Progressive codesign of an architecture and compiler using a proxy application

Arpith Jacob, Ravi Nair, Tong Chen, Zehra Sura, Changhoan Kim, Carlo Bertolli, Samuel Antao, and Kevin O'Brien
IBM T.J. Watson Research Center
1101 Kitchawan Rd., Yorktown Heights, NY, USA.
{acjacob,nair,chentong,zsura,kimchang,cbertol,sfantao,caohmin}@us.ibm.com

***Abstract*—The Active Memory Cube (AMC) is a novel near-memory processor that exploits high memory bandwidth and low latency close to DRAM to execute scientific applications in an energy-efficient manner. Its energy efficiency is derived from a combination of its novel scalar-vector data-flow path combined with its simple control-flow path that required the development of a sophisticated compiler, co-designed with the architecture. Such co-design is commonly done using hand-tuned codes for simple kernels that typically do not capture the nuances of real-world applications or reveal the complexities of programming a heterogeneous system. At the same time, an entire application is intractable to an early-stage compiler.**

In this work we describe a progressive, iterative methodology to the co-design of the compiler and architecture for the AMC using LULESH, a real-world hydrodynamics proxy application. We focus on a procedure that calculates the kinematic variables for domain elements. During the concept phase we looked at simpler kernels directly derived from the procedure, and progressively moved to the entire procedure as the compiler and simulation environment matured. We found this progression from the simpler extracted kernels to the entire procedure useful in gradually exposing new issues in the compiler and architecture. Directly applying optimizations developed for the simpler kernels resulted in poor performance on the proxy application, but after developing new compiler passes, the former optimizations could be applied profitably. Co-design on a proxy application revealed opportunities to refine the micro architecture that were not identified with simple micro benchmarks alone. Development of future accelerators and their programming environments can benefit from a similar iterative co-design through component kernels to entire proxy applications.

I. Introduction

Slower growth in processor performance has led to the consideration of alternate compute architectures, in particular, power-efficient accelerators. Processing in Memory (PIM) devices integrate logic elements near memory cells to provide very high memory bandwidth at low latency [1]. Recent advances in 3D integration have been exploited to attach a logic layer below DRAM layers via Thru Silicon Vias [2], [3], [4], revitalizing interest in this approach.

The Active Memory Cube (AMC) [5] is a recently developed PIM architecture for Exascale computing that uses vector cores directly attached near memory to exploit the large bandwidth and low latency without the energy inefficient structures of traditional complex out-of-order cores. It relies on a powerful compiler to parallelize and optimize programs for high efficiency, also offloading the management of some hardware resources such as the instruction cache.

A realization that Exascale machines will likely use such new architectures and programming models has spurred the demand for collaborative development between scientists and system designers. Proxy applications from existing high-impact scientific codes are identified and used to iteratively guide the design of new architectures, programming models, compilers, and algorithms [6]. Unfortunately, this presents a challenge as during the design phase of a project like the AMC, a stable compiler or even the simulation environment is minimally functional, if even present. In this work we describe a progressive, iterative methodology to the co-design of a compiler and architecture using one such proxy application, Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) [7], [8].

Early on in the project we focused on two simpler kernels derived from the application—in order of increasing complexity, Determinant and HexahedronVolume—relying on hand-tuned assembly code to guide development. We used an iterative approach to develop new compiler optimizations, with Determinant primarily guiding transformations to increase compute efficiency and HexahedronVolume stressing the memory subsystem. As we progressed from the simple kernels to the entire procedure we found new issues in the compiler and architecture that had to be tackled to achieve high performance. A few optimizations were directly applicable to the proxy application, several partially so, and still others not directly applicable. We had to develop new compiler passes for the complex kernels that were then able to unlock opportunities that could be exploited by the earlier developed optimizations. Some of these differences also led us to change the parameters of the architecture.

Our paper makes the following contributions.

1) We describe the co-design of a compiler and accelerator via a progressive, iterative methodology through extracted component kernels to an entire proxy application that makes it feasible to guide the development of the architecture and compiler optimizations. This approach is instructive for designers of future accelerator systems.
2) Through a detailed case study we explore how application developers can map multiple levels of parallelism onto a near-memory processor. This is generally useful as LULESH exhibits characteristics found in a variety of Exascale applications [8].
3) We study the individual benefits of various compiler optimizations on compute- and memory-bound, as

well as direct and indirect memory reference kernels in the context of processing near memory.

The rest of this paper is organized as follows. We first briefly describe the AMC in Section II. We introduce our approach to codesign in Section III with related work in Section IV. We then describe LULESH in Section V and apply our approach to it in Section VI. Section VII concludes with compiler and architecture refinements learnt through our codesign methodology.

II. The Active Memory Cube

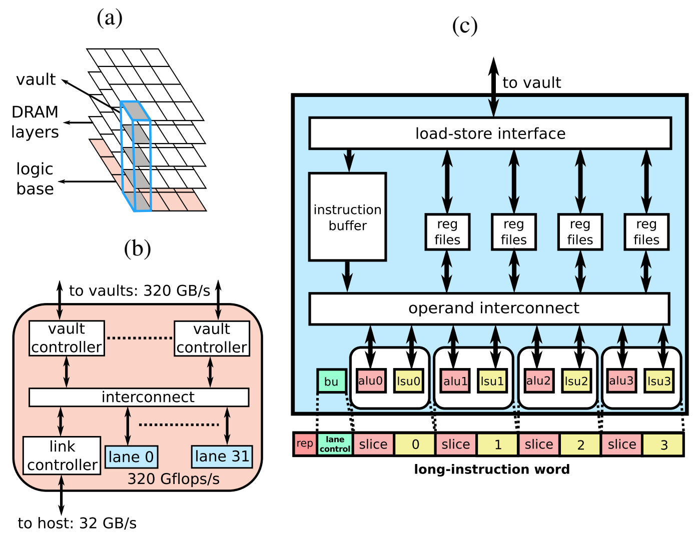


Fig. 1: The Active Memory Cube: (a) Logic and memory layers (16 lanes and vaults depicted), (b) Logic base, (c) Lane.

