Data Access Optimization in a Processing-in-Memory System

Zehra Sura Arpith Jacob Tong Chen Bryan Rosenburg Olivier Sallenave Carlo Bertolli
Samuel Antao Jose Brunheroto Yoonho Park Kevin O'Brien Ravi Nair
{zsura,acjacob,chentong,rosnbrg,ohsallen,cbertol,sfantao,brunhe,yoonho,caomhin,nair}@us.ibm.com
IBM T.J. Watson Research Center, Yorktown Heights, NY 10598

ABSTRACT

The Active Memory Cube (AMC) system is a novel heterogeneous computing system concept designed to provide high performance and power-efficiency across a range of applications. The AMC architecture includes general-purpose host processors and specially designed in-memory processors (processing lanes) that would be integrated in a logic layer within 3D DRAM memory. The processing lanes have large vector register files but no power-hungry caches or local memory buffers. Performance depends on how well the resulting higher effective memory latency within the AMC can be managed. In this paper, we describe a combination of programming language features, compiler techniques, operating system interfaces, and hardware design that can effectively hide memory latency for the processing lanes in an AMC system. We present experimental data to show how this approach improves the performance of a set of representative benchmarks important in high performance computing applications. As a result, we are able to achieve high performance together with power efficiency using the AMC architecture.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*heterogeneous (hybrid) systems*

1. INTRODUCTION

Driven by the need for power efficiency, heterogeneous computing is being used in current high-performance computing systems, and is also being considered in the design of future systems. In the November 2014 version of the Green500 list [4], which tracks the most energy-efficient supercomputers, the top 23 systems are all heterogeneous. Heterogeneous systems include processors with multiple distinct designs. Some of these processor designs (which we refer to as device processors or accelerators) are tailored to exploit specific application characteristics in order to provide both performance and power efficiency. For example, widely-used GPU processors are designed to efficiently execute data-parallel code.

CF'15, May 18-21, 2015, Ischia, Italy

ACM 978-1-4503-3358-0/15/05 ...$15.00
http://dx.doi.org/10.1145/2742854.2742863

We have explored the design of a system, called the Active Memory Cube (**AMC**) system [14], that provides high power-performance efficiency. It uses general-purpose host processors together with specially architected in-memory processors (called AMC *processing lanes*) that would be integrated in a logic layer within a 3D DRAM memory chipstack. An AMC node modeled in 14 nm technology running the double precision matrix-matrix multiply kernel, DGEMM, is projected to achieve 20 GFlops/s/W, compared to 5 GFlops/s/W for current GPUs in 28nm technology [6].

The AMC system has unique memory access characteristics compared to other recent and prevalent heterogeneous systems. Figure 1 illustrates these differences. Figure 1(a) shows a system in which the main memory is directly accessible by host processors only. The device processors are equipped with separate memory that is used to hold the data on which these processors will operate. An example of such a system is the L-CSC supercomputer built with AMD GPUs as the device processors, which is ranked first in the November 2014 Green500 list. Figure 1(b) shows a system in which the host processors and device processors can both access the system-wide main memory, and the device processors are also equipped with local memory. An example of such a system is the IBM Cell BE architecture [2]. The accelerators (SPE processors) have their own local stores and can directly access memory using DMA engines and a shared address space. Figure 1(c) shows a system in which the device processors are integrated in main memory and can directly access it. The AMC system that is the basis of this paper has this organization.

The AMC system architecture includes many design choices and new features, detailed in [14], that contribute to its power efficiency. In this paper, we primarily focus on the issues and performance implications of two of these design choices:

- Use of in-memory processors: Because computation can be done within the memory chip, power is not wasted in moving data to off-chip compute engines. Further, the proximity to memory reduces the latency of memory accesses.
- No caches or scratchpad memory for in-memory processors: This design choice saves area and power that would have been spent on cache/memory management in hardware. All memory accesses from AMC processing lanes go directly to DRAM, and as a consequence their latency is not mitigated by a cache hierarchy.

These design choices result in memory access characteristics that are starkly different from commonplace cache-based architectures. In particular, every access to memory goes directly to DRAM and has a higher latency than the average cached access in a traditional processor. We describe how programming language features, compiler technology, operating system interfaces, and hardware design synergistically work together in the AMC system to hide memory

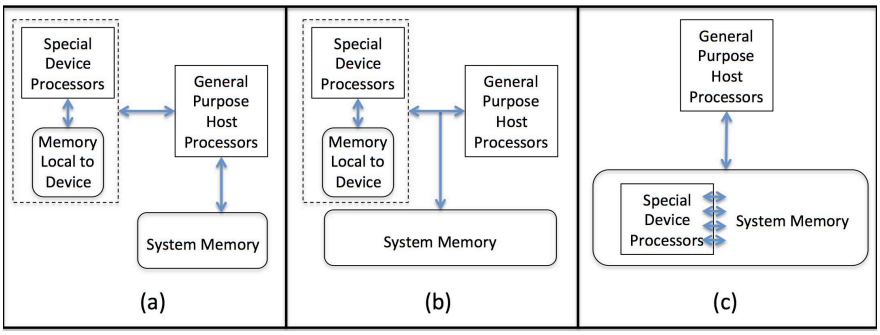

Figure 1: Memory Arrangements in Heterogeneous Systems

latency and achieve high performance.

