Northup: Divide-and-Conquer Programming in Systems with Heterogeneous Memories and Processors

Shuai Che *
cheshuai@gmail.com

Jieming Yin
Advanced Micro Devices, Inc.
jieming.yin@amd.com

***Abstract*—In recent years we have seen rapid development in both frontiers of emerging memory technologies and accelerator architectures. Future memory systems are becoming deeper and more heterogeneous. Adopting NVM and die-stacked DRAM on each HPC node is a new trend of development. On the other hand, GPUs and many-core processors have been widely deployed in today's supercomputers. However, software for programming and managing a system that consists of heterogeneous memories and processors is still in its very early stage of development. How to exploit such deep memory hierarchy and heterogeneous processors with minimal programming effort is an important issue to address. In this paper, we propose *Northup*, a programming and runtime framework, using a divide-and-conquer approach to map an application efficiently to heterogeneous systems. The proposed solution presents a portable layer that abstracts the system architecture, providing flexibility to support easy integration of new memories and processor nodes. We show that Northup out-of-core execution with SSD is only an average of 17% slower than in-memory processing for the evaluated applications.**

***Index Terms*—accelerator, heterogeneous memory architecture, programming framework, recursive execution**

I. Introduction

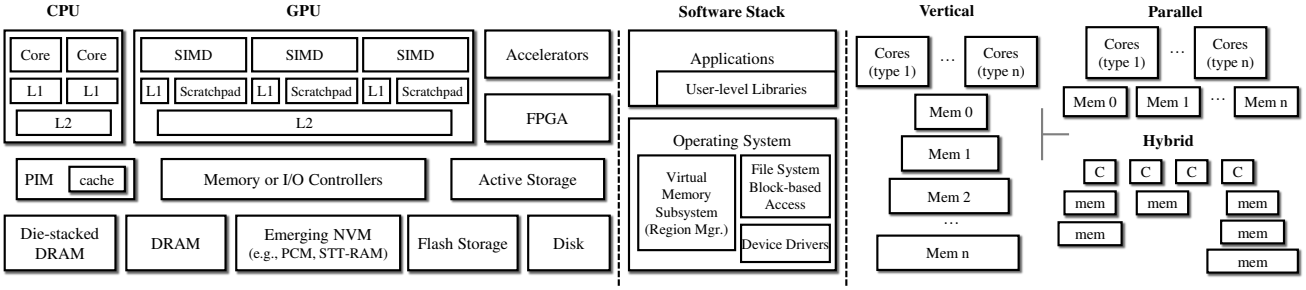
The need for ever more computational power in HPC has never stopped as researchers try to address deeper science problems [1]. The design of future Exascale systems must tackle many challenges. Among these issues, exploiting heterogeneity has become a hot research topic for years [1]–[3]. Inside a machine node, heterogeneity originates from two major sources: processing elements and memories. On the one hand, systems with CPUs, GPUs, and FPGAs are ubiquitous; and these compute elements exhibit dramatic disparities in their compute capabilities. On the other hand, memory hierarchy becomes deeper and more heterogeneous with multiple memory nodes and layers, thanks to the development of die-stacked memories [4], processors in memory (PIM), and non-volatile memories (NVMs). These new memory technologies also present diverse latency and capacity characteristics [5]. As a result, it is not surprising that designers will come up with diverse heterogeneous design options. Moreover, these architectures may be very challenging to program, including using different APIs for different parts of computation and memory management (e.g., CUDA, OpenCL). This leads to complex, unportable, and hard-to-manage code. Thus, a generic strategy for application development is needed and the solution should adapt to diverse architectures.

To achieve efficient parallelism and data movement, we need to rethink the design of today's software stack and programming model. To be specific, regardless how architectures are designed, memory and storage are hierarchically organized in multiple levels: from slow, large capacity to fast, small capacity; and a processor (e.g., CPU, GPU, and FPGA) is attached to a memory node and issues memory requests. The only difference among system designs is the shape of the overall system topology—this is also how programmers usually view a system architecture. Based on this observation, the entire system can then be abstracted in a hierarchical tree structure and in natural synergy with the divide-and-conquer way of problem solving. In other words, a large problem is recursively decomposed into multiple sub-problems, and the required data is moved along the tree edges from the slowest storage, through faster memory, all the way to the caches of processing elements. Parallel computation for the smallest data decomposition is exploited with diverse accelerators at tree leafs. In the end, the solutions of subproblems are combined to generate the final result. A solution based on divide-and-conquer has several benefits. First, an algorithm is constructed in a way that can naturally map well to a system regardless of its architectural topology. Second, the developed application presents good portability (in terms of both code and performance) as long as efficient runtime support is provided for scheduling and load balancing. Once the code is written, it should work across heterogeneous architectures.

To achieve the above goals, we present **Northup**, a programming and runtime framework to support diverse heterogeneous hardware components. In this work, we revisit how applications and algorithms should be developed to meet performance as well as portability needs; we also address new challenges in managing system heterogeneity (e.g., organization of heterogeneous compute and memory components, data movement, non-unified memory management API). Northup is particularly effective in dealing with various uncertainties as architectures evolve rapidly. This paper makes the following contributions:

- We describe how Northup abstracts the system architecture in an asymmetric, hierarchical tree structure, consisting of both heterogeneous memory and accelerator nodes.
- We present a recursive programming style, mapping an algorithm to the underlying tree-based system architecture that is transparent to the programmer.
- We propose an API and runtime techniques to simplify data movement and ensure code portability.
- We demonstrate the usefulness of Northup using a few representative algorithms as case studies.

* The work was performed while Shuai Che was at AMD.



a) Hardware View b) Software View c) Abstract View

Fig. 1: An architecture consisting of heterogeneous memories and processors.

II. Motivation

We first describe the recent development of heterogeneous systems and associated challenges. This paper uses the words "memory" and "storage" interchangeably–both referring to a space where data is stored either permanently or temporarily.