The AMC [5] is a proposed near memory processor that stacks 8 GB of DRAM layers on a logic base of 32 *processing lanes* (see Figure 1a). A cross section of stacked memory elements on top of one processing lane is termed a *vault* and a group of eight neighboring vaults form a *quadrant*. This organization is important, as although any processing lane may access any vault through an interconnect (Figure 1b), the latency of access is lower if lanes access vaults within their local quadrant. Operating system support facilitates data allocation within a particular quadrant [14].

An example compute node in a system could hold a multi-core host processor and up to 16 AMC chips. Each AMC also serves as an external memory chip and memory is kept coherent with the host's data caches. This shared memory architecture greatly simplifies and improves efficiency of communication between the two compute chips.

A. Processing Lane

As illustrated in Figure 1c a lane is a vector processor consisting of four *slices* that execute in lock step. Each slice contains an arithmetic unit, a load/store unit, and associated register files. A slice can serially perform the same operation on up to 32 elements. The arithmetic unit has an integer and a double-precision floating-point pipeline. The load/store unit accepts vector memory operations with simple strided as well as more powerful scatter/gather accesses (i.e., to 32 random addresses).

Each slice has sixteen 32-element vector registers, four 32-element mask registers, and thirty-two scalar registers. A slice's vector registers may be read by the other three slices through shared register file ports. A lane instruction is coded in long-instruction form (VLIW) with sub-instructions designated for the four slices in the lane as well as the branch unit. Each lane stores up to 512 long-instructions in a lane instruction buffer (LIB).

The functional units on a lane fully expose their pipelines to the compiler; for correct execution, sufficient delays must be inserted in the instruction stream between producer and consumer instructions. As the memory access time is unpredictable the hardware provides an interlock that stalls the entire lane when a use of an as yet unavailable memory value occurs.

We have developed a detailed cycle accurate simulator for an AMC node that simulates both the application on the host and the accelerated section on the AMC. We refer the reader to our previous publication for details [5].

III. Overview of Progressive Codesign

Proxy applications are concise representations of a production application that are easier to compile and execute on existing platforms. But where the hardware itself is being designed along with the compiler, the latter is presented with a wide range of optimization choices. Focusing on a proxy application makes it difficult to identify important optimizations and disentangle interactions between them. For example, in an accelerator environment, high-level loop optimizations, scheduling, and register allocation are all intricately linked. It is more feasible to effectively explore a large design space by focusing on kernels even smaller than the proxy application.

Our novel approach is to select simplified kernels *explicitly from* a proxy application to run on a barebones simulator and to iteratively improve the compiler and architecture. We progress from the component kernels to the entire proxy application as the compiler matures but this final step is normally too late to incorporate user feedback. Because the simpler component kernels have characteristics representative of the proxy application, it enables scientists to affect the architecture and compiler optimizations at an early stage.

Our method of application simplification helps first validate individual optimizations, and progression through complex kernels helps tune heuristics as interaction increases.

IV. Related Work

Most PIM studies in the past have focused on the performance of micro benchmarks to characterize the hardware [2], [3], [4]. While instructive to an architect, a study of a real-world proxy application is of more use to an application developer since it contains customizations important to a domain expert, and exposes issues to do with the high-level language constructs for an accelerator, mapping of parallelism, control of resource usage, and code-size. In our study we first characterize performance using simple kernels extracted from LULESH, but then build upon this work to accelerate this procedure. Previous efforts have accelerated LULESH on the Xeon Phi [9], Blue Gene/Q [8], and GPUs [10].

Sottile et al. [11] have developed a semi-automatic technique to extract skeletons from an application to isolate its memory access and compute patterns, I/O, and network communication. Their work can automate the currently manual step of extracting the component kernels of proxy applications.

Dwarfs [12] have been proposed as an alternative to benchmarks. A Dwarf is a class of algorithms that exhibits a common pattern of computation and communication. The goal is to use a few styles of computing to answer general questions on programming abstractions and parallel application requirements, not to characterize one application or its specific optimizations on a single platform. Our goal instead is to guide and evaluate a specific architecture and its compiler, for which a Dwarf is too abstract.

Shalf et al. [6] apply the co-design methodology to climate modeling and seismic imaging for Exascale design exploration. Our work describes the co-design of a novel hardware architecture and its compiler in the domain of hydrodynamics. Mohiyuddin et al. [13] use an automatic tuner in the co-design of a Tensilica system but they only explore parameters such as the number of cores, cache capacities, and off-chip bandwidth.

V. LULESH

Hydrodynamics codes like LULESH are an important Exascale workload. LULESH models the motion of materials when subject to external forces by partitioning a 3D space into volumes using an unstructured mesh. As shown in Figure 2, an *element* is defined at the center of each hexahedron and a *node* at points where mesh lines intersect. Elements and nodes are associated with thermodynamic and kinematic variables respectively. The algorithm starts with a point energy source and progressively evolves the solution of the kinematic and thermodynamic variables through discrete time increments.

