RC25644 (WAT1612-008) December 5, 2016
Computer Science

IBM Research Report

Compiling for the Active Memory Cube

Arpith C. Jacob, Zehra Sura, Tong Chen, Carlo Bertolli, Samuel Antao, Olivier Sallenave, Kevin O'Brien, Hans Jacobson, Ravi Nair, Jose R. Brunheroto, Philip Jacob, Bryan S. Rosenburg, Yoonho Park, Alexandre E. Eichenberger, Changhoan Kim
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

Research Division
Almaden – Austin – Beijing – Brazil – Cambridge – Dublin – Haifa – India – Kenya – Melbourne – T.J. Watson – Tokyo – Zurich

Compiling for the Active Memory Cube

Arpith C. Jacob Zehra Sura Tong Chen Carlo Bertolli Samuel Antao Olivier Sallenave Kevin O' Brien Hans Jacobson Ravi Nair Jose R. Brunheroto Philip Jacob Bryan S. Rosenburg Yoonho Park Alexandre E. Eichenberger Changhoan Kim

IBM T.J. Watson Research Center, 1101 Kitchawan Rd., Yorktown Heights, NY, USA.

{acjacob,zsura,chentong,cbertol,sfantao,ohsallen,caohmin,hansj,nair,brunhe,jacobp,rosnbrg,yoonho,alexe,kimchang}@us.ibm.com

Abstract

In previous work we have introduced a novel processing-in-memory embedded device that achieves high power efficiency by moving computation to data, and with a carefully designed microarchitecture eliminating much of the hardware support and complexity of conventional processors. It relies on sophisticated compiler, runtime, and support software to deliver high performance.

In this work we describe the design and implementation of a compiler for this accelerator that uses the new OpenMP 4.0 accelerator model to offload and parallelize programs. We exploit architectural features such as VLIW and vector capabilities, hide latency to memory, and reuse data using the large vector register files. We achieve high computational efficiency, linear performance scaling, and superlinear performance per watt scaling on memory- and compute-bound kernels. Most importantly, we are able to achieve these results using standard, portable pragmas and no accelerator-specific program code. We believe our work is an important step toward building next-generation, power-efficient computing systems.

1. Introduction

Techniques from embedded devices have long been employed in the design of high-performance systems. The BlueGene series uses low-frequency, low-power embedded processor cores that consistently outperforms high-frequency, high-power microprocessors [7]. A high level of integration was also used so that many more cores in a chip could provide high aggregate performance.

Powerful fast computing systems drive important problems in fields such as energy science, climate modeling, and systems biology. To continue improving performance next-generation systems must be power-efficient. Dennard scaling no longer provides improvements in clock frequency at a constant power density without seriously compromising transistor reliability. To achieve performance improvements it has become necessary to improve upon the inefficiencies of Petascale systems. Figure 1a shows the power consumption of a BlueGene/Q node running an optimized version of the compute-bound double precision matrix-matrix multiply kernel, DGEMM. Just 14% of total energy is spent executing floating-point operations. Overheads include the energy spent within the integer pipelines, the fetch unit, the decode unit, and the issue logic (marked *Core: Other*). Access to external memory and the on-chip caches account for 45% of total consumption (labelled DRAM, Cache, Nest).

A second pressing concern is the limited external memory bandwidth available to applications due to constraints in the number of pins on the chip. Today's supercomputers only provide 0.2 bytes per second (bytes/sec) bandwidth to DRAM for every floating-point operation per second (flop/sec) on the core [9]. The energy to read operands of a floating-point operation from DRAM is projected to reduce at a much slower pace than the energy to perform the operation [21]; hence, either this historically low ratio of bandwidth to compute is likely to reduce even further, or the skewed allocation in Figure 1a will get worse as more energy is budgeted to DRAM and caches.

The *Active Memory Cube* (AMC) [3] is a novel processing-in-memory embedded accelerator created in response to these challenges that places compute elements directly under a 3D stack of DRAM layers known as the Hybrid Memory Cube (HMC) [10]. Thirty-two light-weight vector processing lanes in the base layer of the AMC are directly connected to 8 GB of DRAM using through-silicon vias (TSVs), providing energy-efficient compute and dramatic improvements in bandwidth and latency over existing memory devices. An AMC also acts as an external memory device and is designed to operate in conjunction with a traditional general-purpose processor. A proposed AMC chip is projected to provide a peak of 320 Gflops/s in the vector lanes balanced with a peak memory bandwidth of 320 GB/s.

Figure 1b shows the breakdown of power consumption on a single modeled AMC chip running a hand-optimized, assembly version of DGEMM. The power-efficient microarchitecture of the processing lanes increases the fraction of power spent on the DGEMM floating-point operations from 14% to 36%.

A major reason for the AMC's energy efficiency is due to the exclusively on-chip communication, resulting in a low energy per bit expended to transfer data from the DRAM layers to the pro-

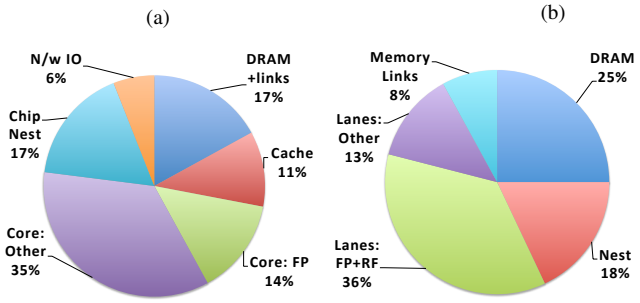


Figure 1: Energy consumption of DGEMM running on (a) Single node of BlueGene/Q at 72 W and (b) Active Memory Cube in 14 nm technology at 9 W.

Table 1: Energy consumption breakdown of DGEMM within a lane.

ALU+VRF	LSU/LSQ	IF/DEC/IS	PCU/PRV	Other
73%	8%	7%	3%	9%

cessing lanes. A second reason is due to a direct path from memory to a large vector register file in lieu of power-hungry data caches. However, this has important ramifications for the programmer, who must now be aware of, and explicitly hide, the latency to DRAM. Similarly, for kernels such as DGEMM that exhibit reuse, the programmer must stage data within the large register files to achieve high performance.

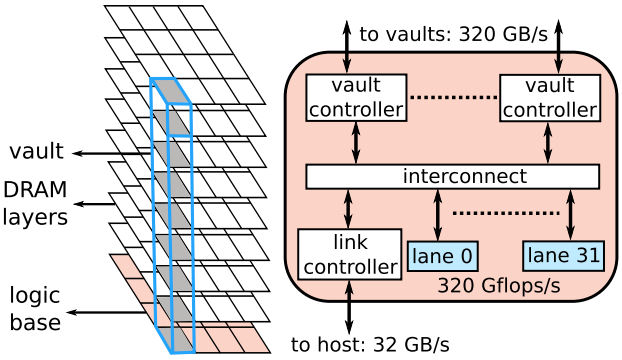
Consider the energy consumption of the various components within an AMC lane as charted in Table 1. A remarkable 73% of total energy is consumed by the arithmetic pipelines and the register files. The instruction fetch and decode units achieve their efficiency due to an instruction set architecture that operates on vectors of up to 32 elements, requiring that programs be vectorized by the user. The lane does not perform dynamic scheduling or issue of instructions in hardware. Instead, the AMC architecture exposes all pipelines and their latencies for software scheduling and for software exploitation of instruction-level parallelism.

