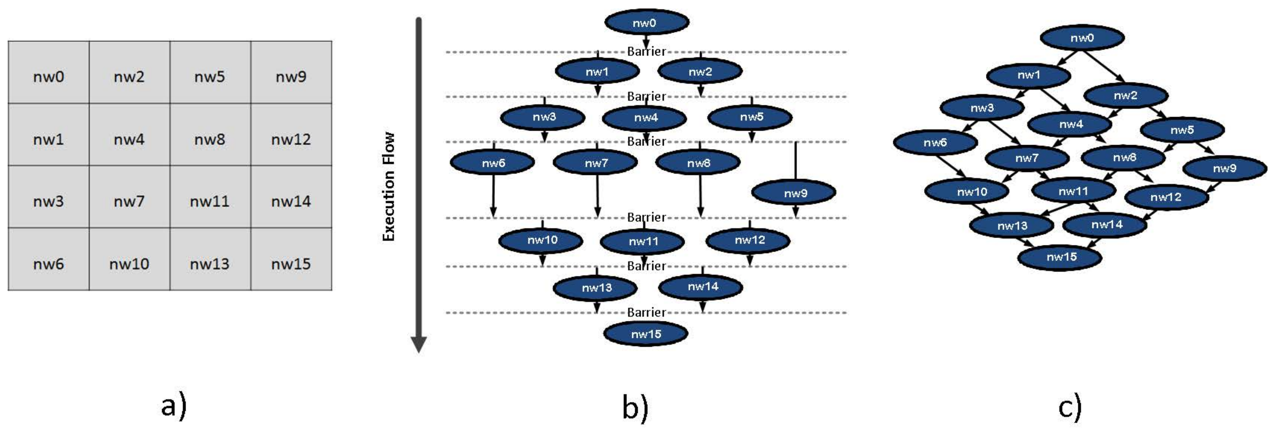

Figure 8. NW Execution: a) Tiled matrix annotated with task numbers b) Bulk-synchronous execution flow c) DAG execution flow

4.3 Needleman-Wunsch

4.3.1 Bulk-synchronous Design

The bulk-synchronous implementation of Needleman-Wunsch splits the execution into separate computational and communication phases. The computation phase of NW executes matrix tiles in a diagonal-strip manner. Each iteration will process one strip of tiles with a GPU kernel call. Different strips are processed in serial. As a result, the next strip of diagonal tiles will begin processing when the previous strip is complete.

Figure 8(c) shows the task graph of 4x4 tiled NW algorithm. Since the same NW kernel operates on all tiles in this application, there is only one type of task in this task graph. Figure 8(a) shows the tiled 4x4 matrix along with the task numbers. The task number specify the tile on which that task is operating on. For example, task nw0 will operate on tile (0, 0) of the NW matrix, task nw1 will operate on tile (1, 0) and so on.

4.3.2 Asynchronous Task-based Design

From the task graph of NW, Figure 8(c), it can be seen that the NW task graph does not expose a lot more parallelism than its bulk-synchronous implementation (Figure 8 (b)).

However, the parallelization opportunity in NW comes from the ability to exploit more fine-grain parallelism when there is not enough tiles in a strip to fill the GPU. This allows us to process tasks across the boundary of original strips. For example, in the BSP implementation, one NW kernel instance operates on tiles (2, 0), (1, 1) and (0, 2). Consequently, even if there is available parallel resource, nw6 task cannot start execution until the completion of the kernel operating on tiles (2, 0), (1, 1) and (0, 2). However, in our task implementation, nw6 task can start execution after the completion of tasks operating on tile (2, 0).

5. Tasking on the HSA Platform

5.1 Heterogeneous System Architecture (HSA)

HSA is a standardized hardware and software platform supported by several industrial and academic partners [17]. Microprocessors, operating systems, and runtimes have recently become available to support HSA including the AMD FX-8800P APU [10]. The HSA standard specifies hardware requirements, such as shared coherent virtual memory, and provides a low-level runtime API with extensive tasking capabilities. Our evaluations in this paper are based upon HSA v1.0, which was released in March 2015 [19].

One of the primary benefits of the HSA programming environment is shared coherent virtual memory between the CPU and the GPU. This benefit improves usability and performance as compared to previous heterogeneous programming environments by eliminating data copies and allowing memory pointers to be directly shared between heterogeneous components. The shared virtual memory also eliminates the memory size limitation which manifests in discrete GPU systems. Hardware maintains coherent shared memory across GPU and CPU cache hierarchies, eliminating involvement from the programmer.