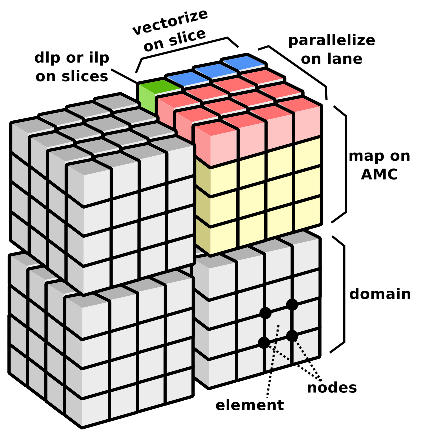
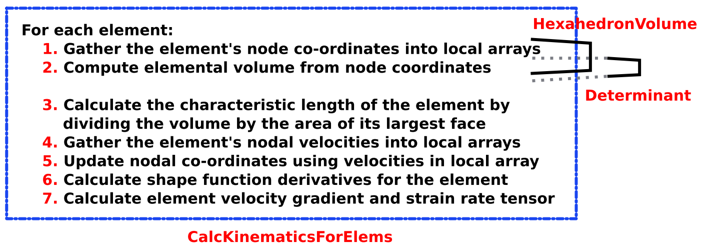


Fig. 2: An illustration of the LULESH problem and a parallelism mapping onto the AMC. As the simulation progresses the grid follows the motion of the material elements so an irregular mesh representation is necessary.

The LULESH application groups a rectangular set of adjacent elements into *domains*. Elements within a domain are stored in a single data structure and may be conveniently located in a single data locale. A simulation *problem* consists of a number of adjacent domains in 3D space. Lagrangian simulation exhibits concurrency that can be weak scaled to a large number of domains.

A. Description of the LULESH Kernels

In this work we focus on the kinematics calculation, which exhibits interaction between an element and its neighboring nodes. The key characteristic is that accessing an element's nodal variables requires indirect addressing through the unstructured mesh, which stresses the memory subsystem.



For each element:
1. **Gather the element's node co-ordinates into local arrays**
2. **Compute elemental volume from node coordinates**
3. **Calculate the characteristic length of the element by dividing the volume by the area of its largest face**
4. **Gather the element's nodal velocities into local arrays**
5. **Update nodal co-ordinates using velocities in local array**
6. **Calculate shape function derivatives for the element**
7. **Calculate element velocity gradient and strain rate tensor**

Fig. 3: The CalcKinematicsForElems kernel computes variables for each element in the domain. HexahedronVolume isolates the volume calculation along with scatter/gather operations that test the memory subsystem. Determinant also computes the volume but reads nodal variables from a unit-strided data structure.

We have outlined this procedure, termed *CalcKinematicsForElems*, in Figure 3. Since it is relatively large, the authors of LULESH have also isolated the gather of an element's nodes and its volume calculation (Steps 1 and 2 in Figure 3) into a smaller kernel we call *HexahedronVolume* (kernel1 in [7], Section 1.6.1.1). Finally *Determinant* performs only the determinant calculation in Step 2, assuming a contiguous ordering of nodal variables in memory. This simulates the scenario when a prior data aggregation step is employed.

1) CalcKinematicsForElems: Each iteration of this loop is outlined in Figure 3. First, an element's eight nodal co-ordinates are gathered into a local array. These variables are then referenced throughout the computations in the subsequent subroutines. Next, the volume of the hexahedron and the area of its faces are calculated. The nodal velocities along the three dimensions are gathered into a local array and used to compute the new co-ordinates of the nodes. These co-ordinates are again used to calculate the shape function derivatives, which along with the velocity variables, are used to calculate the velocity gradient. Finally, the velocity gradient is scattered back to the domain data structure.

This is a relatively complex, compute-bound procedure but it also induces gather and scatter operations that test the memory subsystem due to its random access pattern. The large number of nodal variables that are live across several steps of the computation in Figure 3 also increase register pressure.

2) Determinant: To better characterize the challenges and tradeoffs of the various design parameters of an accelerator platform, Determinant isolates the element's volume calculation in Step 2 of Figure 3. It is summarized in Figure 4 and evaluates the determinant of three matrices obtained from the nodal variables. We assume that these nodal co-ordinates are

```
Real_t CalcElemVolume(Real_t x0, ..., Real_t x7, Real_t y0,
                      ..., Real_t y7, Real_t z0, ..., Real_t z7) {
  // Calculate elements of the three matrices
  Real_t dx61 = x6 - x1;                      Real_t dx25 = x2 - x5;
  Real_t dy61 = y6 - y1;      ..........      Real_t dy25 = y2 - y5;
  Real_t dz61 = z6 - z1;                      Real_t dz25 = z2 - z5;

  Real_t volume =
    TRIPLE_PRODUCT(dx31 + dx72, dx63, dx20,
      dy31 + dy72, dy63, dy20,          // Matrix A
      dz31 + dz72, dz63, dz20) +
    TRIPLE_PRODUCT(dx43 + dx57, dx64, dx70,
      dy43 + dy57, dy64, dy70,          // Matrix B
      dz43 + dz57, dz64, dz70) +
    TRIPLE_PRODUCT(dx14 + dx25, dx61, dx50,
      dy14 + dy25, dy61, dy50,          // Matrix C
      dz14 + dz25, dz61, dz50);
}
```

Fig. 4: The elemental volume computation evaluates the determinant of three matrices obtained from node co-ordinates.

gathered in a pre-processing step into contiguous memory so that they can be effectively accessed by the memory subsystem of most accelerator platforms. Being smaller in size, this kernel enables a more aggressive exploration of optimizations to increase compute efficiency that we can learn from and apply to the entire procedure.