The number of ports in the various register files, and hence the power consumed, is contained by evenly distributing all register files across the multiple functional units in a lane. Careful placement of operands in the register files and instructions in the slices is needed to minimize copy operations between slices. The complexity and power of a lane is further reduced by replacing the traditional instruction cache with a software-managed instruction buffer. In cases where a program is a too large to fit within the buffer, the program itself must ensure that the relevant instruction blocks are loaded into the buffer before execution. Finally, the exploitation of the multiple lanes within the AMC is also done in software.

It is not reasonable to expect a programmer to satisfy the above-mentioned software requirements resulting from the need to make hardware energy-efficient. A strong compiler is therefore crucial to effective utilization of the resources in the AMC.

In this paper we make the following contributions.

1. We present the first end-to-end compiler for a stacked processing-in-memory (PIM) heterogeneous system addressing the unique power efficiency tradeoffs of the AMC. Our compiler accepts standard C, C++, and Fortran programs augmented with OpenMP 4.0 accelerator directives rather than specialized languages like CUDA.
2. We show that hiding latency to memory at compile time is paramount for performance in this regime and achievable by our compiler. This is an alternative to a hardware threading model and may be preferred within the context of stacked 3D memory where latency is between 60 to 200 processor cycles. We show that our compiler is able to exploit static data reuse using vector registers alone without the use of traditional caches. By organizing 3D memory within NUMA domains we achieve low power in hardware but our compiler is also able to exploit data affinity for high performance.
3. We report the performance benefits and tradeoffs of compiler optimizations within the novel context of a stacked PIM.
4. Using a power model for the AMC we project that several compute- and memory-bound kernels run in only 9 to 15 watts. Our compiler achieves a power efficiency of 27.4 Gflop/s/W on DGEMM with just 144 watts when simulating 16 AMCs in a node. This is an order-of-magnitude improvement compared to the 2.1 Gflop/s/W on the BlueGene/Q (45 nm technology), about 2.2 Gflop/s/W on the Xeon Phi at 22 nm technology [13], 3.23 Gflop/s/W on the Texas Instruments Keystone II DSP at 28 nm [16], and around 5 Gflop/s/W on GPUs [8].
5. The compiler optimized code is highly efficient, expending 64% to 73% of power in a lane on compute. In contrast, modern out-of-order high performance microprocessor cores consume 50% of total core power in front end units. We also report good scaling of power efficiency as lanes within an AMC are gradually activated.



(a) Logic and memory layers of the AMC (16 lanes and vaults depicted).

(b) Logic base.

Figure 2: The Active Memory Cube.

The rest of the paper is organized as follows. Section 2 gives an overview of the salient features of the AMC and Section 3 lists the specific challenges in compiling for the AMC. Section 4 introduces the OpenMP 4.0 accelerator model used to program lanes within the accelerator and our implementation of it. We then describe in Section 5 the high-level and backend transformations and optimizations developed to produce high quality code given the aforementioned constraints. Section 6 describes experimental results, followed by Related Work in Section 7. We conclude in Section 8.

2. The AMC architecture

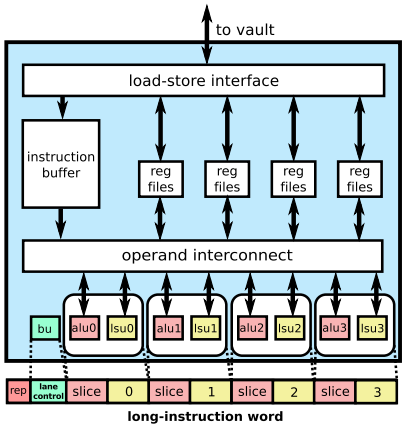


Figure 3: A Vector Processing Lane.

The AMC is a proposed 3D cube composed of 8 GB of DRAM layers stacked on a logic base consisting of a grid of 32 independent compute elements known as *processing lanes* (see Figure 2). The cross section of stacked memory elements on top of one processing lane is termed a *vault* and holds 256 MB of data. A set of eight

neighboring vaults forms a *quadrant*. Any processing lane may access any vault through a high-speed interconnect but the latency is lower if lanes access only vaults within their local quadrant. The lowest latency is achieved when all lanes access only their local vaults. The lanes and DRAM are modeled at 1.25 GHz.

An AMC also serves as an external memory chip through a link that connects directly to the high-speed interconnect. 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.

Memory addresses generated by programs running on the processing lanes are POWER™-architecture effective addresses with standard memory protection facilities. Address translation miss events are handled by the operating system running on the host.

An example compute node in a system would hold a multi-core host processor and up to 16 AMC chips. While a lane in an AMC may access data from other AMCs, this is an expensive operation that involves participation of the host processor. Therefore, the normal mode of operation requires that data be located within the memory of the AMC containing that lane. This is not a significant restriction as typical MPI domain sizes in scientific applications tend to smaller than the 8 GiB capacity of an AMC. Operating system support facilitates data allocation within a particular AMC, a quadrant within an AMC, or a vault within a quadrant.

2.1 Processing Lane

A lane is a vector processor consisting of four *slices* as illustrated in Figure 3 that shares many similarities to scalar, clustered DSPs. A vector instruction serially performs the same operation on every element of its vector operands. Each slice contains an arithmetic unit, a load/store unit, and associated register files. The arithmetic unit has an integer and a double-precision floating-point pipeline. The load/store unit executes update-form memory operations that take a base address and stride as arguments, as well as more powerful scatter/gather operations.

Each slice has sixteen 32-element vector registers, four 32-element mask registers, and thirty-two scalar registers. Mask registers are used to predicate a vector instruction, allowing execution of programs with control flow. Branch instructions are also provided for coarser-grain control flow. A slice's vector registers may be read by the other three slices but access to scalar and mask registers are restricted to the owner slice.

A lane instruction is coded in long-instruction form with sub-instructions, or atoms, designated for the four slices in the lane as well as the single branch unit. Atoms execute in lock step for up to 32 iterations as indicated by the *repeat* field. Branch instructions and scalar instructions on one slice, overlapped with vector instructions on another, execute just once—at the last and first instructions respectively—regardless of the value of the repeat field. Each lane stores up to 512 long-instructions in a lane instruction buffer (LIB).

As noted previously, the functional units fully expose their pipelines to the user; for correct execution, sufficient delays, as determined by the latencies of the functional units, must be inserted in the instruction stream between the issue of 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.

3. Challenges in Compiling for the AMC

The unique features of the AMC make it a challenging target for compilation.