In Section 2, we describe the AMC architectural features that are relevant to the discussion in this paper. In Section 3, we review the OpenMP 4.0 model used in programming the AMC system, and show how it helps in extracting performance. In Section 4, we describe techniques that help to reduce the performance impact of memory access latency in AMC processing lanes. In Section 5, we evaluate the effectiveness of our techniques for reducing the latency overhead. In Section 6, we discuss related work, and in Section 7 we present our conclusions.

2. AMC ARCHITECTURE

In the AMC system concept, a compute node contains an interconnected host procssor and up to 16 AMCs. Each AMC is a 3D chip with 8GiB of DRAM memory and 32 in-memory processing lanes. The DRAM memory is arranged in 32 partitions, or *vaults*, of 256MiB each. Neighbouring vaults are grouped into four *quadrants* of eight vaults each. Each quadrant also contains eight processing lanes. The lanes and vaults in a quadrant are connected by a crossbar switch, and the quadrants are fully interconnected with point-to-point links. Figure 2 shows the organization of lanes, vaults and quadrants within an AMC, and how they are interconnected with each other.

The hardware automatically provides coherence across all memory accesses in the system. The exact latency of a memory access from a processing lane is difficult to predict, because it is affected by dynamic conditions such as bank conflicts and whether a page in the DRAM memory controller is open or not. However, a processing lane is likely to observe lower latency to the vaults in its own quadrant than to vaults in other quadrants. The additional interconnection penalty for accessing a remote quadrant is approximately 24 processor cycles. The peak bandwidth available in an AMC is 320 GB/s for reads and 160 GB/s for writes. The processing lanes and DRAM are modeled at 1.25 GHz.

In our design concept, a processing lane cannot directly access memory in another AMC chip. The host processor would need to copy data from one AMC to another to make it accessible to the processing lanes in the second AMC. The compiler strives to minimize the need for such transfers.

Memory addresses generated by programs running on AMC processing lanes are effective addresses with standard memory translation and protection facilities. However, to save power, the processing lanes only support a software-managed 8-entry ERAT (effective-to-real address translation) table. The entries translate arbitrary ranges of virtual addresses, not fixed-size pages, so even the limited number of entries provided by a lane are often sufficient to map all the memory needed by a lane computation. The host operating system is responsible for populating the ERAT table and updating the table when it receives a signal for an address translation miss.

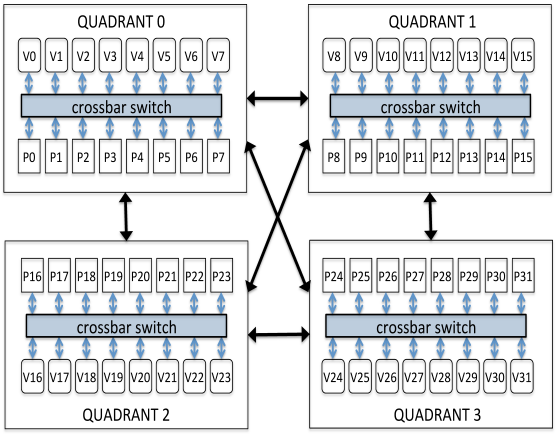

Figure 2: Schematic of Vaults and Quadrants in an AMC

The AMC architecture supports several different interleaving modes simultaneously. A given effective address range can be backed by memory that is interleaved across all the AMCs in the system, interleaved across all 32 vaults of a single AMC, interleaved across the 8 vaults of a particular AMC quadrant, or is contained in a single vault. Memory interleaved in the first mode (across all AMCs) cannot be used by processing lanes but is useful for generic host-only applications for which balanced usage of memory capacity and bandwidth is desirable. The other three interleaving modes can be used by the host and by processing lanes.

Figure 3 shows the architecture of an AMC processing lane. The processing lanes are designed to provide highly parallel and flexible execution. A lane instruction can simultaneously execute scalar, vector, and SIMD operations using 4 *slices* within a lane. Each slice has its own functional units for arithmetic and load/store operations. A slice is capable of executing 2 floating-point operations (corresponding to a multiply-add operation) per cycle. The

length of vectors for vector operations within an instruction is programmable, and can be different for each instruction.

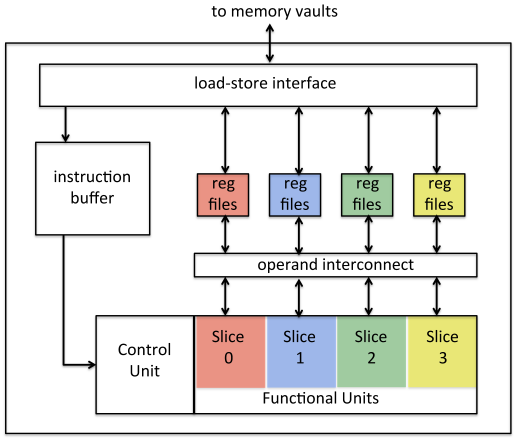

Figure 3: AMC Processing Lane Architecture

A processing lane also has a large number of registers: each of the 4 slices has 32 scalar registers, 4 vector mask registers, and 16 vector compute registers, with a vector size of 32 8-byte elements. The total size of the vector compute register file in a lane is 16KiB. The scalar registers and vector mask registers are accessible only within their own slices, but the vector compute registers can be shared across slices, with some limitations:

- An operation in one slice can read the vector registers of all other slices, but can only write to the vector registers in its own slice.
- The register read and write operations within an instruction must respect the register port constraints in the architecture.