Heterogeneous Memory Hierarchy: New memory technologies such as die-stacked DRAM [6] and NVM [5] provide new memory characteristics and features. In the near future, programmers may face the challenge of programming a rather complex system consisting of diverse heterogeneous memory components. Figure 1a) shows how such a system may look like from a hardware point of view. Figure 1b) shows the components involved in the software stack. In addition, conceptually we can deem such a system in an abstract view as shown in Figure 1c). At a high level, memory system has become increasingly complicated in several aspects. Firstly, the memory hierarchy design presents many alternatives, as a level of memory can be used for backing store, for caching, or for NUMA accesses in parallel with another level. Secondly, different memory subsystems may present heterogeneity and different characteristics. For example, a GPU may have its own memory subsystem [7] (similarly for PIM and Active Storage). Finally, the interfaces to manage these memories can be very different, resulting in a non-trivial effort to manage memory mapping and data movement. *As a result, a simple system abstraction and programming model is crucial to programming heterogeneous systems.*

Hardware versus Software-managed memory: Software-managed memories can be controlled by the operating system, compilers, user-level libraries, application programs or their combinations. A typical memory organization has one or multiple "transitioning" levels as a division point for in-core and out-of-core memory management. The selection of transitioning points has an important implication on how programmers view the whole system, which in turn affects what hardware features are exposed to software and the development of software stack. Recent development in GPGPU and embedded systems has reflected a trend that more memories are designed to be software-managed. The goal is to provide programmers more control and achieve high performance. *Given increasing number of memory levels, we envision that programmers will take more responsibilities to deal with explicit data movement.*

Application Portability: There have been different proposals treating new memories in different ways. For example, stacked DRAM can be used as a high-bandwidth, hardware-managed last-level cache [4]. Alternatively, it can be a part of the physical memory space in parallel to DRAM and managed by the OS [3]. Furthermore, memory can be exposed to software in different ways. A design can treat the NVM as part of physical address space (i.e., accessible through a load/store hardware interface) [8], [9] or as fast storage [10]. In reality, a system configuration is dependent on different requirements and the compatibility to legacy software; however, there is no general guidelines how to configure a system. This makes application development a challenge because applications assuming one model may not be portable with another model. Although works have been done to standardize the model for specific areas [11], a holistic design approach is needed. *Since different types of systems are likely to coexist in future, applications should be portable across platforms and exploit different memory hierarchies efficiently.*

Heterogeneous Compute: GPUs, FPGAs and other co-processors demonstrate better compute efficiency for many applications compared to CPUs. However, to fully utilize them requires significant programming efforts, especially in the presence of heterogeneous memories. One issue is that different processors may not "see" the memory system in a uniform way. For example, a GPU may only access its own device memory in certain systems. There have been efforts to improve this issue, but additional work remains to be done. Recent development of HSA [2] allows host and device to share the same virtual address space and supports the "a-pointer-is-a-pointer" semantic. Also, initial work has been proposed to allow GPU threads to directly access files and I/Os [12]. *Future programming environments should incorporate similar features for better programmability; and ideally all processing elements should be treated in the same way.*

III. Northup

This section describes the design of Northup, which presents several design advantages:

- **Efficient Computation and Data Mapping**. Northup uses a divide-and-conquer approach to enable natural and efficient mapping of computation and data to diverse hierarchically-organized hardware components.
- **System-topology Aware**. Programmers are not required to directly deal with the underlying system topology and hardware details. API is provided to obtain useful mem-

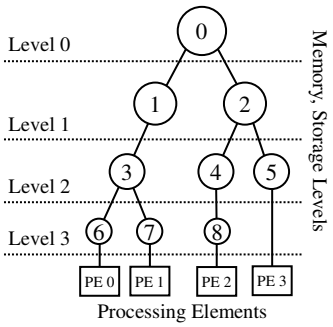


Fig. 2: Asymmetric tree structure representing the system topology. Each circle represents a memory node while each rectangle represents a processor (e.g., CPU, GPU, FPGA).

ory node information whenever needed (e.g., scheduling and performance tuning).

- **Improved Portability**. A unified interface for managing data movement is proposed to improve portability, in contrast to existing models where the storage and memories are exposed to software in different ways. This enables an application to work across different memory hierarchies.
- **Computation and Data Movement Decoupling**. Programming for computation and data movement is decoupled and treated separately, which leads cleaner code.

A. A Recursive Model

Modern computer systems as well as large machine clusters are organized in a hierarchical manner. Typically, in a parallel algorithm, different regions of data are scheduled in a particular order defined by the algorithm and loaded from slower (higher-level) memories to faster (lower-level) memories. This process is repeated until all regions are processed. Similarly, in Northup we break a large problem into sub-problems that are themselves smaller instances of the same problem. We then recursively solve these sub-problems that can fit into faster memories, and finally combine the results. This concept is discussed in Section III-C.

There has been a large body of prior work on recursive algorithms, but in other contexts. Anderson *et al*. [13] and Dongarra *et al*. [14] present linear-algebra algorithms, which exploit the deep memory hierarchy of microprocessors. Yi *et al*. [15] propose compiler techniques to transform iterative codes into a recursive format. Sequoia [16] is a programming model and runtime which allows programmers to explicitly program the memory system and application decomposition. Virtually all the iterative algorithms can be transformed into their recursive counterparts. With the advent of new processor and memory technologies, the recursive approach can be very effective and convenient for application-to-architecture mapping for increasingly complex systems. Inspired by these works, we target a generic, divide-and-conquer approach to map parallel applications to heterogeneous systems, with better heterogeneous resource management.

B. Topological Tree

Northup provides generic support for software-managed memories. To be specific, Northup is capable of differentiating memory spaces regardless of shared or private memory and supporting explicit data movement between memories. In Northup, the system is abstracted in a hierarchical tree structure as shown in Figure 2. In contrast to prior work [16], this tree can be asymmetric and heterogeneous, and further include new structures to store the information for memory and processor nodes. Each tree node is associated with a unique identifier. We number the memory and storage levels in a classic way: the slowest storage is assigned "0" while faster storages are assigned with larger numbers. In this way, physical memory and storage that map to logical tree nodes are virtualized, easy for management. This virtual-to-physical mapping can be reconfigured in different use cases (e.g., NVM for physical memory or storage). In the tree, an inner node or root represents an intermediate memory or storage, while a leaf node represents the transition point from the software to hardware-managed memory [16]. Each leaf node is attached to a processor with its own hardware cache, and computation happens at leafs. Note that there is an exception: the CPU can attach to a non-leaf node in a CPU + discrete GPU system.