Note that in addition to instruction-level parallelism, there is data-level parallelism in the loop body that can be exploited by three threads (calls to **TRIPLE_PRODUCT** in Figure 4). Each of these threads can independently calculate the determinant of one of the three matrices in the loop. Serialization only occurs when the results are summed into the *volume* variable. Similar data-level parallelism is available throughout the larger CalcKinematicsForElems procedure. An interesting question we explore is if it is beneficial to exploit data- or instruction-level parallelism from the loop body.

3) HexahedronVolume: To test the memory subsystem, HexahedronVolume incorporates the gather operations in Step 1 of Figure 3 in addition to the volume computation. This code helps study optimizations that improve memory efficiency. Figure 5 shows the code that gathers the co-ordinates of the eight nodes that surround an element.

```
for( Index_t k=0 ; k<numTotalElems ; ++k ) {
  Real_t  x_local[8],  y_local[8],  z_local[8];
  const Index_t*  const elemToNodes = elemsToNodesConnectivity[k] ;
  // Gather nodal co-ordinates from global arrays and copy into local arrays
  for( Index_t lnode=0 ; lnode<8 ; ++lnode ) {
    Index_t gnode = elemToNodes[lnode];
    x_local[lnode] = x[gnode];
    y_local[lnode] = y[gnode];
    z_local[lnode] = z[gnode];
  }
  ...
}
```

Fig. 5: Gathering position variables of an element's neighbors.

VI. Exploiting Parallelism in LULESH

A domain in the proxy application contains 45^3 elements, and depending on the resolution of the problem space, realistic hydrodynamics codes simulate millions of domains. In a single time step, the work represented by each element is an independent unit that can proceed in parallel without synchronization. Therefore, LULESH contains ample opportunity to exploit parallelism both within the application code and the problem data. The application utilizes MPI to map domains across nodes and we added OpenMP 4.0 directives to offload and parallelize loops.

LULESH is an ideal vehicle for exploring the design space of an accelerator. Our mapping of parallelism in the application to an AMC's compute elements is illustrated in Figure 2. An AMC with 8 GB easily holds the data associated with a single domain so a natural way to use the cubes is to map one MPI rank per AMC. Alternatively, it is easy to support more than one MPI rank in each AMC. However, it is difficult to support the case of one MPI rank spanning multiple AMCs due to the non uniform cost of inter-cube communication.

Elements of a domain (the loop in Figure 3) are partitioned and parallelized across 32 lanes. We then vectorize each partition of elements on a lane. We can exploit the available data parallelism by using the four slices of a lane to each operate on 32 distinct elements of a partition (128 elements are processed concurrently). However, an element requires 24 position and velocity variables, which exceeds an individual slice's vector register file capacity. Moreover, exploiting additional data parallelism increases the demand on the memory subsystem. Instead, we take advantage of the flexibility of the lane micro architecture to co-operatively process 32 elements at a time on the four slices.

Within the loop body itself there is meaningful amount of computation per element that is instruction-parallel. Furthermore, as mentioned in Section V-A2, there is also data parallelism when the determinant calculation is applied to three matrices. We explore these two options in the next section.

A. *Determinant*

We use Determinant as a vehicle to explore the parallelism mapping of the kernel's loop body on a lane.

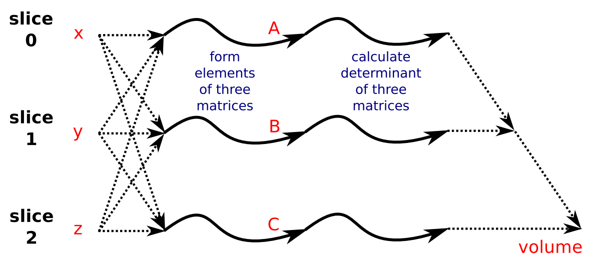

slice 0
slice 1
slice 2
x
y
z
A
B
C
form elements of three matrices
calculate determinant of three matrices
volume

Fig. 6: Data-level parallelism for three threads in the elemental volume computation of Figure 4.

1) Data Parallel Algorithm: As shown in Figure 6, the volume computation of an element can be decomposed into three independent threads that each forms a matrix and evaluates its determinant. We can exploit this data parallelism by mapping these threads of computation on three distinct slices. This approach optimizes for data locality as operations are executed on the slice that holds its operands. It is also easier for a programmer to understand and implement. However, it utilizes only three of the four slices.

This is conceptually similar to Single Instruction Multiple Thread (SIMT) processing—where a slice is equivalent to a SIMT thread—since each slice calculates its own determinant, except that any slice is allowed to read registers from any other slice (in pure SIMT the x, y, and z variables of Figure 6 are replicated on each SIMT thread). At the end, we gather the three determinants and calculate the volume on a single slice, similar to the reduction operation after a join in a multi-threaded implementation.

2) Instruction Parallel Algorithm: In an alternative implementation we perform the volume calculation in Figure 4 on 32 elements and exploit instruction parallelism. We have freedom to load the co-ordinates and co-operatively perform the calculation on any of the four slices. This approach optimizes for parallelism at the expense of locality, but may encounter resource contention as the vector register file ports are shared across functional units. It is also challenging to manage by hand but is easy for a compiler to implement.