Except for memory access operations, the processor pipeline is fully exposed: when generating code, the user/compiler is responsible for inserting sufficient delays in the instruction stream between the producer and consumer instructions. For memory accesses, the hardware provides an interlock that stalls the entire lane when an instruction attempts to use the result of a pending load operation.

3. PROGRAMMING MODEL

We use the OpenMP 4.0 [15] programming model for application code that will execute on AMC processing lanes. It is possible for a single process to use multiple AMCs, but it is also possible to use MPI [8] to structure an application as multiple processes, each of which uses a single AMC. For the purpose of compiler optimization and evaluation, we focus on execution within a single AMC. In this section we describe the OpenMP constructs that are most relevant to exploiting heterogeneity in the AMC architecture, and show how they are used in the AMC context.

OpenMP is a widely-known standard model for parallel programming using directives. OpenMP 4.0 added support for heterogeneous processors or *devices*, introducing the *target* directive for offloading computation to another device.Each device in the system is associated with an independent data environment, i.e., a collection of variables accessible to it. Variables must be explicitly communicated or mapped from the host data environment to that of the device that requires access.

The format of the target directive is as follows:
#pragma omp target [clauses]
structured block
where the structured block is the target region, i.e. the code that will be executed on a device processor, and optional clauses are of type *map* and *device*.

3.1 Map Clause

One or more map clauses specify scalar and array variables that may be accessed in the target region, and must be mapped to the device data environment. The default assumption is that mapping will require making the data values at the entry point of the target region available in the device data environment, and reflecting the data values at the end of the target region so that they are visible to the host processor. The map clause allows the specification of an optional direction of communication that is used to optimize data mapping. A direction-type of *alloc* specifies that values of variables in the map list need not be copied over, direction-type *to* specifies that values of variables need to be copied only on entry to the target region, and direction-type *from* specifies that values of data items need to be made available to the host processor on exit from the target region.

Listing 1 shows the matrix-matrix multiply kernel, DGEMM, written using OpenMP 4.0 directives. Note that parallel regions and loops within target regions can be identified using regular OpenMP directives for parallelism. In the DGEMM example, a *parallel for* directive is used to make the offloaded loop run in parallel on 32 processing lanes of the AMC, with iterations of the i loop distributed across the lanes.

```
double A[P][R], B[R][Q], C[P][Q];

void main() {
// Initialize arrays

// Copy all arrays to AMC 0 (default device),
// if not already resident
// Offload loop nest for acceleration
#pragma omp target map(to:A[0:P*R],B[0:R*Q])
                   map(tofrom:C[0:P*Q])

// Execute iterations of loop i in parallel
// on 32 lanes of the AMC
#pragma omp parallel for num_threads(32)
for (int i=0; i<P; i++)
  for (int j=0; j<Q; j++)
    for (int k=0; k<R; k++)
        C[i][j] += A[i][k] * B[k][j]

// Computed array C is available on the host
}
```

Listing 1: DGEMM Code Using OpenMP 4.0

The memory on an AMC chip is shared between the processing lanes on that chip and the host processor. The hardware includes a coherence mechanism to keep all memory accesses coherent across the processing lanes and host processor. Because of this shared memory architecture, data communication can be eliminated in certain cases: if a program uses processing lanes within a single AMC, and all of its data is resident within the associated DRAM, then the data environment of the host processor can be directly used by the processing lanes, and there is no overhead for data communication. The OpenMP compiler for the AMC system optimizes processing of map clauses for this special case, eliminating all associated overhead. In general, our OpenMP compiler uses the information provided in map clauses for two purposes: data placement to ensure that all data accessed in the target region is contained in the memory of the AMC whose processing lanes

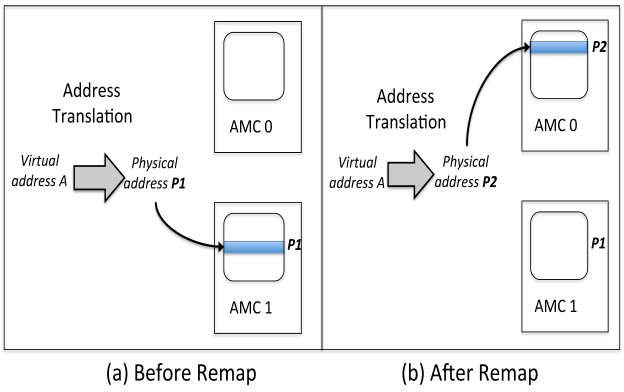

Figure 4: Data Remapping

will execute the target region, and to optimize the effective-to-real address translation (ERAT) in the processing lanes.

Data Placement for Target Regions: Data placement can be efficiently done by *remapping* data that is interleaved across AMCs or is contained in the memory of another AMC, invoking an operating system utility that keeps the same virtual address for that data, but switches the physical memory to which the address translates. Remapping is illustrated in Figure 4, where data address *A* refers to a location *P1* in AMC1 before the remap, but refers to location *P2* in AMC0 after the remap. Remapping requires an initial copying of data, but it can eliminate repeated copying of data on every entry/exit of a target region. However, it is limited in that it can only remap whole pages at a time, with the page size fixed in the operating system. When multiple AMCs are in use, the programmer/-compiler must ensure that data used by processing lanes in different AMCs is not co-located in the same virtual memory page. In programs that primarily compute on large array data structures or that are programmed using an MPI model across AMCs, this can often be accomplished by allocating memory for arrays/data structures aligned to page boundaries, and by padding arrays/data structures to a size that is an integral number of pages.