Listing 1 Data structure of tree nodes

```
node {
 memory_t{
    int storage_type;
    int physical_id;
    int capacity, used; ... }
  processor_t{  //only for leaf nodes
    int processor_type;
    int LLC_size;... };
 int level;
 int node_id;
 bool isLeaf;
 ...
 struct node *parent;
 struct node *children[numBranches];
 list *work_queue[numQueues]; ...};
```

The Northup tree can be maintained by system software or constructed by the runtime library at program initialization. Listing 1 shows a sample data structure of a tree node. A leaf node also contains the information of the attached processor. Data management and job scheduling can refer to this tree structure. For example, by checking the *storage_type* of source and destination, a data movement function internally can determine the correct data copy function to use (e.g., DMA or I/O function). By examining the capacity and usage, a program can decide the blocking size. The tree node can also store the links to work queues which keep track of the recursive tasks; and this allows for the implementation of load balancing across different tree branches (see Section V-E). Finally, a node also includes the pointers to its children. Northup provides various functions to query the Northup tree. For example, $fetch\_node\_type()$ obtains the storage type; $get\_parent()$ and $get\_children\_list()$ returns the parent and the children nodes of a specific node, respectively; $get\_cur\_treenode()$ returns the current tree node id (useful for indexing data), $get\_level()$ provides the current memory level, and $get\_max\_treelevel()$ provides the total number of tree levels.

C. A Recursive Algorithm Template

Listing 2 shows an example of regular OpenCL/CUDA-like code where each element in the matrix is computed in a

Listing 2 Regular pseudocode

```
void myfunction(arg...){
  L0_chunk_size_x = dim_x/L0_x
  L0_chunk_size_x = dim_x/L1_x
  for(i = 0; i < L0_x; i++)
    for(j = 0; j < L0_y; j++)
      //storage allocation code for each chunk
      ...
      L0_size = L0_chunk_size_x * L0_chunk_size_y
      //allocate memory buffer
      buffer = malloc(L0_size * sizeof(T))
      result = malloc(L0_size * sizeof(T))
      file_open(fd[i * L0_y + j], .. )
      file_read(buffer , L0_size,  fd[i * L0_y + j]);
      ...
      for(m = 0; m < L1_x; m++)
        for(n = 0; n < L1_y; n++)
          //set up device context
          ...
          //allocate OpenCL buffer
          dMalloc(d_contex, d_buffer, size, ...)

          dCopyBlockH2D(d_buffer,       //dst
                        buffer,         //src base
                        m * L1_chunk_x, //offset x
                        L1_chunk_x,     //size
                        n * L1_chunk_y, //offset y
                        L1_chunk_y)
          ...
          dLaunchComputation(d_buffer, d_result, ...)
          dCopyBlockD2H(d_result,       //src
                        result,         //dst
                        m * L1_chunk_x, //offset x
                        L1_chunk_x,     //size
                        n * L1_chunk_y, //offset y
                        L1_chunk_y)
        ...
        file_write(result, L0_size, fd_r[i*L0_y+j])
}
```

Listing 3 Northup pseudocode

```
void compute_task(buffer, result){
  if get_device() == GPU
    node = get_cur_treenode()
    dLaunchComputation(buffer[node], result[node])
  else if get_device() == CPU
    ...
}
void setup_buffer(buffer, ...){
  ...
  node = get_cur_treenode()
  buffer[node] = alloc(size, node)
  result[node] = alloc(size, node)
}
void data_down(buffer,...){ //move data to children
  ...
  node_src = get_cur_treenode()
  node_dst = node_src.get_children_list()[0]
  move_data_down(buffer[node_dst],
                 buffer[node_src] + index(m,n), ...)
}
void data_up(result, ...){ //move data to parent
  node_src = get_cur_treenode()
  node_dst = node_src.get_parent()
  move_data_up(buffer[node_dst] + index(m, n),
               buffer[node_src], ...)
}
int myfunction(buffer, result, ...){
  if (get_level()) == get_max_treelevel()
    compute_task() //computation at leaf nodes
  else
   #pragma for all (m, n)
    for m in [0, get_x(get_cur_treenode())]
      for n in [0, get_y(get_cur_treenode())]
        setup_buffer(buffer, result, ...)
        data_down(buffer, result, ...)
        northup_spawn(myfunction(buffer, result, ...))
        data_up(result, ...)
}
```

massively-parallel fashion. A matrix at memory level 0 (disk) is divided into $L0\_x \times L0\_y$ chunks. Each chunk is moved to level 1 (a DRAM buffer) and further divided into $L1\_x \times L1\_y$ smaller chunks. Each smaller chunk is moved to the GPU device memory and a GPU kernel is launched. Note that the code will NOT work if adding a new memory level or changing to another heterogeneous architecture. In contrast, the equivalent Northup code (Listing 3) works on arbitrary heterogeneous systems and can efficiently map to hardware.

In Northup, a typical application is written in a way similar to the pseudocode in Listing 3. The major functionality is expressed in a recursive function. Inside the function, it will first check whether the execution reaches a leaf node. At a leaf node, the program launches computation on specific processors. Level is incremented in each recursion. Based on the Northup tree, the runtime keeps track which storage node the program has reached. At an intermediate level i, it will first set up buffers for the next level $i+1$ using $setup\_buffers()$ and break the data at level i into multiple chunks. The $get\_x()$ and $get\_y()$ functions obtain the dimensions of chunk decomposition in x and y directions. At a given tree node ($get\_cur\_treenode()$), there are a total of $get\_x() \times get\_y()$ chunks to process, and index() calculates the location (e.g., array offset) from where the data movement (up or down) should be performed for a specific chunk. Each chunk is moved to the next level $i+1$ through $data\_down()$. The number of chunks depends on the current available capacity of level $i+1$ and size of the data structure. The program then enters a for-loop performing the same task for each chunk for the next level, by launching recursive calls for the chunks at level $i+1$. Finally, after the call, it moves the result back to level i through function $data\_up()$.

Theoretically, chunks can move and execute at lower memory levels concurrently. However, in reality they may execute sequentially or only a few chunks can move concurrently (along the same tree branch) due to smaller capacities of lower memory levels. Alternatively, level i can spawn multiple tasks each processing one chunk to one of its children at level $i+1$ (e.g., multiple tree branches). For example, in Figure 2, node 3 has two children 6 and 7. We also support task queues to keep track of the progress of data movement for individual chunks. The tasks can be pushed into the per-memory-level queue for scheduling. Given n chunks at level i, n tasks will be enqueued. This enables multi-stage data transfer and better parallelism. Whenever the space of lower memory levels is freed, more chunks can be scheduled for movement. Data transfer optimization is further made for overlapping computation and communications (i.e., OpenCL/CUDA streams) at the leaf node. The recursive tree can be further unfolded to a dependency graph to exploit more parallelism, which we leave for future work.

Note that recursive execution may present the risk of stack overflow. However, this is not an issue for Northup, because Northup recursion is at a coarse-grained level; the total number of recursion levels (i.e., number of memory levels) is small enough such that this issue will not happen in practice if programmed appropriately.