- Programming a heterogeneous system is an onerous task, typically requiring non-standard extensions to specify program offloading and data movement. We desire a programming model for the AMC system that has a low learning curve and requires minimal code modifications. Offloaded sections must also be easily parallelizable across lanes.
- A single lane supports vector execution with programmable length vector operations. Our compiler must exploit this feature for high performance. Support for scatter-gather and predicated execution is also required during vectorization.
- The exposed pipeline architecture requires that the compiler track and optimize hardware resources during scheduling, including latency to memory. The compiler must aggressively schedule code to fully exploit the VLIW capabilities of the lane.
- Scalar and mask register files are not accessible across slices so sharing requires expensive copies. A good mapping of instructions to slices is important to minimize this penalty. This is a challenging problem when the offloaded code consists of a mix of scalar and vector code. We must also deal with a phase-coupling problem since slice mapping, scheduling, and register allocation are interrelated.
- Due to the direct path from DRAM to the lanes it is important for performance that the compiler use the large vector register files to exploit data reuse. Additionally, register spilling is considered expensive and must be minimized. Finally, the bandwidth and latency of accesses vary based on whether a lane references a local or a remote quadrant and so data placement is a primary concern.
- The lane instruction buffer is of a fixed size and so a software instruction cache is necessary to allow execution of arbitrary size programs.

4. Programming Model

Listing 1: DGEMM offloaded and parallelized on the AMC.

```
1  double A[P][R];
2  double B[R][Q];
3  double C[P][Q];
4
5  void main() {
6  // Initialize arrays
7
8  // Copy arrays A, B, and C to AMC 0 if not already resident and
9  // offload loop nest for acceleration
10 #pragma omp target map(to: A[0:P*R], B[0:R*Q]) \
11                      map(tofrom: C[0:P*Q])
12 // Execute iterations of loop i in parallel on 16 lanes of the AMC
13 #pragma omp parallel for num_threads(16)
14 for (int i=0; i<P; i++)
15   for (int j=0; j<Q; j++)
16     for (int k=0; k<R; k++)
17       C[i][j] += A[i][k] * B[k][j]
18
19 // Computed array C is available on the host
20 }
```

We use the recently introduced OpenMP 4.0 accelerator model [18] to offload and parallelize code for the AMC using directives.

The programming model defines a default host device acting as the master and one or more *target* device accelerators that are selectively invoked by the programmer. Each device is associated with an independent data environment, i.e., a collection of variables accessible to it. Variables must be communicated explicitly, or *mapped*, from the host's data environment to that of the target that requires access.

A structured code block that may be offloaded to the target is explicitly identified by the target directive. The AMC compiler supports blocks that may contain serial and parallel code, one or more imperfectly nested loops, and statements that may include simple and SIMD function calls. Since large code sections are handled transparently by a software instruction-cache, the flexibility of the target directive allows the programmer to choose an appropriate region of code for AMC execution so as to amortize the overhead of invoking the accelerator and to increase data reuse within the target section.

The format of the target directive is as follows:

```
#pragma omp target [clauses]
structured block
```

where the optional clauses are of type *device* and *map*.

Device clause. The device clause uses an integer expression to identify a particular AMC in the system as the target for offloading.

Map clause. One or more map clauses specify scalar and array variables that may be accessed by code in the structured block, and an optional direction of communication that is used to optimize data movement. Our compiler automatically maps statically allocated variables referenced within the target region onto the AMC but relies on the programmer to identify indirect accesses.

The OpenMP 4.0 model is designed for both shared and distributed memory environments. On the shared memory AMC architecture, data communication may be eliminated in certain cases: if a program uses processing lanes within a single AMC, and all of its data is already resident within the associated DRAM, then the map clause is ignored by our runtime and there is no overhead for data communication.

Listing 1 shows the matrix-matrix multiply kernel, DGEMM, written using OpenMP 4.0 directives. The nested loop is offloaded onto 16 lanes of the AMC, with iterations of the i loop distributed across the lanes. The compiler generates calls to allocate memory in the AMC's memory space and copy into it the values in arrays A, B, and C. In addition, array C is copied back from the AMC onto the host once the nested loop completes execution. To avoid the overhead of data transfer, the user must explicitly allocate the three arrays in AMC 0's DRAM.

5. A Compiler and Runtime for the AMC

The AMC compiler is built on an industrial-strength C, C++ and Fortran compiler and accepts programs written in any of the three languages. Figure 4 shows its overall structure, highlighting the phases that are significant for generating AMC code.

The compiler frontend processes the input program to mark code regions associated with OpenMP target directives. An outlining and duplication phase creates a new function for each target region. Calls to the runtime are inserted before each target region to acquire one or more AMC lanes.

At this stage the host and AMC procedures follow a different compilation path, one customized for each target, until the respective object codes are linked, along with the OpenMP runtime and the software I-Cache handler, to produce a single fat binary.

In what follows, we highlight specific high-level and backend transformations that help optimize code for the AMC. We only provide an overview, and not the details of the finely-tuned heuristics developed for this accelerator.

5.1 High-level Compiler Transforms

In this section we describe the high-level optimizations used to generate high performance AMC code.

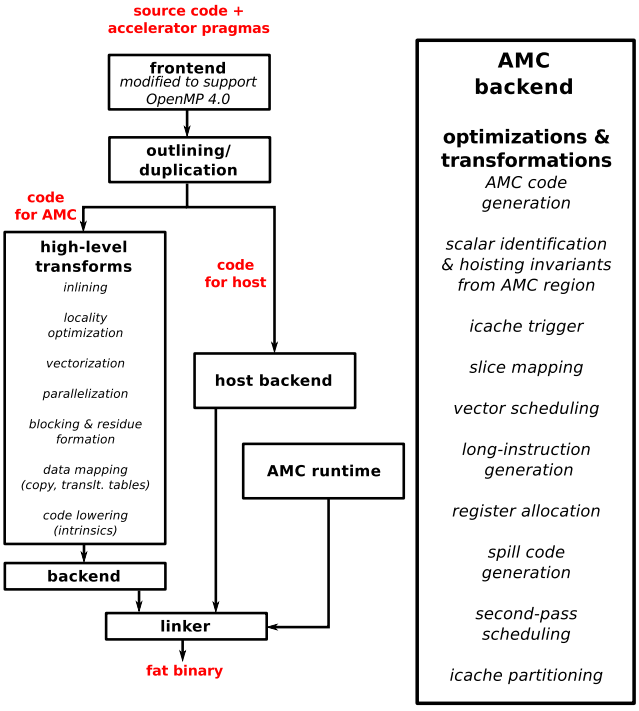


Figure 4: Overview of the compilation steps for the AMC.

5.1.1 Integrated Loop Optimization Using a Polyhedral Framework

The polyhedral framework is a powerful framework to optimize loop nests. In the polyhedral model, loop statements and their data accesses are represented as a polytope and compiler transformations are represented as linear transformations of these polytopes. This representation provides a powerful framework for program analysis and for applying sequences of common transformations, which we use for parallelization and vectorization.