Data ranges can be remapped into any of the three interleaving modes (AMC-wide, quadrant-wide, or single-vault) that are usable on processing lanes, and target locations (AMC, quadrant, or vault) can be specified. The operating system also provides facilities for dynamically allocating memory of any type and location. Dynamic allocations do not have the page-based constraints that apply to remapping operations.

For cases in which data used by processing lanes in multiple AMCs are located in the same memory page, our OpenMP compiler runtime dynamically allocates and maintains a copy of the data for each AMC in which that data is used, copying it on entry to and exit from target regions, as appropriate. However, this mode of operation is expensive and not recommended.

ERAT Optimization: Each AMC processing lane has an effective-to-real address translation (ERAT) table with eight entries. When memory is accessed in a lane, the hardware accesses this table to find a translation for the memory address. If the required translation entry is not found, an exception is thrown to allow the host operating system to install the missing entry. However, a translation miss is an expensive event, so it is preferable to pre-load all required ERAT entries ahead of time.

Before entry to a target region, the compiler inserts a call to an operating system utility function, passing the address and size of each variable listed in the region's map clauses. The operating system can then install initial sets of the needed entries in the ERAT tables of the processing lanes.

An AMC ERAT table entry covers a contiguous range of addresses, not just a single page of some fixed size. With careful assignment of virtual and physical addresses, and with hints from the compiler about what translations will be needed by a lane computation, the operating system is almost always able to pre-load lane ERAT tables with entries that cover all the data accessed in the target region, despite the fact that the tables have just eight entries. Eight entries have been sufficient for all code that we have compiled and run, including code used in the experiments described in Section 5.

3.2 Device Clause

There can be at most one device clause per target region. It identifies the device to use for mapping data and for offloading computation. When no device clause is specified, the implementation defines a default device, which is AMC0 in our case. AMCs (and quadrants and vaults) have logical names within a process, so AMC0 in one process and in another will refer to different physical AMCs. The device clause uses an integer expression to identify a particular device, and the implementation defines a mapping of integers to devices. For an AMC system, a simple mapping would associate an integer with a corresponding AMC, so for a system with 16 AMCs, valid devices would be in the set [0,1,..,15], corresponding to [AMC0, AMC1, ..., AMC15].

However, AMCs have a substructure (quadrants and vaults) that can sometimes be exploited using the data allocation and remapping functionality described in Section 3.1. Therefore we include in the targetable device space not only entire AMCs but also individual quadrants and even individual lanes. This mapping is as follows:

- use the entire AMC0 : [0] corresponds to [AMC0]
- use a specific lane and vault in AMC0 : [1,2,...,32] correspond to [AMC0_lane0, AMC0_lane1, ..., AMC0_lane31]
- use the lanes and memory in a specific quadrant in AMC0 : [33,34,35,36] correspond to [AMC0_quad0, AMC0_quad1, AMC0_quad2, AMC0_quad3]
- use the entire AMC1 : [37] corresponds to [AMC1]
- ...

When the device for a target region is an entire AMC, variables named in the region's map clauses are placed in memory interleaved across all 32 vaults of the AMC, and any or all of the AMC's lanes may be enlisted in the computation. When the device is a specific quadrant, data is mapped into memory interleaved across the 8 vaults of that quadrant, and the quadrant's 8 lanes may be used. When a specific lane is targeted, data is placed in the vault with the same number as the lane. For the latter case, we choose for convenience to pair individual lanes and vaults based on their indices, even though there is no particular affinity between a lane and any given vault within its quadrant.

Listing 2 shows skeleton code that uses the device clause for execution on multiple specific devices.

4. MEMORY LATENCY OPTIMIZATIONS

This section describes memory latency optimizations that are implemented using our OpenMP compiler for the AMC system. This compiler is built on the IBM XL compiler, which is an industrial-strength C, C++, and Fortran compiler. Figure 5 shows a schematic diagram of the compiler.

```
#pragma omp for
for (int i=0; i<N; i++)  {
    int device_num = foo(i);
    #pragma omp target device(device_num)
    {
        ...
    }
}
```

Listing 2: Code Using the OpenMP Device Clause

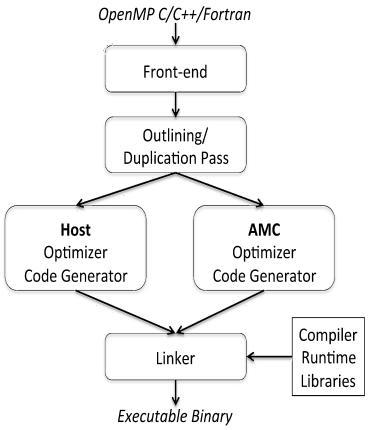

Figure 5: Schematic of OpenMP Compiler for AMC System

The compiler front-end processes the input program and specially marks code regions associated with target directives. It also records the list of variables identified in map clauses. The outlining/duplication phase creates a new function for each target code region, and fills the body of this function with a copy of the code in the target region. It also inserts a runtime library call before each target region to reserve AMC processing lanes and offload computation to them. At this stage, there are two sets of functions within the code being compiled: those that will execute in host processors, and those that were generated in the previous compiler step to execute in AMC processing lanes. Each of these sets of procedures follows a different compilation path for optimization and code generation, but eventually they both get compiled into ELF object files, and are linked together with the compiler runtime libraries to produce a single executable binary.