D. Data Management API and Runtime

In the above code example, programmers or library writers need to define their own data management functions, i.e., $setup\_buffers()$, $data\_down()$, and $data\_up()$. Northup provides buffer allocation and data movement functions for users. As discussed, APIs for different memory and storage nodes (I/O, virtual memory, or specialize memories) are dramatically different. An implementation for one system architecture may not be portable to another. For example, the heap memory can be managed by libc or POSIX memory calls while file I/Os can be operated with a FILE type or file descriptors. Similarly, GPU buffers are declared with special data types (e.g., cl_mem in OpenCL) and managed by special functions.

We provide a high-level abstraction to resolve the diversity of memories/storages. In particular, a unified interface is provided in Northup to deal with different types. The key is that all buffers are associated with the *same* opaque type for portability, and data allocation and movement are achieved with a generic interface. We show an exemplary solution in Table I. For some functions, we make extensions by including tree nodes as source and destination. The runtime system determines the appropriate operations to perform based on the levels and types of tree nodes involved in the function. Alternatively, based on the current memory level it may walk up and down the tree (e.g., with implicit args for source and destination). For demonstration purpose, the current implementation uses void pointers. The data_move function (e.g., a wrapper) internally decides how to dereference pointers depending on the involved storages and types for that specific call. Better support of type safety and C++11 like semantics are left for future work. In actual implementation, a specific universal type (e.g., a $UniversalType$) can be designed for a programming language.

The implementation is transparent to users. For example, if node 0 is a file-based storage, $Alloc(size, node\ 0)$ allocates space on a disk drive and return a void pointer. The implementation of $Alloc()$ may launch the $open()$ system call to return a file descriptor fd. One simple solution is the returned void pointer will point to the actual memory address that stores the fd. Subsequent data movements (e.g., $move\_data()$) also use void pointers. The fd, dereferenced with the pointer, is used for $read/write()$ system calls. Similarly, the same applies to OpenCL cl_mem for GPU memories. Listing 4 shows the pseudocode for the $move\_data()$ wrapper function. It internally examines the source and destination tree nodes. In this case, if writes are from the main memory to the file storage, it calls $file\_write()$, which in turn calls the POSIX $write()$ with file descriptor pointed by the void pointer. Note that $data\_down/up()$ does not necessarily map to data copies; it can be implemented with memory mapping functions too.

For different scenarios, configurations can be made to customize data management. For example, there are different modes to operate a file (e.g., caching). Since our applications directly manage storage data, flags are set to minimize kernel caching effects of I/O (directly to/from user-space buffers) and also guarantee that the call is synchronous when writing to the storage (e.g., O_DIRECT and O_SYNC). The Northup runtime also maintains internal structures in order to implement a universal interface. For example, $Alloc(size, node\ 0)$ allocates space on the disk drive by generating a file. Thus for resource management, we need to maintain a list of file names and their pointers to all the created spaces.

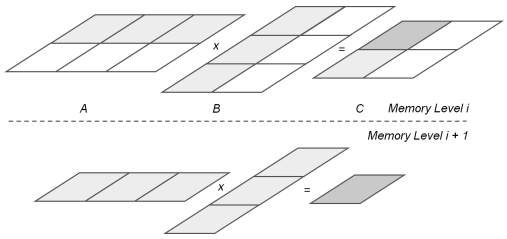


Fig. 3: Dense matrix multiply $C = AB$. The regions of matrices (in gray) are moved from memory level i to level $i+1$.

E. Computation and Acceleration

When data reaches a leaf node, computation is initiated on the processor attached to the leaf. Computation for different accelerators (e.g., CUDA and OpenCL) can be pre-compiled, or a single piece of code (OpenMP, C++ AMP) can be compiled into different architecture targets. At a leaf node, programmers can query the processor type (see Section III-B) and launches the computation kernel on a desirable device. If a leaf node is associated with more than one processor (e.g., the CPU and the GPU in an APU), computation can be launched on either processor or work can be spread across devices in a data-parallel fashion (see Section V-E). One benefit of using Northup is that data movement and computation are decoupled and treated independently. Therefore, programmers can optimize their code independently in *move_data* and *compute_task* functions. This leads to a big advantage: the latest research on optimizing algorithms (e.g., GPGPU) can be easily plugged into Northup. In addition, Northup has another advantage: it is easy to determine the most appropriate processor to run the task. By profiling the execution of earlier scheduled chunks, the system can provide useful information to subsequent scheduling and task-processor mapping. To relate the information to logical problem decomposition steps, programmers still need to understand the overall system topology and required levels to map algorithms (Northup can output the topology). Nevertheless, Northup can help programmers write clean and portable codes.

IV. Algorithm Development

We use several representative, state-of-the-art algorithms to demonstrate the key ideas of Northup. But this framework is generic to a variety of problems.

A. Dense Matrix Multiply

Dense matrix multiply ($C = AB$) is one of the most important operations in numerical analysis, deep learning, and data analytics. We extend an optimized, tiled version of GPU dense matrix multiply [17] for Northup out-of-core execution. The baseline OpenCL version we use is able to achieve more than 80% of peak GPU FLOPS.

In dense matrix multiply, each recursive level performs the following steps starting from root level 0:

TABLE I: Unified Data Management Interface. Sample Functions.

Sample programming interface	Description
(void *)alloc(size_t size, int tree_node)	allocate space on a memory or storage node and return a void pointer
void move_data(void *dst, void *src, size_t size, size_t offset, int dst_tree_node, int src_tree_node)	move data from the source to the destination tree node
void move_data_down(void *dst, void *src, size_t size, size_t offset, int i)	move data to ith the child node, assuming the current node as the parent
void move_data_up(void *dst, void *src, size_t size, size_t offset)	move data to the parent node, assuming the current as a child
void release(void * ptr)	release the storage or memory space pointed by ptr

Listing 4 Pseudocode for interface implementation.

```
// This example only shows the code snippet for file I/Os.
// Other cases (e.g., OpenCL buffers, memcopy and different combinations of dst/src) use a similar approach.
void file_write(int fd, void *buf, size_t count, size_t offset) {
    lseek(fd, offset, SEEK_SET); //using the linux system calls
    cnt = write(fd, buf, count);
}
void move_data(void *dst, void *src, size_t count, long offset, int dst_tree_node, int src_tree_node) {
    //obtain the information of both the destination and source tree nodes
    if ((fetch_node_type(dst_tree_node) == FILE_TYPE) && (fetch_node_type(dst_tree_node) == MEM_TYPE)){
        int fd = *(int*)dst;  //get the file descriptor pointed by ptr
        file_write(fd, src, count, offset);  //launch the file write operation}
    //other cases ...
}
```