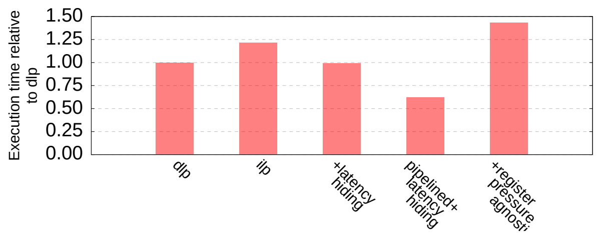


Fig. 7: Comparison of hand-written data parallel algorithm vs. compiler generated instruction parallel algorithm.

Figure 7 shows execution time relative to a hand-optimized version of the data parallel algorithm. The compiler generated instruction parallel algorithm takes 22% more cycles to execute despite using all four slices. Close to 51% of execution time is spent waiting on memory compared to 32% for the hand-optimized code—a consequence of aggressively exploiting parallelism. Stalling is due to the direct path from registers to DRAM: if a load is followed too closely by its consumer, a hardware interlock pauses all slices until the requested value is available. An advantage of the the data parallel algorithm is that it is easier for the programmer to hide this latency by scheduling independent instructions during this interval.

In response to this observation we modified the scheduler to separate the load from its uses by a user-defined latency. Non-dependent instructions are scheduled in the intervening period to keep the slices utilized. Without latency hiding, the scheduling heuristic places loads and their first use close together, with the majority separated by a distance of only 32 cycles (the vector length). The third bar in Figure 7 shows equivalent performance to the hand-written code when 160 cycles are hidden, where stalling falls to just 3.2% of execution time. Nevertheless, this approach has its limitations. In this example, there simply aren't enough instructions to keep the slices fully utilized.

3) Software Pipelining to Exploit Inter-iteration Parallelism: We can effectively hide load latency and increase slice utilization by exploiting another dimension of parallelism, inter-iteration pipeline parallelism through modulo scheduling [15], [16]. Briefly, consecutive vector iterations executing on a lane (each operating on 32 elements) are initiated at constant intervals. By partially overlapping two or more consecutive iterations we can exploit some of the available data parallelism without the register requirements of a fully data parallel algorithm. As shown in the fourth bar of Figure 7, pipelining two vector iterations while hiding 160 cycles improves slice utilization, achieving a 61% speedup compared to the original hand-optimized code.

The final bar in Figure 7 shows a dramatic drop in performance with an aggressive scheduling heuristic that does balance register usage across slices but does not aggressively avoid spilling. We observed this slowdown with only three vector register spills in the loop. Therefore our compiler as far as possible selects a schedule that avoids register spilling, even at the cost of decreased slice utilization.

In summary, we can draw a few conclusions that can be applied to optimize subsequent kernels. Compared to a fully data-parallel approach, software pipelining can exploit instruction parallelism and some amount of data parallelism to more effectively increase slice utilization without overloading the memory subsystem. Hiding latency to memory is critical for performance; indeed, the observed latency will only increase due to the random access pattern of the subsequent kernels. Finally, unlike traditional general-purpose architectures, even a few register spills dramatically decrease performance.

4) Scaling Across Lanes: Next we studied weak scaling of Determinant by varying the number of active lanes and measuring flop efficiency. Efficiency is measured as the number of floating-point operations retired per cycle (fused operations count as two) relative to the floating-point capacity (four units per lane, each capable of executing a fused operation).

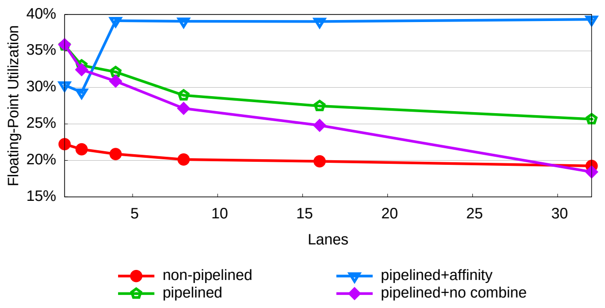


Fig. 8: Performance scaling of Determinant with increasing number of lanes.

Figure 8 shows the results of our experiments. The first two curves show the *non-pipelined* (labeled *ilp+latency hiding* in Figure 7) and *pipelined* (labeled *pipelined+latency hiding* in Figure 7) cases. We see that pipelining always achieves superior performance but both degrade significantly as more lanes are activated. Indeed, while pipelining is 61% faster than the non-pipelined case with one lane, the advantage is only 33% when all lanes of the AMC are active. As more lanes are activated, congestion on the memory crossbar, and consequently, load latency increases.

As mentioned in Section II, vaults are organized into quadrants, and bandwidth across quadrants is lower than within. Fortunately, we can reduce congestion by exploiting data affinity. We use directives to organize data within quadrants such that every lane only accesses one of the eight vaults within its quadrant. Figure 8 (*pipelined+affinity*) shows perfect scaling with this approach.

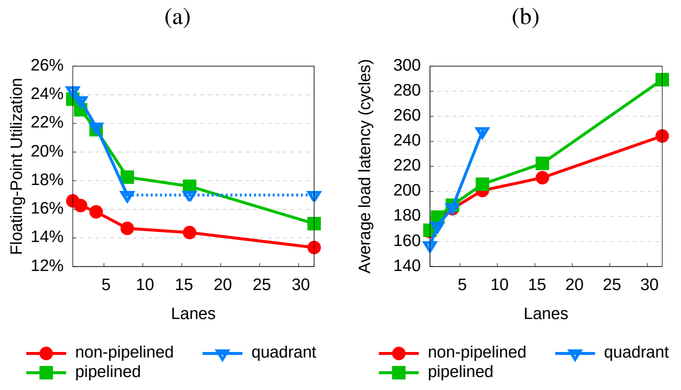


Fig. 9: Performance and access latency variation for HexahedronVolume with increasing number of lanes.