HSA provides user-mode task queueing, where task data structures can be read or written to by the application, thereby enabling low-latency task dispatching. Furthermore, HSA provides scalable features including multiple task queues, ordered or out-of-order execution of tasks within the same queue, and task signalling that can block kernel execution based on a set of dependencies. These dependencies can be defined as a list of predecessor tasks that must all be completed (AND), a list of predecessor tasks where any one must be completed (OR), or any combination of dependencies. When its dependencies are satisfied, a blocked task will automatically become eligible for execution without any additional software involvement. Both the structures to notify task completion and to track task dependencies are kept in shared memory and a dedicated hardware continuously monitors these shared memory locations to automatically detect resolve dependencies.

Our DAG implementations use the HSA dependency management feature to identify the set of dependent tasks when a task is placed on a task queue. To implement our particular DAGs we only need to specify AND dependencies. When a task is created (dispatched, but not necessarily executing), only the dependent (upstream) tasks are required in order to build the dependency list. No consideration is required for downstream tasks. As a result, DAGs can be constructed dynamically but must be constructed in a top-down fashion.

To summarize, we use three HSA features to implement low latency fine-grained tasking:

1. Heterogeneous Uniform Memory Access (hUMA) ensures GPU and CPU share system memory, alleviating the programmer from explicitly managing data between disjoint memory spaces.

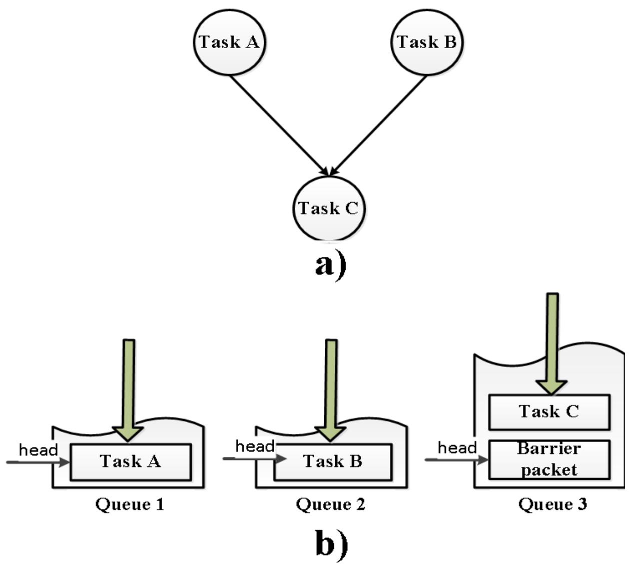

Figure 9. HSA Tasking: a) Example DAG. b) Task Queue Implementation

2. HSA user-mode queuing allows the programmer to specify the DAG using HSA queues without using system software.
3. Dedicated hardware monitors task dependencies, identifies the ready tasks, drains the HSA queue with ready tasks, and launches theses ready tasks without involvement from the programmer.

5.2 Asynchronous Task Management Interface (ATMI)

The HSA runtime API is a low-level interface that is intended to be used by compiler and system runtimes to leverage the functionalities of HSA-capable architectures. We developed ATMI on top of HSA, as a descriptive programming model to easily create and dispatch tasks, and manage task dependencies. The ATMI programming model consists of standard data structures to specify task identities and its launch parameters (Figure 11). The launch parameter data structure also stores other task attributes like task dependencies (requires and needs any), stream object and a Boolean synchronous flag. An ATMI task signature extends the GPU kernel declaration or a standard C function declaration by adding a launch parameter structure as its first argument. An example ATMI program is shown in Figure 10, which corresponds to creating and dispatching dependent tasks, as depicted in Figure 9. An ATMI task may declare GPU kernels or standard C subroutines as its task implementation via custom function attributes. We also developed a GCC compiler extension to handle the custom function attribute and generate the ATMI task definition, which contains corresponding low-level HSA runtime code that invokes the corresponding task implementation. Thus, ATMI abstracts the HSA tasking capabilities into a data structure that defines the launch parameters of user-defined kernels and their dependencies. Furthermore, ATMI eliminates the verbose prescriptive API of other task-based programming models like OpenCL, thereby making the code much more readable. As a result, ATMI achieves the following objectives:

1. ATMI defines a simple descriptive task programming model. Programmers can write kernels using traditional shared memory arguments, and invoke them like any standard subroutine.

```
atmi_lparm_t lp_a, lp_b, lp_c;
atmi_task_t *parents[2];
parents[0] = task_a(&lp_a, ...);
parents[1] = task_b(&lp_b, ...);
lp_c.num_required = 2;
lp_c.requires = parents;
lp_c.synchronous = true;
atmi_task_t *c = task_c(&lp_c, ...);
```

Figure 10. ATMI example to create and dispatch dependent tasks

```
typedef struct atmi_lparm_s {
    unsigned long    gridDim[3];
    unsigned long    groupDim[3];
    atmi_stream_t*   stream;
    int              kernel_id;
    bool             synchronous;
    int              num_required;
    atmi_task_t**    requires;
    int              num_needs_any;
    atmi_task_t**    needs_any;
    atmi_lprops_t    properties;
} atmi_lparm_t;
typedef struct atmi_task_s {
   atmi_state_t     state;
   atmi_tprofile_t  profile;
} atmi_task_t;
```

Figure 11. ATMI Launch Parameters and Task Handle Structures

2. The backend HSA runtime code is automatically generated with our GCC compiler plugin.
3. Constructing DAGs is highly simplified with ATMI because the programmer only needs to set relevant data structure fields in the launch parameters to denote the parent-child relationship.
4. ATMI can be used as an intermediate target for other compilers that implement tasking and GPU acceleration without binding them explicitly to the HSA runtime, thereby allowing ATMI programmers to target different platforms in the future.
5. The intermediate-to-advanced programmer can also provide any non-default launch parameter values to control the execution of the task.

We used ATMI to implement all presented application designs. In the next few subsections, we explain the design of the HSA runtime code that gets generated from ATMI tasks.

5.3 Task Creation and Dispatch

Task dispatch in our implementation heavily uses the HSA runtime features [19]. An HSA task is encoded as a *kernel dispatch packet*, and a task dispatch is equivalent to a simple enqueue of the corresponding kernel dispatch packet to an HSA queue. The HSA queue is a user-mode queue that can be read or written to by the application, and is associated with the GPU of the system. Thus the entire packet creation and queuing operation happens in the user domain and does not require any system services, avoiding any protection domain crossing overhead.

Every kernel dispatch packet has a *completion signal* associated with it. It is the responsibility of the GPU hardware to signal the completion of a task using the completion signal. Only the data structures used to represent completion signals are managed by the target runtime (e.g., ATMI) or the HSA programmer. ATMI uses reference counting to manage the signal data structures. However, unlike other software-based task completion detection techniques, completion signals are triggered by the driver as hardware interrupts and software is not on the critical path for launching data-dependent tasks. We will describe the role of this completion signal in managing task dependencies next.

5.4 Task Dependency Management

The HSA API [19] defines barrier packets to specify and manage dependencies. Each barrier packet can have up to 5 dependency signals associated with it. The list of signals can be AND'ed or OR'ed. Our implementation uses the AND version of the barrier packet, which waits for all completion signals in its dependency list before any tasks enqueued after the barrier packet can make forward progress. To support more than 5 task dependencies, barrier packets can be arranged in hierarchy and ordered sequentially in a single HSA queue. The only overhead with this hierarchical approach is the processing time of the additional barrier packets by the dedicated hardware.

We rely on the properties of kernel completion signals and HSA barrier packets to manage task dependencies without any user intervention. We will explain our scheme with the help of a simplified top-down DAG shown in Figure 9(a). This DAG has three tasks A, B and C. Task C is dependent on both A and B. Tasks A and B do not have any dependencies and can start immediately. We create two kernel dispatch packets corresponding to tasks A and B and then enqueue them on two different HSA queues as shown in Figure 9(b). Since task C depends on tasks A and B, we first create a barrier packet connected to the completion signals of tasks A and B. This barrier packet is then enqueued to another HSA task queue after which the kernel dispatch packet for task C is enqueued. The result is the barrier packet forces the issuing of task C to respect its dependencies on A and B. While this example is a very simple DAG, the scheme can be extended to implement any complex DAG as long as the parent tasks are dispatched before the dependent child task. We leave other types of DAG construction to future work.