The compiler is capable of analyzing loop nests and program blocks marked with the OpenMP target pragma to discover opportunities for vectorization. This allows vectorization even when a candidate has not been explicitly identified by the user. In a typical case, several loop transformations are applied to vectorize the innermost loop of a nest:

Blocking.
Blocking is important for aggressive scheduling by a software pipeliner. The innermost loop is blocked by the vector length. By default a vector of 32 iterations is selected but if the trip count is known at compile time to not be a multiple of 32, a suitable smaller size is selected to avoid a residue loop, when possible. With a statically known vector length the scheduler is able to generate efficient, tightly packed codes. A less efficient residual loop is generated for residual loop iterations.

Unrolling.
Loop unrolling exposes instruction level parallelism in the loop body and can improve code scheduling especially when the loop body is small. We perform loop unrolling after vectorization and blocking, taking special care not to degrade performance. For example, preserving unit-stride memory references after unrolling achieves best performance on the AMC.

Data access pattern analysis.
We analyze array references using the polyhedral representation to determine whether they are affine, and use this information during code generation. A memory operation is translated into

one of the following types according to its reference pattern in the loop nest:

- **Regular vector load/store.** When a subscript is an affine function of the innermost loop, vector update-forms of the load/store instruction are generated.
- **Gather/scatter.** When the subscript is not affine or constant, a load/store is translated into gather/scatter.
- **Expanded scalar load/store.** When a scalar is privatized in a vectorized loop, the scalar must be expanded so that it has a separate location for each element of a vector. In this case, the scalar is expanded to the innermost loop and later mapped to a vector register. A final store operation correctly writes only the last element of the vector to the original scalar variable.

It is difficult to generate good code by applying each of these transformations independently. One of the challenges is to pass the semantics of one transformation to another. For example, the residual loop after blocking or unrolling has to be marked so that later transformations can ignore it for performance optimization. Also, loop unrolling poses challenges because of the modulo operation in the loop bounds expression.

Our solution is to apply all these transformations together in the last stage of the polyhedral framework, where they operate on the polytope representation of the loop nest and transform it into the intermediate representation used by the subsequent compiler passes. By combining a set of loop transformations in a single translation step, we are able to minimize code size while retaining high quality code in a majority of cases.

Listing 2: Illustration of integrated loop optimization. **VL** is the vector length, **UF**, the unroll factor, and **NS**, the number of stages after software pipelining.

```
1  // Block size is a constant product of the selected vector length
2  // and unroll factor
3  int BLOCK_SIZE = VL * UF;
4  int UB = N >= BLOCK_SIZE * NS ? (N - N % (VL * UF)) : 0;
5  // Main loop, software pipelined for high performance
6  for (int i=0; i<UB; i+= BLOCK_SIZE) {
7    vectorized loop body 0 of length VL;
8    ...
9    vectorized loop body UF-1 of length VL;
10 }
11 // Predicated execution of residual iterations
12 for (int i=UB; i<N; i+=VL) {
13   int P0 = [i,i+1,...,i+VL-1] < N;
14   P0: vectorized loop body;
15 }
```

Integrated loop generation is illustrated in Listing 2. In the transformed loop, there is only one main loop and one residual loop. The new upper bound of the main loop is chosen to be a multiple of the product of the vector length and unroll factor if there are enough iterations for vectorization, unrolling and software pipelining. The remaining iterations are executed in the residual loop.

With statically known constant vector length unrolling and a guaranteed minimum iteration count, aggressive software pipelining may be readily applied. The residual loop on the other hand, may have fewer opportunities for optimization because of its unknown vector length and smaller number of iterations. If iteration counts are known to be small, software pipelining and loop unrolling are disabled and only the residual loop is generated.

When the vector length of the residue loop is unknown at compile time, the scheduler must add no-operation instructions (no-ops) to ensure sufficient delay between producer and consumer instructions resulting in increased code size. An alternative approach that trades minimal performance for code size is to extend the number of vector iterations to 32 and guard the operation with a predicate.

5.1.2 Handling Reductions

Reduction to a scalar variable is handled by first reducing partial results to a vector register in the main loop, before finally reducing the elements of the register to a scalar with a newly generated reduction loop. This allows software pipelining to proceed unaffected.

Reduction to elements in an array is a common calculation pattern in real applications. To correctly execute such reductions in parallel, locks or critical sections are usually used to make sure that every load-add-update sequence is atomically executed. However, introducing this synchronization disables vectorization. An alternative approach is to use additional temporary arrays to store the results in the loop, and then perform owner updates in another loop nest. But this introduces extra load and stores for array elements that may adversely impact performance.

The AMC architecture provides an instruction for performing an atomic memory floating-point store-with-add operation that we use to efficiently implement reductions on array elements. The compiler uses pattern matching to discover these situations; alternatively, user-provided directives could be used to help identify these cases.

5.2 Backend Compiler Transforms

In this section we briefly describe a few of the important backend passes of the AMC compiler.

Once a low-level AMC instruction stream has been generated, the main task is to first schedule, and then perform register allocation. Since a lane encapsulates four identical slice units, there is freedom in placing instructions.. An assignment pass is used to intelligently partition instructions onto slices before scheduling.

Slice assignment places constraints, both in time and space, on the scheduling of instructions. Meanwhile, it is difficult to optimally place an instruction on a particular slice without scheduling and resource usage information from the entire program. Ideally, this phase-coupling problem is solved by whole-program, integrated assignment and scheduling; however, this is computationally expensive. We instead separate the two passes, but introduce a global slice assignment technique that is superior to one with a basic-block-only perspective.

5.2.1 Slice Assignment

While an instruction may be executed on any of the four slices, its placement may increase execution time if it is on the critical path and a predecessor needs to be copied. On the other hand, in some cases, it may be beneficial to copy register values across slices in order to exploit instruction level parallelism (ILP)–this is typically true for high fanout producers. In many cases it may be desirable to introduce copies and increase the length of the static program schedule if it avoids expensive register spills due to overloading of a register file on any particular slice. We designed our slice assignment algorithm to first exploit ILP and then minimize register spills, before reducing slice-to-slice data transfers.

In our first implementation slice assignment was integrated with scheduling at a basic-block scope, but we found it to perform poorly on target sections containing substantial serial codes interspersed with vector regions. A good algorithm must determine the best slice assignment for an instruction taking into account its effect

on predecessors and successors within the same and other basic blocks.

Our globally aware slice-assignment phase maps instructions to slices and inserts any necessary data transfer instructions. This is followed by acyclic code scheduling and software pipelining of loops, which strictly respects the selected assignment.

We build on an efficient and fast multilevel graph partitioning algorithm [11] with estimate-based heuristics to make good partitioning judgements. Since the algorithm considers dependencies across basic blocks, we find it performs well on a mix of scalar and vector code in relatively large programs. Although the slice assignment algorithm is a distinct pass, we find that it rarely inhibits the scheduler from achieving high performance within vectorized loops.

5.2.2 Scheduling

Vector loops are software pipelined for maximum performance while acyclic code is scheduled by a backtracking list scheduler.