The AMC-specific part of the compiler includes most classical compiler optimizations, including data-flow, control-flow, and loop optimizations. For some optimizations (such as inlining and loop unrolling), only the heuristics need to be tailored to fit the characterisitics of the AMC architecture. Other optimizations (loop blocking, reduction handling, data access pattern analysis, vectorization, instruction scheduling, register allocation) have been re-developed specifically for the AMC architecture.

In this paper, we only focus on the memory latency optimizations that we have developed for the AMC architecture. These include optimizations to *hide* the memory latency, to *reduce* the memory latency, and to *eliminate* the memory latency.

4.1 Hiding Latency to Memory

Registers in an AMC processing lane are directly loaded from DRAM memory, without any intervening caches. Memory references within any of the 4 slices of a processing lane enter a single load-store queue at the lane, traverse an on-chip network, and are serviced at a DRAM vault. The latency of this operation depends on the utilization of the load-store queue, the traffic on the network, the location of the vault that contains the requested data, any bank conflicts at the destination vault, and whether the DRAM page being accessed is open or closed. Optimizing for each of these factors is important for high performance.

The AMC compiler uses a sophisticated instruction scheduler that optimizes for the complex exposed pipeline of the AMC processing lane achitecture. This scheduler also hides latency of an access by enforcing a minimum separation of a predetermined number of cycles between the issue of a load instruction and its first use. The hardware has interlocks for load dependencies, which stalls all 4 slices in the processing lane if a consumer attempts to read a data value that has not yet arrived. In contrast, the compiler can schedule non-dependent operations such that some of the slices in the lane do useful work while waiting for the value of the load to become available. By explicitly separating a load from its first use, and by scheduling other instructions in the intervening period, program stalls can be significantly reduced. The scheduler incorporates software pipelining, which is well-suited for scheduling instructions within this interval.

The instruction scheduler also tracks the instantaneous occupancy of the load-store queue to ensure that it never exceeds its capacity. The issue of load or store instructions may be deliberately delayed if the scheduler detects the possibility of a highly-loaded queue. This approach is able to considerably reduce lane stalling due to a full load-store queue.

Currently, the minimum number of cycles by which a load should be separated from its first use is fixed in the compiler to be 32 cycles (maximum length of a vector operation). There is a compiler command-line flag that allows changing this fixed value, but the same value is used for all loads in the code. This functionality is sufficient for an initial evaluation of the AMC performance. Ideally, the number of cycles for latency hiding would be intelligently determined on a per-load basis, taking into account information about the affinity of data referenced by the load (located in a local vault, a local quadrant, or a remote quadrant), and the density of compute operations available to fill the instruction schedule.

4.2 Reducing Latency Through Data Affinity

The latency of a sequential access to DRAM memory varies a lot (from tens of cycles to 250+ cycles) depending on the traffic contention in the interconnect network, whether a lane accesses a single vault, vaults in its local quadrant, or vaults in remote quadrants. This non-uniform access time is primarily due to the hierarchical design of the memory network, guided by the goals of low complexity and low power consumption.

The operating system exports routines to allow memory allocation within a specific vault or specific quadrant to enable users to exploit this characteristic. The AMC compiler implementation is able to automatically partition one-dimensional arrays into local quadrants, but relies on the programmer to manually place more complex pointer-based data structures within the desired regions. The use of the OpenMP device clause in exploiting data affinity on the AMC has been discussed in Section 3.2. Subsequently, the compiler can exploit the lower latency to decrease the scheduled interval between the issue of loads and their first use, often resulting in a schedule that lowers execution time.

4.3 Eliminating Memory Accesses via Reuse

The AMC processing lane architecture provides large vector register files instead of power-hungry data caches. Each lane is equipped

with 64 32-element vector registers, totaling 16KiB of vector compute registers. For loops that exhibit data reuse, it is possible to exploit the large register file to reduce traffic to memory. Currently we rely on a combination of compiler support and some manual refactoring (applying specific standard loop transformations by hand) for this purpose. The manual loop transformations can later be automatically applied in the compiler, once an appropriate cost model has been built into the compiler loop optimization framework.

We illustrate the optimization using Listing 3, which is a re-written version of DGEMM. Loops *j* and *k* have been tiled (tile iterators not shown) with tile sizes 32 and 48 respectively. The compiler fully vectorizes loop *j* and eliminates the control flow. Thereafter, the compiler transforms the code such that elements of array *B* are loaded into 48 vector registers; this code is invariant within the loop nest and is hoisted out, achieving the desired staging of data within vector registers. Finally, the scheduler pipelines iterations of loop *i*.

```
#pragma omp target map(to:A[0:P*48],B[0:48*32]) \
                   map(C[0:P*32])
//Parallelize iterations of loop i across 32 lanes
//and pipeline loop iterations
#pragma omp parallel for num_threads(32)
for (int i=0;i<P;i++) {

  //Fully vectorize and eliminate loop j
  for (int j=0;j<32;j++) {

    //Array B is staged at startup into 48 vector regs,
    //and reused for the entire duration of loop nest
    C[i][j]+=A[i][0]*B[0][j] + \
             A[i][1]*B[1][j] + \
             ...
             A[i][47]*B[47][j];
  }
}
```

Listing 3: DGEMM Re-written to Exploit Reuse

5. EVALUATION

We evaluate performance by running compiled binaries on a simulator that models a host processor and the AMC. We use a functional simulator for the host processor to allow us to simulate the entire operating system. The AMC is modeled by a timing-accurate simulator that precisely represents lanes, the memory interconnect, the vault controllers, and the DRAM [14].