5.5 Task Scheduling and Execution

The task dependencies are managed with the help of completion signals and barrier packets. Once the task dependencies are specified and packets enqueued to HSA queues, the underlying hardware is responsible for triggering the completion signals, resolving the dependencies, and scheduling the tasks. On the GPU, HSA-compatible hardware is responsible for analysing the packets enqueued on the HSA queues. Hardware is also responsible for signalling the completion of a task using its completion signal and for draining packets from HSA queues in FIFO fashion. When the head of HSA queue has a barrier packet, the hardware evaluates the packet's dependency signals and checks for the completion of the packet. Younger packets on the queue are not processed till the completion of the older barrier packet. Since a barrier packet can only stall the launch of a single queue, the hardware is free to launch independent tasks from other queues. This dynamic load balancing leads to maximum utilization of available parallel resources of the GPU.

Conceptually, the same approach can be used for launching tasks on CPU. However, in our implementation, CPU kernels run on separate persistent pthreads and use the hsa_signal_wait API call provided by HSA runtime [19] to wait for the completion of dependent tasks.

To summarize, we use HSA API calls, packets, and signals as well as the underlying hardware support to execute asynchronous tasks on all the CPU and GPU cores of the APU. After the host code specifies the dependencies and launches the tasks, the HSA runtime and hardware dynamically track and manage dependencies, and schedule tasks to provide load balancing and correct dataflow execution without any user intervention.

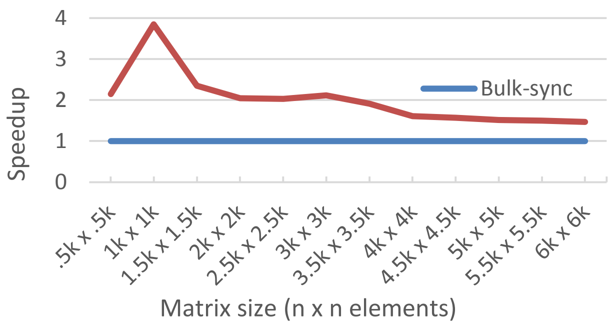

Figure 12. Cholesky Speedup

6. Results

This section demonstrates the HSA capabilities of fine-grain tasking. The execution times for the asynchronous task-based implementations of all three applications are compared against their bulk synchronous counterparts. Both implementations avoid copying data between the CPU and GPU by using HSA's shared coherent address space. Thus comparing execution times demonstrates the low overhead tasking capability of HSA versus the conventional bulk-synchronous execution.

We use an AMD FX-8800P APU [10] to evaluate tasking capabilities of HSA. This APU has 8 GPU cores and 4 CPU cores. The CPU runs at 3.7GHz and the GPU runs at 720MHz. This APU supports HSA features [17] such as pageable shared virtual memory, user mode queuing and signalling. Each ATMI application uses 24 HSA queues, with each queue having the capacity to hold up to 128,000 tasks.

6.1 Cholesky Factorization

Figure 12 shows the speedup of the Cholesky asynchronous task-based implementation versus the Cholesky bulk-synchronous implementation. The same clBLAS kernels are used for both the bulk-synchronous and asynchronous task-based implementations. These kernels are optimized for 128x128 tile size, so we fixed the tile size to 128x128 for all input matrix sizes. For smaller input matrices, the asynchronous task-based implementation experiences significant speedup−3.8x for the 1k x 1k matrix. These impressive speedups are due to a combination of the heterogeneous parallelism and the task parallelism available in the Cholesky asynchronous task-based implementation. Previously, Section 4.1 identified that the POTRF tasks in Cholesky executed on the CPU. In the asynchronous task-based implementation, these CPU tasks can run in parallel with independent GPU tasks (SYRK, GEMM, and TRSM). Meanwhile in the bulk-synchronous implementation, the POTRF tasks serialize execution to a single CPU thread. In addition, the asynchronous task-based implementation exposes more task parallelism across GPU task types, which increases GPU resource utilization for the smaller inputs when the parallelism within a single task type is minimal. At larger input sizes, the asynchronous task-based implementation of Cholesky achieves at least a speedup of 1.5x. At these larger inputs, even the bulk-synchronous tasks occupy all the GPU resources when the GPU task types are executing. However, the asynchronous task-based implementation also allows the CPU and GPU tasks to execute simultaneously, whereas the bulk-synchronous implementation always serializes on the CPU POTRF tasks.