- Determine if the execution reaches a leaf (i.e., $l = max\_level$) where l represents level. If so, conduct matrix multiply on the smallest sub-matrices on a processor.
- Otherwise, divide the current input matrices $A(l)$ and $B(l)$ into smaller sub-matrices, each with a size of $dim_{A-x}(l) \times dim_{A-y}(l)$ and $dim_{B-x}(l) \times dim_{B-y}(l)$, respectively (where $dim_{A-y}(l) = dim_{B-x}(l)$). To compute each sub-matrix $C_{i,j}(l)$, first set up data buffers for a row of sub-matrices extracted from the ith row strip of $A(l)$, and a column of sub-matrices extracted from the jth column strip of $B(l)$. We call these two strips a row and a column shard, respectively. Then move the row and column shards at level l to level $l+1$.
- For all sub-matrices, launch dense-matrix multiply recursively, and finally copy the result matrix $C_{i,j}(l)$ back from level $l+1$ to level l.

In Figure 3, the program loads the two shards from the slower memory to the faster memory. Similarly, "dot product" at the block level (i.e., sub-matrix) will be performed by 1) first computing partial results from the corresponding blocks of A and B, and then 2) accumulate the partial sums. There is one immediate optimization that we can apply. For row shard m, it is reused multiple times to calculate the sub-matrix (m, j), where j is $[0, num\_column\_shards)$. Therefore, the row shard m can stay in the $l+1$ level and the program just iteratively loads column shards to the next level. Similar process is repeated for $m+1$, $m+2$, and so on.

At leafs, computation is executed on the GPU. We leverage the GPU's per-compute-unit (CU) local memory (shared memory in CUDA) for each data block and manage the data movement explicitly. Note that the GPU on-chip data movement may also be integrated into Northup's recursive model. However, this requires additional compiler support, which we leave for future work. A $4k \times 4k$ blocking size is used in DRAM, and 16×16 blocking size is used in GPU local memory. These sizes are manually chosen through experimentation while further performance tuning can be applied.

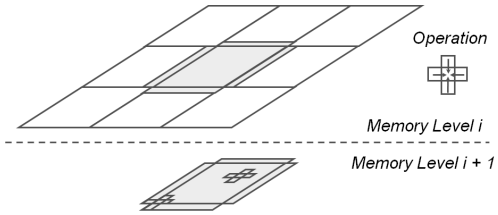


Fig. 4: Each HotSpot-2D block and its four borders are loaded to the memory level $i+1$ for computation.

B. Thermal Simulation (Stencil)

Stencil operations, a key building block in HPC (e.g., molecular dynamics, thermal simulation), are widely used in applications solving partial differential equations. We use the HotSpot-2D [18] thermal simulation for demonstration, which we extend to out-of-core Northup execution. HotSpot-2D achieves about 8× speedup on the GPU compared to the CPU and is also a benchmark in SPEC ACCEL [19].

In Northup HotSpot-2D, each recursive level performs the following steps starting from level 0:

- Determine if the execution reaches the leaf level. If so, conduct HotSpot on the smallest block on a processor.
- Otherwise, divide the current block matrix into smaller sub-blocks each with a size of $dim(l) \times dim(l)$, where l is the current level. For each sub-block, the data structure includes two $dim(l) \times dim(l)$ matrices (input and output), two compact vectors for four borders of the sub-block. We then move these data from level l to level $l+1$.
- For all the sub-blocks, launch the stencil computation recursively, and copy the data of the result matrix from level $l+1$ to level l.

In Figure 4, one issue is some border elements of each block are stored non-contiguously (east/western borders), which leads to inefficient memory accesses. We allocate vector buffers and pack the border data in a contiguous manner and pass them to the next level.

At leafs, we use the OpenCL HotSpot-2D [18] for GPU computation. Given the data in the leaf memory, we launch a GPU kernel with a number of 2-D workgroups, and each

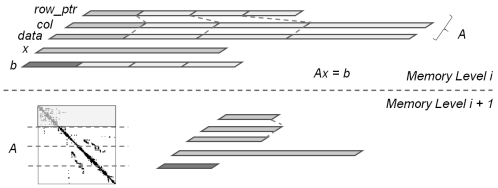


Fig. 5: The regions of SpMV vectors (in gray) are moved from memory level i to memory level $i+1$ for computation.

workgroup is responsible for processing a small block. We load the data block and its borders (that can fit into the on-chip cache) to $(block\_size+2)\times(block\_size+2)$ local-memory arrays for better locality and low latency. An $8k\times 8k$ blocking size is used in DRAM, and 16×16 blocking size is used in the GPU local memory.

C. CSR-Adaptive (SpMV)

Sparse-matrix vector multiplication (SpMV) is an important primitive in HPC and enterprise computing. It multiplies a sparse matrix and a dense vector, and generates a dense vector (i.e., $Ax=b$). We use the CSR-Adaptive algorithm as the baseline [20], which is implemented in OpenCL and achieves a 4.5× speedup compared to the SpMV routine in cuSPARSE 7.5. CSR uses three compact vectors to represent a sparse matrix: *row_ptr*, *col_id* and *data*. We break these data structures in the row dimension of the matrix into smaller chunks; and each chunk with multiple rows is called a shard.

In Northup CSR-Adaptive, each recursive level performs the following steps, starting from the root level 0:

- Determine whether program execution reaches a leaf node. If so, conduct SpMV computation for the shard on a processor.
- Otherwise, divide the shard in the current level further into smaller sub-shards. Each sub-shard contains a subset of rows in the current shard (i.e., a portion of *row_ptr*, *col_id* and *data*). For *col_id* and *data*, the portion of data constituting a sub-shard is determined with $row\_ptr[start]$ and $row\_ptr[end]$), where *start* and *end* mark the starting and ending offsets of the rows in the shard. We then move sub-shards from level l to level $l+1$.
- For each sub-shard, we launch CSR-Adaptive recursively, and copy the result back from level $l+1$ to level l.