Finally, we note that since Determinant uses sequential stride-one memory operations, the lane hardware is able to reduce traffic on the crossbar by combining four load or store requests into just one on the crossbar. The curve labeled pipelined+no combine shows performance with the load- and store-combining feature disabled (without exploiting affinity). As $4\times$ the number of memory requests are generated, performance drops by 49% with 32 lanes compared to a single lane and 28% compared to the case where combining is enabled.

The scaling experiments enable us to draw insights applicable to the ensuing kernels. Since they employ gather and scatter operations those kernels cannot exploit the combining feature nor restrict accesses to a lane's quadrant. We expect the increased network traffic to dramatically increase memory access latency, unless further optimizations are employed.

B. *HexahedronVolume*

Our compiler completely unrolls the inner loop shown in Figure 5 that gathers nodal co-ordinates. We also modified the inlining heuristic to inline all function calls within the accelerated region. The primary challenge of HexahedronVolume is in handling the random access pattern of the gather operations.

1) Scaling Across Lanes: Using our experience from the previous kernel, we apply software pipelining to exploit instruction-level parallelism and inter-iteration parallelism while hiding memory latency. Figure 9a reports performance scaling as the number of active lanes are increased. Pipelining always pays off compared to a simple list-scheduling technique, however, there is a 66% performance differential in the best run of this kernel compared to Determinant.

This drop in performance is explained by a $2.47\times$ increase in load latency observed at the lanes. A total of 98.3% of the memory accesses are 8-byte requests that are not combined to more efficient 32-byte ones. This is due to the inherent nature of the mesh accesses where random accesses dominate. Additionally, only 25% of references are to a lane's local quadrant while the remaining are to remote quadrants. This is reflected in Figure 9b where observed latency increases 71% as all lanes in the AMC are activated.

We can avoid the cross-quadrant link-bottleneck by exploiting the nature of the problem in LULESH. Recall from Figure 2 that adjacent elements are grouped into a domain that is processed by one MPI rank per AMC. If instead we map four MPI ranks to an AMC, one per quadrant, such that all data associated with a domain is stored entirely within the eight vaults of a quadrant, we can avoid inter-quadrant traffic during computation. The curve labeled *quadrant* in Figure 9a shows performance using this approach. With all eight lanes of a quadrant activated, performance is slightly lower than when the bandwidth of all 32 vaults of the AMC are available. However, this efficiency is expected to remain constant as the remaining three quadrants (MPI ranks) are activated.

To summarize, we see a significant performance drop when the memory access pattern changes from sequential to random access. This is attributed to the low utilization of memory bandwidth due to 8-byte random accesses. A lesser contributing factor is the bottleneck at the inter-quadrant links. This problem is overcome in LULESH by a quadrant-local mapping of domains.

Finally, when there are indirect accesses, we may try resolving the indirection by gathering data in advance. In this example, we may gather the nodal data into a new data structure before the computation loop of HexahedronVolume. Obviously this requires temporary memory space but may be beneficial if the application contains several kernels requiring indirection through the mesh.

C. *CalcKinematicsForElems*

In this section we first describe the compiler challenges in accelerating this procedure before discussing performance. Many of these issues occur in other real-world applications and so are broadly applicable.

Handling increased code size. An obvious way to accelerate this procedure is to fully unroll inner loops and inline all function calls. However, the resulting code may be too large to fit within an accelerator's instruction buffer. Moreover, optimizations such as software pipelining further increases code size. To support arbitrarily large programs, we modified our compiler to implement a software managed instruction cache (I-Cache) that partitions a program into overlays that are loaded on demand into the LIB. We choose to unroll inner loops and inline all functions and rely on this I-Cache functionality. In what follows, we investigate the benefit of optimizations given the increase in code size.

Alias analysis for the Standard Template Library. LULESH uses the STL vector class to store node co-ordinates. The generic template allows a user-defined memory allocator, because of which our compiler is unable to assert the *no alias* relationship. To circumvent this problem, STL vector was replaced by the STL valarray class.

1) Accelerating the Inlined Procedure: Our first attempt compiles the entire kernel as a single loop on a lane. We experimented with a non-pipelined schedule that hides a memory latency of 160 cycles. The first pass of scheduling creates a schedule that has an expected flop efficiency of 24.62% assuming all loads are satisfied within 160 cycles. However, after register allocation there are 89 vector register spills and with all lanes active, the observed flop efficiency is just 4.6%.

The scheduler uses two heuristics to control register pressure that generally work well with the kernels we have tested.

The order in which instructions are considered for scheduling affects exploited parallelism and register use. Our preferred one, Swing Ordering [15], strikes a good balance between the two. Second, our scheduler tracks the live ranges of variables to never overflow register file capacity, reducing slice utilization if necessary. In the case of CalcKinematicsForElems, however, these alone are insufficient and we must turn to higher level transformations to drastically cut register pressure.

2) Loop Distribution to Minimize Register Use: Loop fission is an optimization where a single loop is broken into multiple loops over the same iteration space. By decreasing the maximum live values in a loop, fission can decrease register pressure. The disadvantage is that if the same data is referenced by one or more of the multiple loops, additional references to memory is required.