Table 2. LUD DAG vs. Bulk-Synchronous Speedup across multiple Task and Input Sizes

Input size	Task size					
	128	320	640	1280	2560	6400
.5k x .5k	1.37	1.39	1.33	1.36	1.38	1.31
1k x 1k	1.42	1.47	1.51	1.49	1.52	1.50
1.5k x 1.5k	1.48	1.58	1.62	1.57	1.55	1.53
2k x 2k	1.40	1.57	1.62	1.50	1.50	1.48
2.5k x 2.5k	1.37	1.63	1.63	1.56	1.49	1.49

6.2 LUD

Next we evaluate the performance of the bulk-synchronous and asynchronous task-based implementations of LUD. To avoid performance variation that could be introduced by using different types of kernels, the exact same GPU kernels from Rodinia are used for both the bulk-synchronous and asynchronous task-based implementations. Only three out of the four kernels used in the Rodinia implementation are easy to adapt to different tile sizes. In addition, some kernels do not support arbitrary sizes due to hardware resource limitations. For example, the kernels extensively use the local data store (LDS) for better data reuse and low latency and workgroup-level synchronization for its correct operation. In the case of a diagonal task, any tile size larger than 32x32 will require more LDS than available on a single GPU compute unit. So we had to fix the diagonal kernel to operate on a 32x32 tile and force the diagonal task in our implementation to operate on a single tile of 32x32 elements.

Figure 13 compares the performance of task parallel LUD implementation against the bulk-synchronous reference implementation from Rodinia. The asynchronous task-based LUD achieves better performance by allowing independent tasks to execute concurrently leading to effective utilization of available parallel hardware resources. However, this scheme works well only if there is availability of unused execution resources. For small input matrices, the kernels used by the bulk synchronous implementation are small and use only a subset of execution resources available in the GPU. The asynchronous task-based implementation increases the utilization of available hardware resources for small matrices by allowing concurrent execution of independent tasks. Hence, the greater speedup with the asynchronous task-based implementation for small matrices. However, for large input matrices, the kernels launched by the bulk synchronous implementation are large leaving no unused hardware resource. Thus the parallel task implementation cannot significantly increase performance versus the bulk synchronous implementation.

Even for large input matrices, fine grained tasking can still improve performance by launching tasks faster than the bulk synchronous design. The bulk synchronous implementation has to notify the host CPU of kernel completion after which the host will launch the next kernel to the GPU. This represents the communication stage of the BSP model where the execution resources are kept idle. However, for task-based implementation, the communication stage is completely removed and the next task is launched immediately when execution resources are available. The modest performance gain of task-based implementation over bulk synchronous for large matrix sizes comes from this removal of communication stage overhead in task-based implementation.

We also performed a task size sensitivity study to determine the optimal tile sizes for different input matrices. **Table 2** shows the tile sizes of different tasks for various input matrices. The diagonal tile is fixed to 32x32 elements. Different perimeter_row tasks operate on 32x32 and 128x32 tiles and different row perimeter_col tasks operate on 32x32 and 32x128 tiles. Internal tasks operate on 32x32, 32x128, 128x32 and 128x128 tiles. We report the task size sensitivity study results by changing the maximum tile size in one dimension. The diagonal, perimeter and internal tile sizes as discussed above represents the task size 128 in the table.

It can be seen that the asynchronous task-based implementation is significantly faster than the bulk synchronous implementation for small input matrix sizes across all task size. For the 1k x 1k input matrix, the asynchronous task-based implementation is almost 50% faster than its bulk-synchronous counterpart for all task sizes. Overall, we see that the 320 task size performs best, so that is the size we selected for the results showed in Figure 13.