In our experiments, performance is reported in terms of *flop efficiency*, which we define as the fraction of the lanes' peak theoretical FLOP rate utilized in an execution. It is calculated as the number of floating-point operations retired per cycle per lane divided by 8 (because there are 4 slices in a lane, and each slice is capable of executing 2 floating-point operations per cycle corresponding to a multiply-add operation). All binaries generated in our experiments for a given benchmark contain the same number of floating point operations to execute at runtime.

5.1 Benchmark Kernels

We added OpenMP directives to several codes important to the supercomputing community, to offload and parallelize sections of the computation on AMC processing lanes. We present results for 3 benchmark kernels that are important in high performance applications, and that all depend on the performance of the memory subsystem:

- *DAXPY*: a memory-bound kernel that stresses the memory subsystem of the AMC.

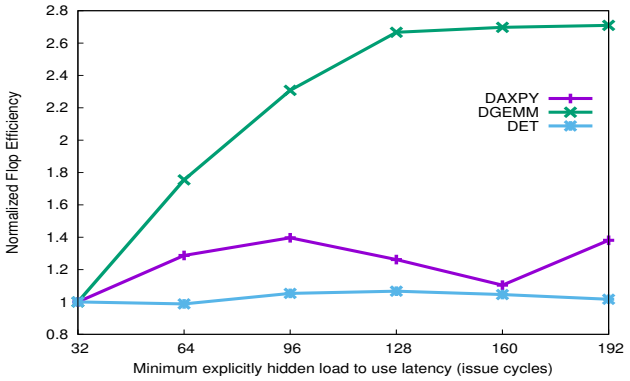

Figure 6: Effect of Latency Hiding in Compiler

- *DGEMM*: a double-precision matrix-matrix multiply kernel often used to benchmark high performance systems. It tests the ability of the compiler to exploit vector registers for data reuse to achieve high floating point utilization.
- *DET*: a kernel that computes the elemental volume by calculating the determinant of three matrices. The code presents an interesting case for the AMC compiler, because the compiler has to map an odd number of compute strands (three determinant calculations) onto the four slices of a processing lane, which is not easy to manually code. This benchmark is similar to the computation code in the LULESH application [11].

Our evaluation is focused on AMC performance, and these kernels are representative of the code sections from whole applications that we would expect to be offloaded for computation on AMC lanes.

5.2 Compiler Latency Hiding

In this section, we report on the effectiveness of the latency hiding optimization described in Section 4.1. We use a compiler command line flag to specify the number of latency cycles the compiler tries to explicitly hide by separating loads from first uses. We generate multiple code binaries for each benchmark, varying the minimum load-to-use interval, and we run each binary on 32 processing lanes. Figure 6 plots the performance of these runs, showing how the normalized flop efficiency (Y-axis; higher is better) changes as the compiler increases the minimum number of cycles between each load and its first use (X-axis). The base case for normalization is explicit hiding of 32 cycles, which is the default behavior of the compiler and corresponds to the maximum vector length of 32 elements.

We observe that increasing the number of cycles of latency hiding has relatively low impact on the performance of DET; its maximum performance at 128 cycles is 6.67% higher than its performance at 32 cycles. DET is a compute-bound kernel that has many arithmetic/logic operations, so even in the base case with only 32 cycles of explicit latency hiding, the scheduler naturally arrives at a schedule that effectively hides latencies. There is a higher impact on the performance of DAXPY; its maximum performance at 96 cycles is 39.7% higher than its performance at 32 cycles. DAXPY is a memory-bound computation with few arithmetic/logic operations, but the compiler can effectively use unrolling and software pipelining to hide latency. The highest impact is on the performance of DGEMM; its maximum performance at 192 cycles is 2.7 *times* its performance at 32 cycles.