We have implemented software pipelining [20] of loops to increase the utilization of the functional units in a lane. Unlike traditional methods, we pipeline successive vector iteration blocks consisting of at most 32 iterations of the original loop. The lanes have an exposed pipeline, and so the scheduler ensures that all the latency requirements of dependent instructions are met at issue time. Architectural constraints enforce specific register file access rules that must also be satisfied to generate a correct program.

We use heuristics based on the Swing Modulo Scheduling algorithm [15], which schedules instructions based on a greedy strategy enforcing resource constraints. Iterative backtracking is used to escape from dead-end states. Heuristics are employed to hide latency to memory, control register pressure, minimize the length of the critical path, and control the amount of instruction-level parallelism that is exploited.

5.2.3 I-Cache Partitioning

To support arbitrary sized target regions within the 512-element LIB, we have designed and implemented a software-based I-Cache [5]. The I-Cache operates by treating the LIB as eight equal sized slots of 64 instructions each. The target region is partitioned into 64-instruction blocks and loaded on demand by a handler resident in slot 0. The I-Cache algorithms and the handler code are carefully tuned to minimize the overhead introduced by the software I-Cache.

Many programs execute in phases: if a loop nest belonging to a phase completely fits within the LIB, the overhead of the I-Cache is low, and this feature allows the execution of large programs on the AMC without involvement of the host processor.

5.3 Data Environment Management

Each lane has an effective-to-real-address translation (ERAT) table with up to eight entries. When memory is accessed on a lane, the hardware references this table and if the access cannot be translated, an exception is thrown to allow the host operating system to install the missing entry. However, this is a high overhead event, and so we prefer to pre-load the ERAT entries ahead-of-time.

The compiler scans AMC code regions and determines variables that are directly accessed. For each variable in the list the compiler uses the static declaration to determine its size and invokes a runtime library call to install the corresponding ERAT entry.

While the compiler can automatically detect direct data accesses in the target construct, it relies on the map clauses specified by the user to (a) identify indirect accesses, (b) direct or indirect accesses in any functions that are called within the target region, and (c) array slices, when only smaller sections of an array need to be accessed within the AMC.

The compiler also processes map clauses and target update directives to record information about the direction of data transfer and inserts runtime calls to perform copies. The runtime in turn checks if the data is already resident within the target AMC and only performs the copy if needed.

5.4 Optimizing Memory References

As with most architectures the memory subsystem requires careful treatment in order to achieve high performance. In this section we describe scheduling techniques, operating system support, and code transformations that optimize access to memory.

5.4.1 Hiding Latency to DRAM

Due to the direct path from DRAM to the register files, all memory references within a lane initiate a request at the load-store unit that is fed into a queue at the lane, traverses an on-chip network, and is 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, and any bank conflicts at the destination vault. Optimizing for each of these factors is critical for high performance.

The scheduler hides latency of an access by enforcing a separation of a predetermined number of cycles between the issue of a load instruction and its first use. Recall that the hardware has interlocks for load dependencies, which stalls the entire lane if a consumer attempts to read a data value that has not yet arrived. By explicitly separating a load from its first use, and by scheduling other instructions in the intervening period, program stalls can be significantly reduced. Software pipelining is well suited for scheduling instructions within this interval.

During scheduling we also track 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.

The compiler also recognizes the vector store-with-add pattern and generates an atomic store-add operation that is processed near the vaults, thus avoiding an extra load operation.

5.4.2 Lowering Memory Latency by Exploiting Data Affinity

The latency of a sequential access to DRAM varies from 50 to 180 cycles or more depending on whether a lane accesses its local or remote quadrant and on the congestion in the interconnect network.

The operating system exports routines to allow memory allocation within a specific quadrant to exploit this characteristic. Our runtime map phase is able to automatically partition one-dimensional arrays onto local quadrants but we rely on the programmer to manually place more complex data structures within the desired regions. Subsequently, the compiler can exploit the lower latency to decrease the scheduled interval between the issue of loads and their first use, often achieving a lower iteration pipelining period, and therefore, lower execution time.

5.4.3 Data Reuse using Vector Register Files

As noted previously, power hungry data caches are replaced on the AMC with large vector register files. Each lane is equipped with 64 thirty-two element vector registers. For loops that exhibit data reuse, it may be possible to exploit the large register file to reduce traffic to memory. Currently we rely on a combination of manual refactoring through standard loop transformations and low-level compiler support for this purpose. These loop transformations can be added to our polyhedral framework with an appropriate cost model.

Listing 3: DGEMM rewritten to exploit reuse.

```
#pragma omp target map(to: A[0:P*48], B[0:48*32]) \
                    map(tofrom: C[0:P*32])
// Parallelize iterations of loop 'i' across 16 lanes and pipeline
// iterations of the loop.
#pragma omp parallel for num_threads(16)
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 registers
    // and reused for the entire duration of the loop nest.
    C[i][j] += A[i][0] * B[0][j] + \
               A[i][1] * B[1][j] + \
               ...
               A[i][47] * B[47][j];
  }
}
```

Table 2: Lines of code in regions offloaded to the AMC.

App	LoC	App	LoC	App	LoC
DAXPY (C)	4	DGEMM (C)	12	DET (C)	58
LULESH (C++)	259	NEKB. (F77)	134		

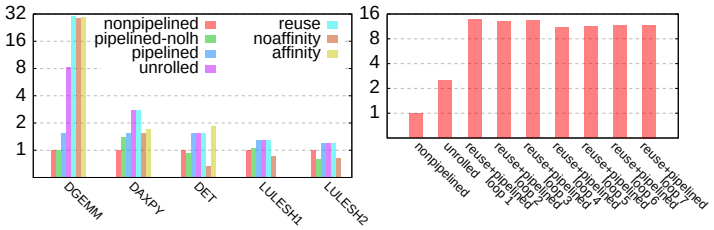
We illustrate the optimization using Listing 3, which is a rewritten version of DGEMM. Loops j and k have been tiled (tile iterators not shown) with tile sizes 32 and 48 respectively. The compiler fully vectorizes loop j and eliminates the control flow. Thereafter, elements of array B are loaded into 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.

6. Experimental Results

We evaluate performance by running compiled binaries on a full system simulator that models a host and the AMC. We use a functional simulator for the host to allow us to simulate an operating system and real programs. The AMC is modeled by a timing accurate simulator that precisely represents lanes, the memory interconnect, vault controllers, and DRAM [3]. Since the AMC's DRAM is also used as the external memory for the host processor there is no data communication overhead for acceleration.

Kernels & Applications. We added OpenMP directives to accelerate codes important to the supercomputing community, focusing on vectorizable codes suitable for our accelerator (Table 2). DAXPY is a memory-bound kernel that stresses the memory subsystem of the AMC. DGEMM is a double-precision matrix-matrix multiply kernel often used to benchmark HPC machines. It tests the ability of the compiler to exploit vector registers for data reuse to achieve high floating-point utilization. DET is a compute-bound kernel that formulates the elemental volume by calculating the determinant of three matrices. It tests the ability of the scheduler to map an odd number of compute strands (three determinant calculations) onto the four slices of a lane.