A simple strategy is to evenly divide rows (see Figure 5). However, it is possible that different rows (chunks of rows) have different number of non-zeros (*nnzs*). This information can be easily calculated (e.g., $row\_ptr[end]-row\_ptr[start]$) to determine the desirable size to move to the next-level. For example, if the *nnz* of a shard is too large to fit in the next-level memory, it can be further broken into smaller shards. Northup has a unique advantage to handle this situation thanks to its recursive scheme. Note that one requirement for SpMV is the fastest memory has to be big enough to hold the vector b. Therefore, it may be useful to provide a solution, combining programmers-controlled (e.g., Northup) and system/hardware-controlled memory management. Once a shard is mapped to a leaf node, we use CSR-Adaptive [20], which dynamically choose kernels based on the shapes of sparse matrices for optimized performance. The inputs we used have 16 million rows, stored in SSD/disk drive. The matrix is divided into four chunks in row-dimension to load into DRAM.

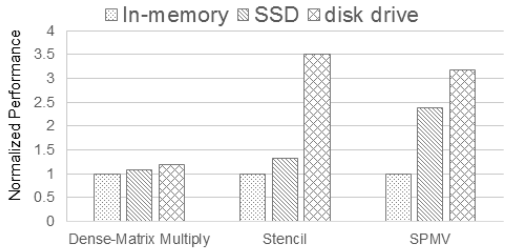


Fig. 6: Performance comparison: in-memory processing, SSD, and disk-drive.The y-axis represents the normalized runtime (slowdown) to the baseline.

V. Experimental Results

A. Experimental Setup

The experiments use AMD A10-7960k APU and A10-7850k + FirePro W9100 GPU. We use Ubuntu 16.04, and HSA with SNACK (OpenCL kernels) [21] for APU development while ROCm [22] with OpenCL for discrete GPU development. We use an entry-level HyperX Predator PCI-E SSD with a speed up to (1400, 600) MB/s, (read, write) performance. Our hard drive is a SATA-based Western Digital WD5000AAKX. A 2 GB of main memory is configured as a staging buffer for out-of-core execution. For in-memory processing, we use 16 GB memory holding the entire working set. For inputs, we use $16k\times 16k$ and $32k\times 32k$ float matrices for dense-matrix multiply and HotSpot. The SpMV inputs are from the Florida sparse-matrix collection [23]. The same algorithms are used in the Northup leafs and in the original non-Northup baseline.

B. Baseline in Memory versus Northup Execution

We first compare Northup (out-of-core) against the baseline in-memory processing [17], [18], [20], on an APU with two levels of Northup-managed storage: main memory and file storage (SSD or disk drive). For in-memory processing, we assume all the data is already loaded into memory while for out-of-core processing we use the same input and assume that a program starts execution from the storage level (the tree root). For the latter, there is a one-time overhead of preprocessing the original file and reorganizing it in one or multiple files for chunking. Preprocessing is a common practice in distributed systems for HPC and data analysis.

Figure 6 shows the performance comparison of in-memory processing, SSD and disk drive. With Northup, all three algorithms are able to exploit multiple memory layers, despite the fact that they have different characteristics. Dense-matrix multiply is the most compute-intensive among the three with high data reuse. Therefore, the overhead of accessing slow storages can be effectively hidden. For the more memory-intensive HotSpot-2D and CSR-Adaptive, there is little computation and data reuse to hide latency. They experience a 2-2.5x slow-down switching from in-memory to Northup

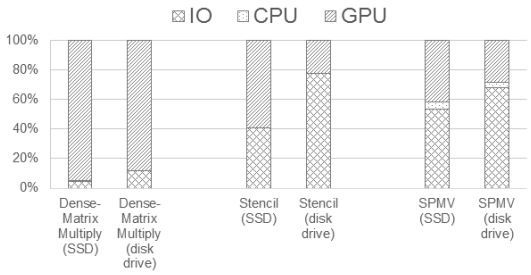


Fig. 7: Execution breakdown due to CPU, GPU and buffer setup, transfers and I/Os. Two-level Northup tree: main memory and disk drive/SSD. The processor is the GPU on an APU.

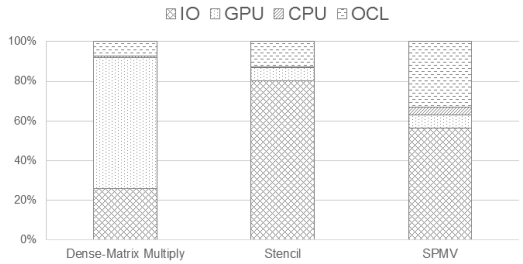


Fig. 8: Execution time breakdown for a discrete GPU system. The Northup tree has three levels: GPU device memory, main memory, and disk drive.

processing with a disk drive. Using a faster SSD, HotSpot-2D and CSR-Adaptive exhibit 0.3-1.4x slow down. HotSpot-2D obtains more performance benefit than CSR-Adaptive, because it uses relatively regular blocks with better I/O performance as compared to variable buffer sizes by CSR-Adaptive.

Dense-matrix multiply and CSR-Adaptive tend to lie on two extremes in terms of computational intensity and randomness of data accesses. We hypothesize that the performance of many applications will lie within the spectrum of what we have experimented. Also, with faster storage and new memory technologies, which we discuss later in Section V-D, performance can be further improved. In addition, our runtime is lightweight. It makes two levels of recursive calls in an APU system (three levels in a discrete GPU system) with additional task control and lookup on the Northup tree. Since a typical system only consists of a few memory levels, the runtime overhead is quite low. Overall, the measurement shows the runtime overhead is less than 1% of the total execution time. Of course, this also depends on the chosen blocking sizes– overly fine-grained problem decomposition results in many calls and low hardware utilization. Our manually selected blocking sizes (Section VI) are small enough to fit into the storage and big enough to fully utilize the GPU.