We also measure the impact on performance of DGEMM when

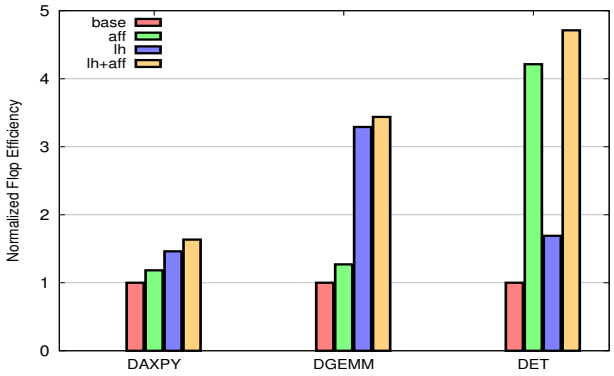

Figure 7: Performance of Latency Hiding and Data Affinity

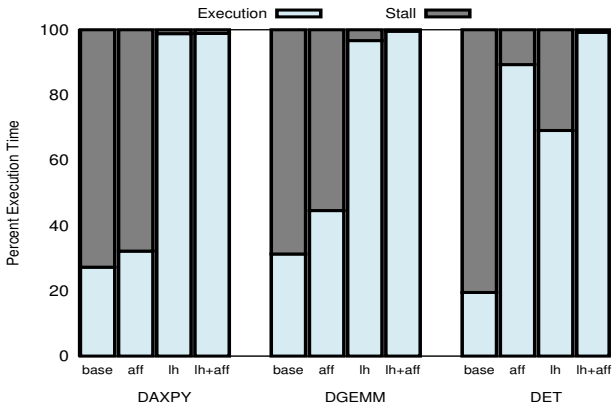

Figure 8: Exectuion Time vs Memory Stall Time

transformations to exploit data reuse are applied, as described in Section 4.3. DGEMM exhibits significant improvement in performance (3.63 times) when reuse is exploited and the number of accesses to the memory subsystem is reduced.

5.3 Data Affinity

Figure 7 shows the performance effects of various data-placement choices. Performance is shown in terms of normalized flop efficiency (plotted on the Y-axis; higher is better) when executing using 32 processing lanes. For each benchmark, we plot 4 bars:

- **base**: the base case where data is allocated across the entire AMC memory, and scheduling in the compiler hides load latencies for the default 32 cycles.
- **aff**: the case where data is allocated in the quadrants in which it is used, with default latency hiding.
- **lh**: the case where data is allocated across the entire AMC memory, and latency hiding optimization (as described in Section 4.1) is applied.
- **lh+aff**: the case where data is allocated in quadrants, and latency hiding optimization is applied.

When latency hiding is used, the minimum number of cycles separating each load and its first use is the value experimentally determined to be best for each benchmark (refer to Section 5.2).

We observe that both affinity and latency hiding optimizations help improve the performance of all 3 benchmarks, and combining both optimizations results in higher performance compared to

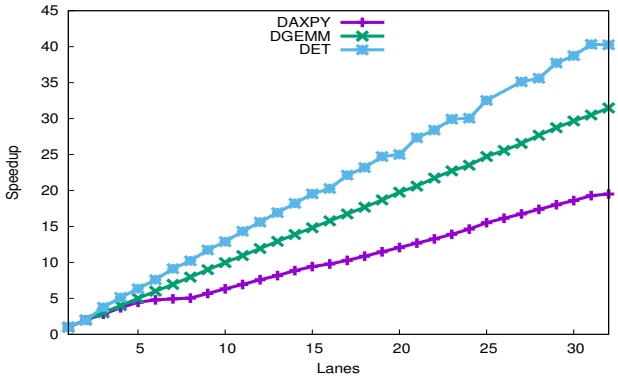

Figure 9: Performance Scaling Across Multiple Lanes

either optimization alone. Figure 8 shows the percentage of execution time spent stalling in the load-store units. The bars plotted in Figure 8 correspond to the performance bars plotted in Figure 7, and there is a clear correlation between the amount of time spent in memory stalls and the observed performance.

In our experiments, using vault-specific data mapping resulted in the same performance as quadrant-specific mapping. However, depending on the code characteristics and data sizes, vault-specific mapping can potentially improve performance over quadrant-specific mapping.

When all lanes are active and data affinity is not exploited, performance compared to the single lane version shows a slight drop for DGEMM (95% of the best single lane version) and significant deterioration for DAXPY (45% of the best single lane version). This drop in performance directly correlates with an increase in the amount of execution time spent stalling in load-store units. Exploiting affinity, when possible, helps negate this loss. DET achieves higher performance with 32 lanes than the single lane case even without the use of data affinity.

5.4 Scaling

We study scaling of the compiler-optimized benchmark codes on the 32 lanes of the AMC, plotting the performance speedup relative to one lane in Figure 9.

DAXPY: A single lane achieves a flop efficiency of 8.5% but this drops to 5.3% when eight lanes are activated as they compete within their local quadrant. The drop in performance can be attributed to increased bank conflicts as observed by the rise in load latency from 150 to 270 cycles. The compiler is able to hide a latency of up to 192 cycles, i.e. the size of the load-store queue in each lane, by unrolling the loop 4 times. Because we are able to exploit data affinity, flop efficiency remains constant as each new quadrant is activated. For comparison, a hand-optimized version of DAXPY that organizes data within a local quadrant achieves a flop efficiency of 5.2%. For the compiled version with 32 lanes we achieve a read bandwidth of 136 GB/s and a write bandwidth of 68 GB/s, which utilizes a significant fraction of the peak memory bandwidth available on an AMC chip (320 GB/s for reads and 160 GB/s for writes). Also, when extrapolated to 16 AMCs in a node, this translates to using 2.1 TB/s of read bandwidth and 1.0 TB/s of write bandwidth, nearly 8 times what can be achieved using the host processor alone.

DGEMM: We see near perfect scaling for DGEMM, with a speedup of 31 on 32 lanes. The compiler is able to achieve a flop efficiency of 77% using techniques that include exploiting data reuse

and affinity, and hiding memory latency. This compares favorably with the best hand-optimized assembly code that achieves peak flop efficiency of 83% on the AMC.

DET: When compiled, this kernel has 82 ALU and 25 LSU operations. Only eight of the ALU operations are multiply-adds and the upper bound for flop efficiency with this mix is 46.4%. Scaling up to 32 lanes, we achieve a near constant flop efficiency of 38%, and the load latency is also nearly constant at about 130 cycles.