LULESH [14] and NEKBONE [6] are two applications released by the supercomputing community. We studied the *CalcKinematicsForElems* function in LULESH, which performs a hydrodynamics calculation. It tests the ability of the compiler to vectorize scatter-gather operations and stresses the memory subsystem due to random accesses. NEKBONE is a fluid dynamics Fortran code that solves a Poisson equation using a Conjugate Gradient solver. We focus on the *axi* subroutine since it dominates execution time.



(a) (b)

Figure 5: Flop efficiency improvement relative to unpipelined loop due to compiler optimizations, for (b) NEKBONE, and (a) remaining applications.

The code exhibits reuse but the length of vectors is only 9, exposing alignment issues in traditional SIMD units.

Compiler Optimizations on a Lane. We measure performance as the flop efficiency of a lane, calculated as the number of floating-point operations retired per cycle per lane divided by 8 (four slices times two flops for a multiply-add operation). We first evaluate the effect of various compiler optimizations, applied when possible, including software pipelining, unrolling with pipelining, and exploiting reuse. We then run on 32 lanes with and without exploiting data affinity. Figure 5 reports these results normalized to a non pipelined loop compiled using an optimized backtracking list scheduling algorithm.

Software pipelining universally improves performance 20-50% due to a tighter packing of instructions. Large improvements indicate codes with high instruction-level parallelism. In the case of LULESH, pipelining is only possible because the compiler unrolls small inner loops and inlines all function calls within the offloaded loop. Due to high register pressure we applied manual loop fission to generate two loops. NEKBONE cannot exploit pipelining because the innermost loop has only 9 iterations. Register pressure within DET and LULESH preclude an even tighter packing.

Nevertheless, in all cases observed performance is between 38-95% of predicted performance due to stalling in the LSU (62% of execution time for DAXPY, 10-17% for DET and LULESH, and 4% for DGEMM). Stalling worsens significantly as more lanes are activated. Unrolling with pipelining not only provides more vector instructions that help better hide latency, but additional instruction-level parallelism that can be exploited in the case of DAXPY, DGEMM, and NEKBONE (1.83×, 5.44×, and 2.50× improvement respectively). DET and LULESH were not unrolled because they cause spilling, which degrades performance considerably.

DGEMM exhibits significant improvement in performance when reuse is exploited (3.63× over unrolled) as stress on the memory subsystem is relieved. We performed manual loop distribution and interchange in NEKBONE to exploit reuse and pipelining, with Figure 5b showing improvements of 4.34-5.51× over the unrolled case. These results show that our compiler is able to exploit reuse within vector registers that is traditionally handled within a cache hierarchy on general-purpose machines.

Performance when all lanes are active and data affinity is not exploited shows a slight drop in performance for DGEMM (95% of the best single lane version) and significant deterioration for the others (43-67%). Exploiting affinity, when possible, helps negate some of this loss. DET actually achieves higher performance than the single lane case, indicating that affinity may be beneficial even with one lane active.

Scaling Across Lanes. Next, we study scaling on the 32 lanes of the AMC, charting flop efficiency and the average load latency observed at a lane, as well as speedup relative to one lane in Figure 6. We keep the per lane workload constant (weak scaling),

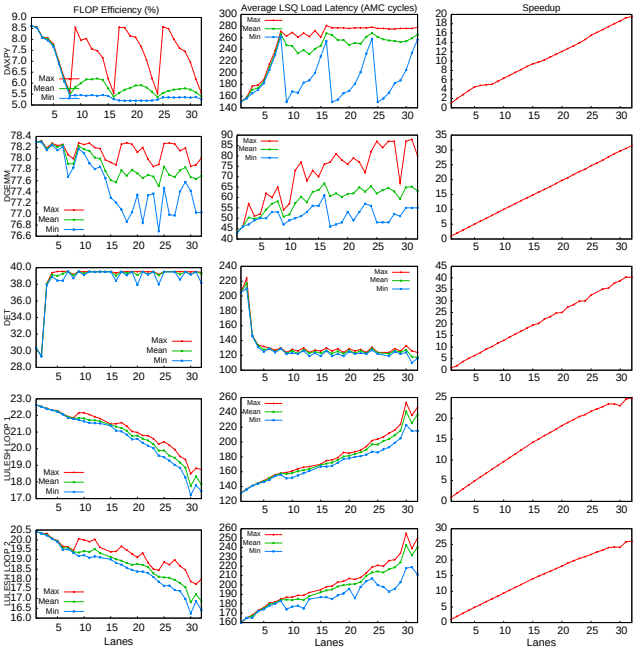


Figure 6: Performance curves for the kernels on the AMC.

except for LULESH, where the total work is kept constant (strong scaling).

DAXPY. We achieve excellent scaling on up to five lanes ($4.45\times$) but then speedup tapers off and is limited to $5\times$ on eight lanes. Note that we are exploiting quadrant affinity so the kernel is restricted to the memory bandwidth of only eight vaults. DAXPY is a memory bound kernel so we see high bandwidth pressure at the vaults, resulting in an 80% increase in load latency with eight lanes as compared to one. However, the advantage of quadrant affinity is that we can achieve the same behavior on the next quadrant of eight lanes, i.e., excellent initial scaling followed by a tapering off. We achieve a speedup of $9.8\times$ with sixteen lanes and $19.5\times$ when all lanes on the AMC are active. Extrapolated to sixteen AMCs in a node, we achieve a read and write bandwidth of 2.1 TB/s and 1.0 TB/s respectively, nearly $8\times$ than achieved if using the node's host processors alone.

DGEMM. This kernel is compute-bound and exerts very little pressure on the memory subsystem after exploiting data reuse. Therefore when exploiting quadrant affinity, the eight vaults per quadrant easily satisfy the bandwidth requirements of DGEMM. Speedup with eight lanes is $7.95\times$ and with 32 lanes, $31.48\times$. This is attributed to the excellent schedule, which is both able to tightly pack the ALU instructions across the floating point units of the four slices in a lane while also hiding memory latency. We achieve a flop efficiency of 77%, which compares favorably to the 83% of peak performance achieved by careful hand optimization.

A node is expected to sustain 3.9 teraflop/s, and at 9 W per AMC (refer Figure 8), a power efficiency of 27.4 Gflop/s/W without accounting for power consumed by the host.

DET. This kernel is neither memory-bound like DAXPY nor compute-bound like DGEMM but is somewhere in between (3:1 ALU to LSU ratio). As mentioned earlier, there are only three determinant computations, however, our scheduler is able to efficiently pipeline the loop across all four slices. Register pressure heuristics while scheduling are critical for DET so that register use is balanced across slices and a less aggressive ALU utilization is selected to avoid spilling.