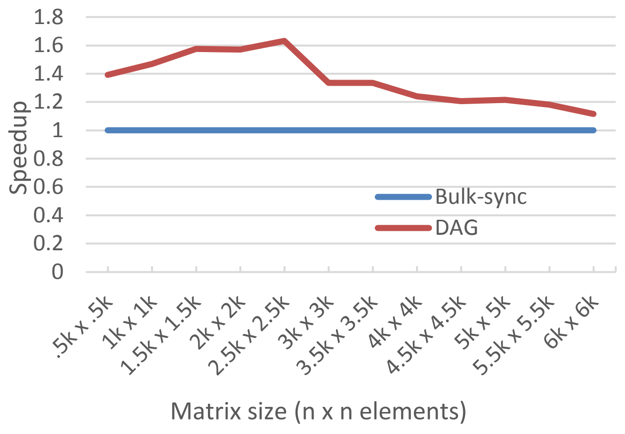

Figure 13. LUD Speedup

6.3 Needleman-Wunsch

As discussed in earlier sections, the NW task graph does not expose more parallelism than its BSP implementation. Instead the opportunity to improve performance in this application comes from the ability to immediately fill available execution resources with fine-grain parallel tasks. Figure 14 shows performance of task parallel NW against bulk synchronous implementation. The basic tile is the 32x32 element which is a limitation imposed by the NW kernel from the Rodinia implementation. The task size is increased by increasing the number of tiles in a task as opposed to increasing the tile size of a task.

The bulk-synchronous model allows the execution of only a single instance of a kernel at a time. Consequently, this model unnecessarily stalls parallel resources while waiting for the last thread in that kernel to complete, leading to idling of these resources. This is the only opportunity that the asynchronous task-based implementation of NW has to increase performance over the bulk-synchronous Rodinia implementation. The idling of resources only becomes significant at larger kernel sizes when the entire kernel cannot fit in the available parallel hardware resources. Consequently, the asynchronous task-based implementation shows a performance advantage for these large input matrices (Figure 14).

7. Conclusion

Asynchronous task-based programming models are capable of efficiently utilizing the resources in a heterogeneous computing system. However, supporting task-based applications require certain hardware features to achieve low-overhead task dispatch and dependency resolution. Heterogeneous System Architecture (HSA)

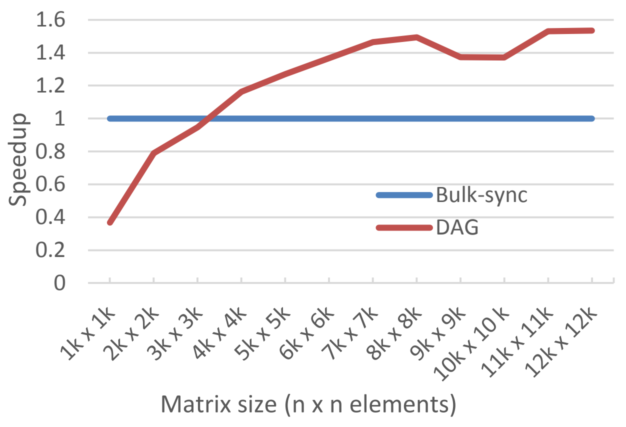

Figure 14. Needleman-Wunsch Speedup

significantly advances the performance and usability of task-based programming models. We evaluate the capabilities of HSA for fine-grain task management by implementing three well-known applications using asynchronous tasks: Cholesky factorization, Lower Upper Decomposition (LUD) and Needleman-Wunsch (NW). We demonstrate that HSA's native support for dynamic scheduling allows for the execution of high-volume, fine-grain tasks.

Going forward, we would like to focus on tools like ATMI to adapt to more scenarios with fine-grain tasking beyond the static task graphs evaluated in this paper. We would also like to port existing tools and compilers that expose asynchronous tasks to HSA and leverage its low-overhead automatic scheduling of tasks. In addition, we would like to compare our HSA implementations to other software implementations and fully evaluate the benefits of hardware managed tasking. Finally, we plan to explore various task queue scheduling policies and low-latency launch mechanisms.

Acknowledgements

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies. OpenCL is a trademark of Apple, Inc. used by permission by Khronos.

References

[1] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief and S. Tomov, "LU factorization for accelerator-based systems." In *Proceedings of the 2011 9th IEEE/ACS International Conference on Computer Systems and Applications*, 2011

[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," in Journal of Physics: Conference Series, vol. 180, no. 1. IOP Publishing, 2009, p. 012037.

[3] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, K. Yelick, P. Balanji, P. C. Diniz, A. Koniges, and M. Snir, "Exascale Programming Challenges." *Proceedings of the DOE Workshop on Exascale Programming Challenges, Marina del Rey, CA, USA*. Jul 2011. http://science.energy.gov/ /media/ascr/pdf/programdocuments/docs/ProgrammingChallengesWorkshopReport.pdf

[4] AMD Accelerated Processing Units (APUs). http://www.amd.com/en-us/innovations/software-technologies/apu

[5] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, "LAPACK Users' Guide." SIAM, 1992.