6. RELATED WORK

Power consumption and memory bandwidth are the biggest challenges associated with the design of exascale systems [5]. Recently, processing-in-memory solutions have been envisioned to overcome those challenges, as enabled through 3D die stacking technology. To our knowledge, none of these efforts have gone through a detailed conceptual design as described for the AMC [14].

For instance, the idea of placing compute logic under the die-stacked DRAM in a Hybrid Memory Cube (HMC) has been discussed in [1]. According to this study, PUM (processing-under-memory) is expected to increase the memory bandwidth, which can yield up to 4.2 speed improvements on studied exascale workloads, as a majority of them would be converted from memory-bandwidth-limited to compute-limited.

Similarly, many configurations of PIM (processing-in-memory) architectures have been envisioned in [17, 16]. Projections showed promising results to address data movement bottlenecks and overheads, however the paper states that PIM can only be adopted with appropriate programming models and strong support from the compiler, runtime and operating system.

Many-threads accelerator architectures such as the Intel Xeon Phi [9] and NVIDIA GPU [10] are now widely adopted [1]. Compared to these, the AMC architecture provides power-performance characteristics projected to be within the 50 GF/W targeted for Exascale computing [14]. In particular, there is no need to move the data from the main memory to the accelerator memory, which can represent a significant amount of time and energy (up to 50 times the execution of the kernel [7]).

7. CONCLUSION

The AMC is a processing-in-memory device designed for high-performance power-efficient computing. It gains power efficiency by moving computation to data, and by using a carefully designed microarchitecture that eliminates much of the complexity of conventional processors including the power-hungry cache hierarchy. Instead, the AMC design relies on sophisticated compiler, runtime, and operating system support to deliver maximum performance for the budgeted power.

In this work we have focused on the performance impact of eliminating the hardware cache hierarchy. We have described how hardware features (hierarchical memory layout, large vector register files, support for multi-way parallelism), operating system interfaces (for affinity-based data allocation and mapping-hardware management), compiler techniques (code transformations for hiding/reducing/eliminating latency of memory operations), and the programming model (OpenMP 4.0 accelerator pragmas and clauses) all work together to overcome the absence of hardware caches. We have achieved high computational efficiency and linear performance scaling on a representative set of benchmark kernels.

We believe that our study using the AMC architecture demonstrates the compelling value of processing-in-memory in the design of power-efficient scientific computing systems.

[1] See http://www.top500.org

8. REFERENCES

[1] P. Balaprakash, D. Buntinas, A. Chan, A. Guha, R. Gupta, S. H. K. Narayanan, A. A. Chien, P. Hovland, and B. Norris. Exascale workload characterization and architecture implications. In *Proceedings of the High Performance Computing Symposium*, pages 5:1–5:8, San Diego, CA, USA, 2013.

[2] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation-a performance view. *IBM Journal of Research and Development*, 51(5):559–572, Sept 2007.

[3] M. Chu, N. Jayasena, D. P. Zhang, and M. Ignatowski. High-level programming model abstractions for processing in memory. In *1st Workshop on Near-Data Processing in conjunction with the 46th IEEE/ACM International Symposium on Microarchitecture*, 2013.

[4] W. Feng and K. Cameron. The Green500 List: Encouraging sustainable supercomputing. *Computer*, 40(12):50–55, Dec 2007. http://www.green500.org.

[5] A. Gara and R. Nair. Exascale computing: What future architectures will mean for the user community. In *PARCO'09*, pages 3–15, 2009.

[6] R. Garg and L. Hendren. A portable and high-performance general matrix-multiply (GEMM) library for GPUs and single-chip CPU/GPU systems. In *Proceedings of the 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2014.

[7] C. Gregg and K. Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 134–144, Washington, DC, USA, 2011.

[8] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-passing Interface*. Number v. 1 in Scientific and engineering computation. MIT Press, 1999.

[9] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.

[10] D. Kirk. NVIDIA CUDA software and GPU parallel computing architecture. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, pages 103–104, New York, NY, USA, 2007. ACM.

[11] Lawrence Livermore National Laboratory. Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). https://codesign.llnl.gov/lulesh.php.

[12] B. Li, H. Chang, S. Song, C. Su, T. Meyer, J. Mooring, and K. Cameron. The power-performance tradeoffs of the Intel Xeon Phi on HPC applications. In *Proceedings of the Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2014.

[13] G. Mitra, E. Stotzer, A. Jayaraj, and A. Rendell. Implementation and optimization of the OpenMP accelerator model for the TI Keystone II architecture. In L. DeRose, B. de Supinski, S. Olivier, B. Chapman, and M. Muller, editors, *Using and Improving OpenMP for Devices, Tasks, and More*, volume 8766 of *Lecture Notes in Computer Science*, pages 202–214. 2014.

[14] R. Nair et al. Active Memory Cube: A Processing-in-memory Architecture for Exascale Systems. *IBM Journal of Research and Development*, 59(2/3), 2015.

[15] OpenMP ARB. OpenMP application program interface version 4.0, May 2013.

[16] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. TOP-PIM: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 85–98, New York, NY, USA, 2014.

[17] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. Greathouse, M. Meswani, M. Nutter, and M. Ignatowski. A new perspective on processing-in-memory architecture design. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '13, pages 7:1–7:3, New York, NY, USA, 2013.