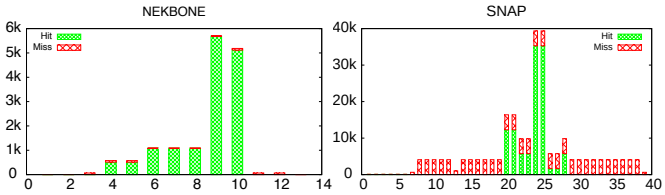


Figure 7: Software I-Cache behavior of two applications.

We see linear speedup for all lanes, made possible because we are able to exploit affinity, as otherwise the inter-quadrant links become a performance bottleneck leading to longer observed load latencies and stalling at the lanes.

LULESH. By pipelining two iterations in both cases, the compiler achieves a flop efficiency of 27.2% and 22.5% on a single lane despite spilling 12 and 8 vector registers in the two loops respectively.

Figure 6 shows strong scaling for LULESH compiled for multiple lanes using a single pipeline stage. Performance is affected by the AMC wide random accesses of the gather operations. The compiler can only hide a load latency of 192 cycles (size of LSQ), so as the observed latency increases beyond this value, performance drops appreciably. Nevertheless, a speedup close to $25\times$ is achieved with 32 lanes.

NEKBONE. We present initial results for NEKBONE on a single lane. Although only nine of the 32 elements of vector registers can be used, it presents no issues for vectorization or code generation (unlike SIMD). The compiler exploits the iteration counter to only issue nine elements for vector operations, producing efficient code. We observe a flop efficiency between 13.2% and 16.8% for the seven loops.

Software I-Cache Analysis. LULESH and NEKBONE are excellent demonstrations of the utility of the software I-Cache. Figure 7 shows the I-Cache behavior of NEKBONE (labelled *unrolled* in Figure 5b). Recall that an AMC program that is too large to completely fit within the LIB is partitioned into blocks of 64 lane instructions. The compiler generates 13 such blocks for NEKBONE. The graph shows the number of invocations of each of the blocks, where a block invocation occurs whenever there is a block-to-block control flow event. Hit and miss events are also marked. We measure cache performance at the block instead of the instruction granularity since a miss event is an inter-block control flow that requires the loading of an entire 64-instruction target block.

As shown in Figure 7, although the offloaded section of NEKBONE overflows the LIB, the loop nest is packed into seven blocks, each of which maps to a distinct location in the LIB, achieving a greater than 95% hit rate and an overhead of only 3.08% of execution time. Similarly, LULESH shows negligible I-Cache overhead because the "hot" loop easily fits within the LIB and the handler is only activated to load the residual loop.

Another application we studied is SNAP, which has 41 blocks within the offloaded loop, all of which are accessed in every iteration. With only seven slots in the LIB, blocks have to be loaded repeatedly by the handler, resulting in a high I-Cache overhead of 44.76%. One way to diminish the overhead is to explore ways to distribute the outer loop to limit the number of blocks within each loop. Another option is to prefetch instruction blocks so as to hide the cost of a miss. A more involved approach is to split a program across multiple lanes and to execute it in a streaming manner.

In summary, the advantage of the software I-Cache is that costly host invocations are avoided. It can also enable aggressive software pipelining, which increases code size due to the prologue and

epilogue, but can generate a highly performant kernel that can still fit within the LIB.

Power Analysis. To gauge the energy efficiency of the AMC we provide an evaluation of the power for the four kernels. The power of the AMC was evaluated for a 14 nm SOI FinFET technology node with a base chip frequency of 1.25 GHz and a C4 Vdd voltage of 0.656 V. The voltage of the DRAM stack was 1.0 V. The base chip timing was modeled at 1.5 sigma slow, with added frequency and voltage guard bands to account for worst case clock jitter, power grid IR drop and noise, and product lifetime. The power model includes lanes, I/O links, on-chip interconnect network, vault controllers and the DRAM stack. Technology scaling assumptions used by actual products were used to derive power and a target of 90% installed clock gating was assumed for the lanes, leaving 10% of the AC power non-gateable. The AMC power model is based on performance statistics from the AMC simulator with a set of utilization equations representing the activity behavior of each AMC component. Power gating of the compute lanes was applied whenever the lanes were not active. I/O links were assumed gated throughout the AMC parts of the execution.

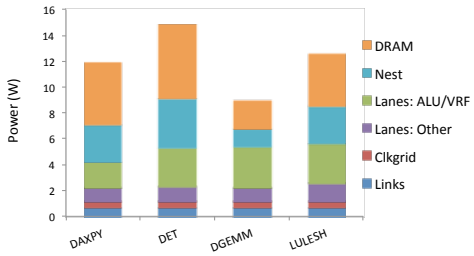


Figure 8: Power consumption of the studied kernels within the entire AMC.

The power breakdown of the AMC when running the kernels with 32 lanes is illustrated in Figure 8. DGEMM exhibits very good reuse of the data in the register files thus requiring a low memory bandwidth. This results in a full 47% of the AMC power being spent to perform useful computation in the lanes, as opposed to transferring data to and from memory. Due to the higher memory bandwidth used by the other kernels, the nest and DRAM consume a larger fraction of the total AMC power but a healthy 26% to 36% of the power is still being spent for useful computation in the lanes.

The AMC compute lane was architected for power efficiency. Traditional processor cores consume significant power in the instruction fetch, decode, issue and branch units. The vector nature of the AMC compute lane allows these units to be active as few as one out of every 32 cycles, significantly reducing the AC power. This fact is illustrated by the relative compute lane power breakdowns of Figure 9. The majority of the lane power, as much as 81%, is spent performing effective computation and data transactions in the ALU, VRF and LSU part of the lane while as little as 7% of the power is spent in the instruction fetch, decode, issue and branch part of the pipeline. This is in stark contrast to modern out-of-order high performance microprocessor cores where such front end units can consume over 50% of the total core power.

Figure 10 illustrates how the Gflop/s/W efficiency scales with the active lanes. Efficiency improves superlinearly for DAXPY, DET and DGEMM. This is due to the amortization of the constant power fraction of the AMC, such as I/O link power, leakage and non-gateable AC power, over the increased flops provided by the larger number of active lanes. Only LULESH displays a slight reduction in flops at higher lane counts resulting in a more linear scaling of Gflop/s/W. The fact that the Gflop/s/W efficiency for most of

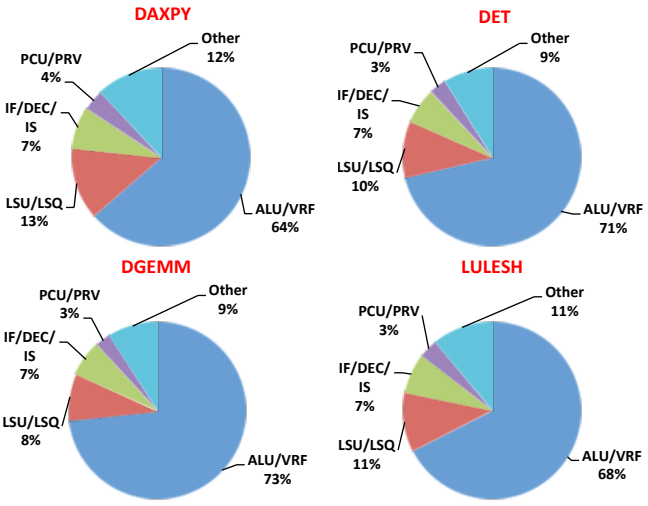


Figure 9: Breakdown of power consumed within a lane.