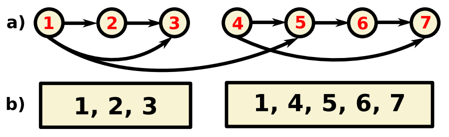


Fig. 10: (a) Data flow between steps of the CalcKinematicsForElems kernel and (b) the result of loop distribution to reduce register pressure.

Fortunately the CalcKinematicsForElems kernel in Figure 3 lends itself to this optimization. We manually distributed this loop into two smaller ones such that redundant data movement is minimized. Figure 10a shows the data flow between the steps of the algorithm and Figure 10b, our preferred partition, where only the nodal co-ordinates are loaded twice.

To study the behavior of CalcKinematicsForElems we compiled and executed the first loop after distribution on the AMC (the second loop is similar to the first). Here we assume that the residue iterations are executed on the host. Pipelining overlaps two vector iterations, initiating a new one every $2,482$ cycles (compared to $3,636$ otherwise), so we expect it to perform better. While the total number of vector registers that are spilled to memory are roughly the same in both cases (16), the pipelined schedule only spills half the number within the loop. Compared to the original single loop of CalcKinematicsForElems, the loop after distribution achieves a flop efficiency of 20.2% with 32 lanes, a $4.4\times$ improvement in performance. This is due to the lower memory bandwidth pressure because of the order-of-magnitude fewer spills.

3) Scaling Across Lanes: Next, we studied the strong scaling behavior of this kernel and report results in Figure 11a. The graph confirms the superiority of software pipelining. Note that flop-efficiency is better than HexahedronVolume due to a higher arithmetic intensity. While there are more memory references in CalcKinematicsForElems the additional ones address memory sequentially. This is further confirmed in Figure 11b where observed load latency is consistently 40-55 cycles lower than for HexahedronVolume.

As in the previous kernel, we can avoid the cross-quadrant link-bottleneck if we map four LULESH domains to an AMC. Figure 11a reports 24.3% flop utilization with 8 lanes which we expect to remain constant as all lanes are activated. This is

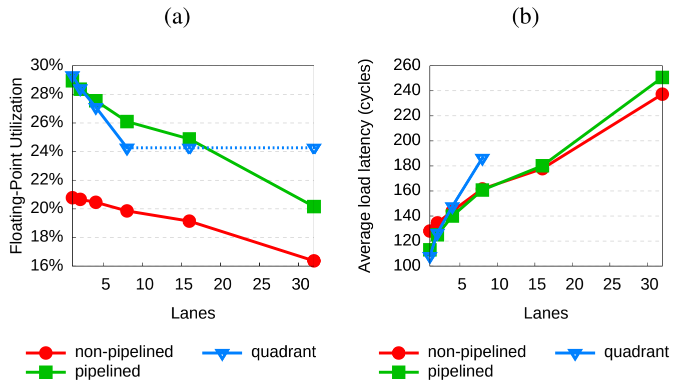


Fig. 11: Performance and access latency of the first loop of CalcKinematicsForElems after loop distribution with increasing number of lanes. Results for the second loop are similar.

TABLE I: List of optimizations considered for LULESH.

	Det	HexV	CalcKFE
Vectorization	✓	✓	✓
DLP - Slice	✓	-	-
DLP - Lane	✓	✓	✓
ILP	✓	✓	✓
Latency Hiding	✓	✓	✓
Pipelining	✓	✓	✓
L/S Combine	✓	✓	✓
Gather/Scatter	-	✓	✓
Array Partitioning	✓	✗	✗
Domain Locality	-	✓	✓
Software I-Cache	-	-	✓
Loop Distribution	-	-	✓

a 20% performance improvement compared to when quadrant affinity is not exploited.

VII. Discussion and Conclusions

In this paper we have described the co-design of a compiler and an accelerator for processing near memory through a case study on a real-world hydrodynamics application. We were able to refine the architecture and integrate compiler optimizations through progressive, iterative acceleration from simpler component kernels to a complex procedure.

A. *Co-design: Progressive Acceleration of a Proxy Application*

By using an iterative methodology that builds from constituent kernels to a proxy application, we were able to develop relevant compiler optimizations and refine the architecture piecemeal. We found the profile of the two extracted kernels to be quite different compared to CalcKinematicsForElems.

Table I summarizes the applicability of the developed optimizations. A few optimizations developed for the simple kernels were directly applicable to the application (scheduling to exploit instruction-level instead of data-level parallelism, latency hiding, register use balancing); a few partially useful (load/store combining); and still others not directly applicable (we could not exploit quadrant affinity through array partitioning but it led us to consider domain locality).

Indeed, without codesign, we would only have developed optimizations of rows 1-4 and 6 in Table I. Compiling LULESH with these optimizations gives a floating-point efficiency of only 4.6%. Our methodology helped identify and build all optimizations in Table I via three iterations through Det, HexV, and CalcKFE. Examples of the effects of codesign are: HexV requiring gather/scatter to enable vectorization, and CalcKFE requiring loop distribution and software I-Cache to enable software pipelining.

B. Co-design: Architecture Refinements

We made several refinements in the architecture during the co-design process. We found real-world applications to have a substantial mix of scalar and vector instructions. The original scheme that separated scalar from vector lane instructions resulted in a large footprint. In response, we changed the ISA to produce more compact code.

Initially the scatter/gather instruction only used a single vector operand containing random addresses. We found that we were able to generate more efficient unstructured mesh accesses if these instructions accepted two operands instead, a base address and an offset.