[6] 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, Special Issue: Euro-Par 2009, 23:187, 2011.

[7] M. Bauer, S. Treichler, and A. Aiken, "Singe: leveraging warp specialization for high performance on GPUs." In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming* (PPoPP '14). ACM, New York, NY, USA, 119-130. DOI=http://dx.doi.org/10.1145/2555243.2555258

[8] 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 *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[9] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing." Technical Report ICL-UT-10-01, Innovative Computing Laboratory, University of Tennessee, April 2011.

[10] D. Bouvier and B. Sander, "Applying AMD's "Kaveri" APU for Heterogeneous Computing." Hotchips 26, August 2014. http://hotchips.org/wp-content/uploads/hc_archives/hc26/HC26-11-day1-epub/HC26.11-2-Mobile-Processors-epub/HC26.11.220-Bouvier-Kaveri-AMD-Final.pdf.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, S. Lee, J. W. Sheaffer, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing." In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC),* Oct 2009.

[12] N. Christofides, "Graph Theory: An algorithmic Approach." 1975.

[13] clBLAS library, https://github.com/clMathLibraries/clBLAS.

[14] A. Duran, J. Perez, E. Ayguadé, R. Badia, J. Labarta, "Extending the OpenMP tasking model to allow dependent tasks." In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism (IWOMP'08)*, Rudolf Eigenmann and Bronis R. De Supinski (Eds.). Springer-Verlag, Berlin, Heidelberg, 111-122.

[15] A. Fernández, V. Beltran, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, " Task-Based Programming with OmpSs and Its Application", in *Workshop on Software for Exascale Computing (SPPEXA), Porto, Portugal*, 2014, 602–613.

[16] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk - 5 Multithreaded Language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation , New York, N.Y., USA,* 1998, pp. 212 – 223.

[17] HSA Foundation. 2015. HSA Platform System Architecture Specification. Version 1.0 (Jan. 2015). http://www.hsafoundation.com/standards

[18] HSA Foundation. 2015. HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer, and Object Format (BRIG). Version 1.0 (Feb. 2015). http://www.hsafoundation.com/standards

[19] HSA Foundation. 2015. HSA Runtime Programmers Reference Manual. Version 1.0 (Feb. 2015). http://www.hsafoundation.com/standards

[20] "Intel Threading Building Blocks." Available: http://www.threadingbuildingblocks.org/

[21] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS '14), 6:1--6:11*, 2014

[22] H. Ltaief, S. Tomov, R. Nath, P. Du, J .Dongarra, "A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators." In High Performance Comptuting for Computational Science – VECPAR 2010 .DOI= http://dx.doi.org/10.1007%2f978-3-642-19328-6_11

[23] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, "Legion: Expressing Locality and Independence with Logical Regions." In *the International Conference on Supercomputing*, 2012

[24] M. A. Saunders, "Large-Scale Linear Programming Using the Cholesky Factorization." Technical Report. Stanford University, Stanford, CA, USA.

[25] G. E. Moore, "Cramming More Components onto Integrated Circuits," Proc. IEEE, vol. 86, no. 1, pp. 82–85, Jan. 1998.

[26] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins". Journal of Molecular Biology 48 (3): 443–53, 1970

[27] OpenMP4.5 Specification. 2015. The OpenMP Architecture Review Board. http://www.openmp.org/mp-documents/openmp-4.5.pdf

[28] POSIX.1003.1-2008. IEEE Standard, Portable Operating System Interface (POSIX). The Open Group Standard Base Specification. http://standards.ieee.org/findstds/standard/1003.1-2008.html

[29] G. Rodgers and S. Ramalingam, "CLOC: C Language Offline Compiler. Version 0.9 (April. 2015)." Github repository. http://github.com/HSAFoundation/CLOCs.

[30] Thread Building Blocks (TBB). https://www.threadingbuildingblocks.org/

[31] S. Tomov, J. Dongarra, V. Volkov, and J. Demmel. MAGMA Library, version 0.1. http://icl.cs.utk.edu/magma, 08/2009.

[32] S.Tzeng, A. Patney, and J. D. Owens. Task Management for Irregular-Parallel Workloads on the GPU, High Performance Graphics,2010

[33] L. G. Valiant, "A bridging model for parallel computation," Communications. ACM, vol. 33, no. 8, pp. 103-111, 1990.