C. Execution Breakdown

We break down the overall Northup execution time into CPU and GPU execution, buffer setup, and data transfers such as I/Os (e.g., open, read/write, close). Since computation is offloaded to GPU , CPU computation is a small portion in these algorithms. CSR-Adaptive uses the CPU for binning rows into different categories and spends relatively more

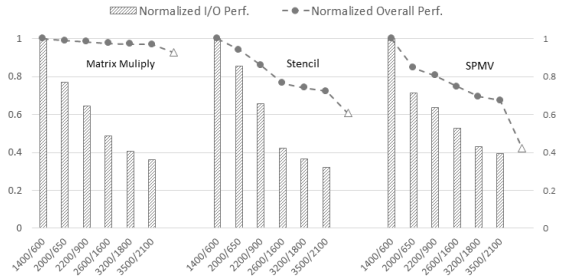


Fig. 9: I/O and overall performance using faster storages. All numbers are normalized to the 1400/600 SSD case with 2-level hierarchy and GPU processing (in APU). The Δ points represent the performance of the in-memory version.

time. In Figure 7, using the disk-drive configuration, dense-matrix multiply spends majority of time on GPU computation, while HotSpot-2D and CSR-Adaptive spend 22% and 28%, respectively. This again suggests that HotSpot-2D and CSR-Adaptive are more sensitive to memory/storage bandwidth. They can obtain significant benefit if switching to new memory technologies such as stacked DRAM and NVM. By switching to SSD, the portion spent on GPU computation becomes 59% and 41% for HotSpot-2D and CSR-Adaptive, respectively. We conduct another experiment with three levels of tree nodes as shown in Figure 8. In this experiment, we offload computation to a discrete GPU with an additional disjoint memory space compared to the previous setup. OpenCL transfers contribute 7%, 12%, and 33% of execution time for dense-matrix multiply, HotSpot-2D and CSR-Adaptive.

D. Performance Benefits with Faster Storages

The PCI-E SSD used in this study has a sequential read-/write bandwidth of 1400/600 MB/s. To quantify the potential benefits of Northup with faster storage, we develop an emulator capable of performing a first-order projection by keeping track of read/writes issued by application I/Os and considering read/write bandwidths of the storage. We also include the I/O time into the overall runtime (the other components being constant). We consider a spectrum of configurations with different (read/write) bandwidths, ranging from (1400/600) to (3500/2100) representing some fastest PCE-E SSDs on the market. In Figure 9, the gain of switching to faster hardware is significant, especially for memory-intensive workloads such as HotSpot-2D and CSR-Adaptive—an improvement of 65% for I/O and 30% for overall performance. We still have not used other NVMs, which may provide faster performance. Our baseline in-memory OpenCL version assumes all the data is ready in DRAM and excludes I/O for execution time measurement. Therefore, it is considered to be the performance upper-bound that Northup can achieve. The performance differences between Northup (with SSD) and in-memory versions are 5%, 15%, and 30% for dense-matrix multiply, HotSpot-2D, and CSR-Adaptive, respectively. Faster NVMs may further improve Northup's performance and bridge the gap.

One key takeaway is with emerging memory technologies, the extremely wide gap between DRAM and storage (SS-

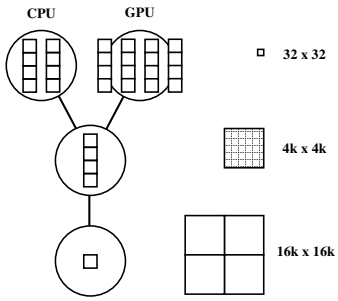


Fig. 10: Work queue organization and data decomposition.

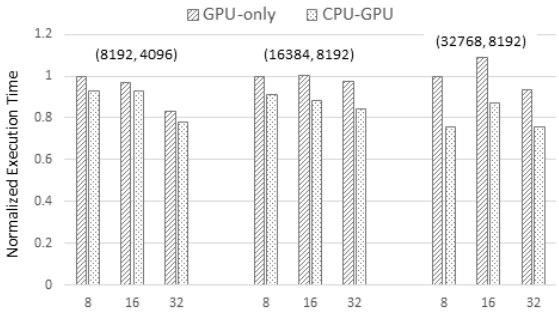


Fig. 11: HotSpot-2D (Stencil) performance of load balancing (APU+main memory+SSD) normalized to GPU-only Northup execution (GPU+main memory+SSD). For each input point, 8, 16, and 32 queues are experimented for the GPU.

D/disk drive) can be filled for better performance. Similarly, the same applies to die-stacked memory which fills the gap between SRAM and DRAM. This results in a future memory hierarchy with a good coverage of memory and storage with different latency/bandwidth spectrum, allowing more effective latency hiding across memory levels than before. Thus it makes Northup a promising framework to program and schedule applications.

E. Task Queuing and Load Balancing

Future systems will exhibit an asymmetric memory hierarchy in Figure 2. The system is subject to load imbalance when uneven workloads are assigned to different subtrees. Northup's topological tree structure is able to naturally support dynamic load balancing when tree nodes store information such as on-going tasks at different subtrees. Each tree node can be associated with one or multiple work queues, keeping track of the progress of the recursive tasks. In particular, examining the status of a subsystem can be easily accomplished by checking the queue that associated with the root of a subtree.

We perform a case study to show the capability of Northup for load balancing. We exploit data parallelism of HotSpot-2D and spread work simultaneously on both CPU and GPU at the leaf node (a shared-virtual memory APU). We use the SSD as storage. Because data cannot fit into the main memory, for an inner node each queue element represents a task of moving a chunk from SSD to the main memory. At a leaf node, we set up multiple queues to track computation progress; each queue is associated with a GPU OpenCL workgroup or a CPU thread (see Figure 10). When a data chunk reaches the main memory, it is broken into smaller blocks. The task of each row of blocks is assigned to one queue. Thus each queue is assigned with multiple rows of blocks. Based on this organization, we enable work stealing across the CPU and the GPU. A CPU thread or a GPU workgroup pops elements from its local queue (at its tail pointer) for computation. GPU workgroups may process tasks faster than CPU threads, so GPU workgroup may steal elements pointed by the head pointer of another CPU queue. Atomics with the platform-scope and acquire memory ordering [2] are used to implement the lock-free stealing [24].

Figure 11 shows the benefit of CPU-GPU load balancing against GPU-only execution. With work-stealing, stencil achieves a performance improvement up to 24% against the GPU-only Northup. We experiment three configurations (m,n) where m represents the dimension of the square input matrix in SSD and n represents the dimension of the chunk that is loaded to the main memory. Each workgroup processes a row of blocks with dim $16 \times n$ block. The parameter n has to be big enough so there are enough elements per queue while small enough to fit into the main memory. We vary the number of queues and find that using 32 queues achieves the best performance, since multiple workgroups per GPU SIMD engine is needed to fully utilize GPU hardware and hide latency. In summary, Northup structures can facilitate scheduling and load balancing of applications in a heterogeneous system.

VI. Discussions

High-level Programming Language: Northup can be an intermediate layer interacting with a higher level abstraction. A high-level language and API (e.g., OpenMP tasks, Cilk) can be extended with pragmas, so the compiler can emit code with calls to low-level libraries and transform codes to a recursive format to exploit the memory hierarchy with explicit data transfers and accelerated computation. Either programming in a high-level language or in Northup, the system must be able to identify both the underlying memory topology and properties of memory nodes, which makes the proposed tree structure useful in practice.