A slice may access any of the vector register files through one of four shared ports. When exploiting instruction-level parallelism we detected a flaw in the ISA where two of the four ports were over utilized. This diminished the possibility of scheduling binary arithmetic and memory instructions in parallel. By modifying the ISA to favor the first two ports for arithmetic and the last two for memory instructions, we were able to remove this constraint.

C. Optimizations for Near Memory Processing

We developed three classes of compiler optimizations for high performance near memory, in order of decreasing importance, those that hide latency to memory, manage bandwidth demand, and increase compute efficiency. The focus on the memory subsystem is contrary to intuition given near memory processing, but the direct path to DRAM and the focus on applications with minimal data reuse necessitates it.

Vectorization (temporally on a slice) instead of traditional SIMD (for example, four-way on slices) is more effective at hiding latency so the compiler maximizes vectorization opportunities. A scheduler that considers latency hiding as a primary objective is required. Non uniform memory accesses across quadrants is exploited by intelligently partitioning data and by exploiting locality inherent in an application. Further research is required to understand how this can be automated.

To control memory bandwidth demand the compiler exploits instruction-level parallelism through co-operative scheduling across slices rather than treating them as independent units to exploit four-way data-level parallelism. The compiler ensures that at most one memory operation is issued per cycle across the four slices and combines unit-strided accesses to reduce memory traffic and fully exploit vault bandwidth. Finally, we found software pipelining is required to increase utilization of slices. Given the direct path to memory, the scheduler must avoid register spills.

CalcKinematicsForElems was more challenging to the compiler than the simplified kernels. In order to successfully optimize the proxy application, the compiler had to be more powerful, sometimes exceeding common assumptions in traditional compilers. One example is in the scalarization of array references. There is a relatively large two dimensional array in LULESH that must be scalarized. Commonly, a conservative heuristic is used to avoid unnecessary code explosion. In the context of compiling for non-traditional accelerators, the design of such heuristics is not recognized or well understood.

Another example is code scheduling. In the whole program, besides latency hiding, it is important to reduce register spills. The scheduling algorithm must adjust its heuristics for different goals (reducing spills vs. increasing slice utilization) depending on the input code. A traditional compiler is seldom designed for multiple optimization objectives. We also highlighted the importance of high level loop transformations to avoid spilling. This requires changing the cost model of loop optimization frameworks developed for cache-based architectures and is an interesting area for future study.

We found a few necessary compiler optimizations were missing in traditional compilers. Even though array reduction is recognized as one of the more important parallelization techniques by Polaris [17] from the ’90s, many compilers still do not have automatic recognition for array reduction. This may partly be because references in array reductions are often non-affine and need complex handling.

VIII. Acknowledgements

This work was supported in part by Department of Energy Contract B599858 under the FastForward initiative.

References

[1] P. Kogge, “EXECUBE-A new architecture for scaleable MPPs,” in *Conf. Parallel Processing*, vol. 1, Aug 1994, pp. 77–84.

[2] D. H. Woo, N. H. Seong, D. Lewis, and H.-H. Lee, “An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth,” in *High Performance Computer Architecture*, 2010.

[3] D. H. Kim *et al.*, “Design and analysis of 3D-MAPS,” *Trans. Computers*, vol. 64, no. 1, pp. 112–125, Jan 2015.

[4] D. Zhang *et al.*, “TOP-PIM: Throughput-oriented programmable processing in memory,” in *Par. and Dist. Comp.*, 2014, pp. 85–98.

[5] R. Nair *et al.*, “Active memory cube: A processing-in-memory architecture for exascale systems,” *IBM J. Res. and Dev.*, vol. 59, 2015.

[6] J. Shalf *et al.*, “Rethinking hardware-software codesign for exascale systems,” *Computer*, vol. 44, no. 11, pp. 22–30, Nov 2011.

[7] “Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory,” Tech. Rep. LLNL-TR-490254.

[8] I. Karlin *et al.*, “Exploring traditional and emerging parallel programming models using a proxy application,” in *Symp. Parallel and Distributed Processing (IPDPS)*, 2013, pp. 919–932.

[9] W. Wang *et al.*, “Energy auto-tuning using the polyhedral approach,” in *Workshop on Polyhedral Compilation Techniques*, 2014.

[10] M. Chabbi *et al.*, “Effective sampling-driven performance tools for GPU-accelerated supercomputers,” in *Supercomputing*, 2013.

[11] M. Sottile *et al.*, “Semi-automatic extraction of software skeletons for benchmarking large-scale parallel applications,” in *Principles of Advanced Discrete Simulation*, 2013.

[12] K. Asanovic *et al.*, “The landscape of parallel computing research: A view from berkeley,” University of California, Berkeley, Tech. Rep.

[13] M. Mohiyuddin *et al.*, “A design methodology for domain-optimized power-efficient supercomputing,” in *Supercomputing*, 2009.

[14] Z. Sura *et al.*, “Data access optimization in a processing-in-memory system,” in *Computing Frontiers*, 2015.

[15] J. Llosa, “Swing modulo scheduling: A lifetime-sensitive approach,” in *Parallel Architectures and Compilation Techniques*, 1996.

[16] B. R. Rau, “Iterative modulo scheduling: an algorithm for software pipelining loops,” in *Symp. Microarchitecture*, 1994, pp. 63–74.

[17] W. Blume *et al.*, “Parallel programming with Polaris,” *Computer*, vol. 29, no. 12, pp. 78–82, 1996.