Northup for HPC: In traditional HPC, high-speed networks (e.g., InfiniBand) are used to stream data to a local compute node. Thus NVMs (e.g., SSDs) are usually treated as a general-purpose caching layer or burst buffer between compute nodes and storages. However, this may only be efficient for a subset of workloads with a high-degree of reuse. With new NVMs, bandwidth of these devices is already beginning to eclipse available point-to-point network bandwidth [25], which makes implementation of NVM as a form of per-node slower memory promising. A future Exascale compute node may use die-stacked memory as a small capacity, fast memory while using NVM as large capacity, slow memory [1]. In addition, PIM can be naturally supported as a Northup subtree. In fact, Northup can incorporate any subsystem with its own memory hierarchy. Therefore, Northup-like solutions can be useful to manage such deep, complex memory hierarchy in a future system, while interacting with other software for distributed memory management and computation–this interdisciplinary area is insufficiently studied by prior work.

Memory Space Northup assumes several logical memory or storage spaces to operate (though not a requirement). Programmers prefer a simple abstraction of single, flat address space. Ideally the underlying software (e.g., OS) or hardware can intelligently detect and determine desirable mappings of data to different physical memories. However, it is unclear if an effective solution exists for increasingly complex memory hierarchy. Though the best approach to abstract memories remains an open question, the notion of differentiating spaces and regions (e.g., memory scopes) is useful for programmers in many cases. Northup is especially suitable for these cases.

Data Layout: Different architectures may favor different memory layouts and access patterns (e.g., row versus col-major, AoS versus SoA) [26]. For sparse-matrix problems, the choice of data layouts not only depends on architectures but also on inputs [27]. Furthermore, some applications may switch access patterns in different execution phases [28]. One can imagine when data migrates across memory levels, chunks can be transformed and stored in different formats. In fact, layout transformation is beneficial for applications with sufficient data reuse [29] and exposing layout control to the programming API might also be useful. Northup can be easily extended to support this with a special version of *move_data*() function.

VII. Conclusion And Future Work

Northup incorporates the fundamental divide-and-conquer technique in computer science with the goal of making programming a heterogeneous system easier. A heterogeneous, asymmetric tree-based model and associated structures are designed to abstract diverse hardware components including new memory/storage devices and accelerators. Programs developed in a Northup style are portable and adaptable to system architectural changes. Furthermore, it allows efficient, recursive mapping of a problem to fully exploit the resources of the heterogeneous system. Northup is further enhanced by a unified interface for managing data placement and movement, and it can easily integrate runtime techniques to improve performance (e.g., queue-based dynamic load balancing). We use several state-of-the-art parallel algorithms and demonstrate the performance benefits of Northup. With emerging memory and storage devices, more and more tree levels and nodes will be added to the memory hierarchy of the computer systems. Northup is an effective model for managing the heterogeneous memory hierarchy. Also, our solution decouples memory management and computation; and computation can be a standalone "plug in" to the program regardless of which acceleration approach to use (FPGA, GPU, and other many-core processors). Future work includes extending the model to support distributed systems. Some part of Northup is currently optimized in a manual way, we will include better compiler support in future.

Acknowledgment

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.

References

[1] M. J. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, and G. Rodgers, "Achieving exascale capabilities through heterogeneous computing," *IEEE Micro*, vol. 35, no. 4, pp. 26–36, 2015.

[2] "Heterogenous System Architecture," http://hsafoundation.com/.

[3] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *HPCA*, 2015.

[4] G. Loh and M. D. Hill, "Supporting very large dram caches with compound-access scheduling and missmap," *IEEE Micro*, vol. 32, no. 3, pp. 70–78, 2012.

[5] Y. Xie, "Modeling, architecture, and applications for emerging memory technologies," *IEEE Design Test of Computers*, vol. 28, no. 1, pp. 44–51, 2011.

[6] "High Bandwidth Memory," http://www.amd.com/en-us/innovations/software-technologies/hbm/.

[7] P. Wu, D. Li, Z. Chen, J. S. Vetter, and S. Mittal, "Algorithm-directed data placement in explicitly managed non-volatile memory," in *HPDC*, 2016.

[8] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *ASPLOS*, 2011.

[9] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ASPLOS*, 2011.

[10] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance," in *USENIX ATC*, 2008.

[11] "Storage Networking Industry Association. NVM programming model," Tech. Rep., 2013.

[12] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "Gpufs: Integrating a file system with gpus," in *ASPLOS*, 2013.

[13] B. Andersen, F. Gustavson, A. Karaivanov, J. Wasniewski, and P. Yalamov, "Lawra linear algebra with recursive algorithms," in *International Workshop on Applied Parallel Computing*. Springer, 2000.

[14] J. J. Dongarra and P. Raghavany, "A new recursive implementation of sparse cholesky factorization," in *16th IMACS World Congress*, 2000.

[15] Q. Yi, V. Adve, and K. Kennedy, "Transforming loops to recursion for multi-level memory hierarchies," in *PLDI*, 2000.

[16] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *ICS*, 2006.

[17] "HSA Matrix Multiplication Example," Web resource. https://github.com/HSAFoundation/CLOC/tree/master/examples/snack.

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.

[19] "SPEC ACCEL," Web resource. https://www.spec.org/accel/.

[20] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *SC*, 2014.

[21] "CL offline compiler and SNACK," Web resource. https://github.com/HSAFoundation/CLOC/.

[22] "Radeon Open Compute. Rocm," Tech. Rep.

[23] "The university of florida sparse matrix collection," Web resource. http://www.cise.ufl.edu/research/sparse/matrices/.

[24] P. Tsigas and D. Cedermann, *Dynamic Load Balancing Using Work-Stealing*. GPU Computing Gems Jade Edition, 2011.

[25] M. Jung, E. H. Wilson, III, W. Choi, J. Shalf, H. M. Aktulga, C. Yang, E. Saule, U. V. Catalyurek, and M. Kandemir, "Exploring the future of out-of-core computing with compute-local non-volatile memory," in *SC*, 2013.

[26] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in *PACT*, 2010.

[27] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," *NVIDIA Technical Report NVR-2008-004*, 2008.

[28] Z. Majo and T. R. Gross, "Matching memory access patterns and data placement for numa systems," in *CGO*, 2012.

[29] B. Akin, F. Franchetti, and J. C. Hoe, "Data reorganization in memory using 3d-stacked dram," in *ISCA*, 2015.