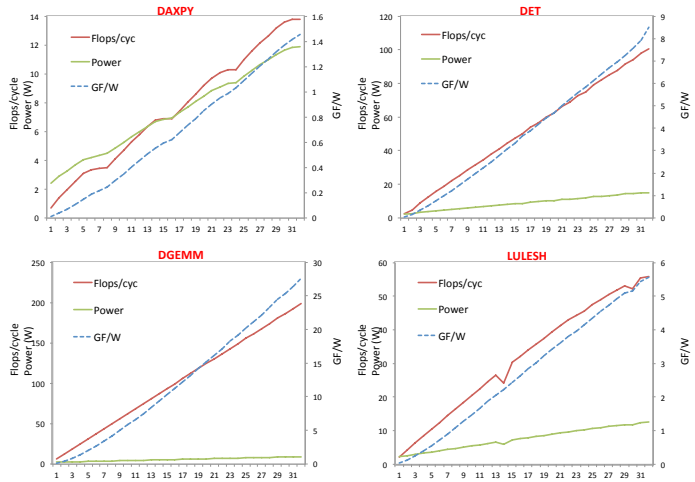


Figure 10: Superlinear scaling of performance per watt with number of active lanes.

the applications continues to increase without noticeable slackening, which is not typical in other compute architectures, demonstrates the effectiveness of the high bandwidth AMC architecture and also the ability to exploit data affinity in local quadrants.

7. Related Work

3D PIM. The idea of using processing capability in the base logic layer of a stack such as the HMC has recently been investigated [23, 24]. They incorporate cores of existing processor architectures in the base layer and hence cannot approach the energy-efficiency of the AMC. Standard compiler techniques suffice in these cases. The AMC's lane architecture is more similar to a DSP, specifically designed with the energy-efficiency goal of Exascale computing. As far as we are aware, ours is the first to explore novel architecture/-compiler tradeoffs within this context.

OpenMP 4.0. Since the recent introduction of this accelerator model [18] implementations have been released for the GPU [4], Xeon Phi, and the Keystone II DSP [16]. Our work provides evidence on the suitability of this platform independent approach for a non-traditional microarchitecture. A key distinction is the freedom

of our runtime to execute a target region on the host if the AMC is busy. This is possible because data is shared and our compiler generates optimized code for both targets.

Vector VLIW scheduling. Several approaches to software pipelining for DSPs [15, 20] target only scalar instructions; we target both scalar and vector operations, including variable length vectors. The challenge is to schedule the mix across functional units without an explosion in generated lane instructions. We also use aggressive register pressure heuristics to prevent spilling at all costs.

Latency hiding. Variable memory latencies due to caches have been a particular challenge to static VLIW scheduling. Various solutions use hardware threads or prefetch units [22]. Some architectures provide large register files and deep LSQs to hide worst-case latency. The AMC avoids this to improve energy efficiency. We show that techniques in our software pipelining stage can manage LSQ capacity and hide latency effectively.

Data reuse. Loop transformations to exploit data reuse through register tiling [12] can be applied to the AMC's vector register file. In this work we have shown how a subsequent software pipelining stage can exploit tiled code for high performance.

Slice mapping. Instruction mapping onto clustered functional units of DSPs has been studied in the context of cyclic [17] and acyclic codes [19]. Previous techniques integrate mapping with scheduling [2, 17] due to their phase coupling relation. We consider this prohibitively expensive. Instead, we use two distinct passes with a global graph partitioning based mapping similar to Aletà [1] extended for multiple basic blocks. We achieve superior mapping of an offloaded region containing scalar, vectorized and residue loops.

8. Conclusions

The Active Memory Cube implements a processing-in-memory architecture designed for Exascale computing that gains power efficiency by moving computation to data. It is strongly influenced by principles of embedded design, achieving power efficiency using a microarchitecture that eliminates much of the complexity of conventional processors and instead relies on sophisticated compiler, runtime, and support software to deliver performance.

In this work we have described a compiler that uses the OpenMP accelerator model to offload and parallelize programs on AMC lanes. Our compiler exploits VLIW and vector capabilities, scatter/gather, predication, fully utilizes functional units via software pipelining, performs global slice assignment, hides memory latency and exploits data reuse through vector registers, exploits data affinity, and transparently handles codes of arbitrary size. Our experiments show high computational efficiency, linear performance scaling, and superlinear performance per watt scaling on memory- and compute-bound kernels. We are able to achieve these results using standard, portable pragmas and no accelerator-specific code.

The ideas described in this work can be extended to an AMC for an embedded application incorporating a simple host processor on the logic layer. A device delivering hundreds of Gflops/s for 10 W or less is possible based on extrapolation of our results.

Acknowledgments

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

References

[1] A. Aletà et al. Graph-partitioning based instruction scheduling for clustered processors. In *Symp. Micro.*, pages 150–159, 2001.

[2] A. Aletà et al. AGAMOS: A graph-based approach to modulo scheduling for clustered microarchitectures. *IEEE Trans. Comput.*, 58(6): 770–783, June 2009.

[3] Anonymized. Active memory cube: A processing-in-memory architecture for exascale systems, 2015. Submitted for publication.

[4] L. Chunhua et al. Early experiences with the OpenMP accelerator model. In *Workshop on OpenMP*, pages 84–98, 2013.

[5] A. E. Eichenberger et al. Optimizing compiler for the CELL processor. In *PACT*, pages 161–172, 2005.

[6] P. Fischer and K. Heisey. *Nekbone 3.0*, 2013. URL `https://cesar.mcs.anl.gov/content/software/thermal_hydraulics`.

[7] A. Gara et al. Overview of the Blue Gene/L system architecture. *IBM J. Res. Dev.*, 49(2):195–212, Mar. 2005.

[8] R. Garg and L. Hendren. A portable and high-performance general matrix-multiply (GEMM) library for GPUs and single-chip CPU/GPU systems. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 672–680, 2014.

[9] R. Haring et al. The IBM Blue Gene/Q compute chip. *Micro, IEEE*, 32(2):48–60, 2012.

[10] J. Jeddeloh and B. Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *Symposium on VLSI Technology*, pages 87–88, 2012.

[11] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *ACM/IEEE Conference on Supercomputing*, 1995.

[12] R. Lakshminarayanan et al. Compact multi-dimensional kernel extraction for register tiling. In *Conference on High Performance Computing Networking, Storage and Analysis*, pages 45:1–45:12, 2009.

[13] B. Li et al. The power-performance tradeoffs of the Intel Xeon Phi on HPC applications. In *Parallel and Distributed Processing*, 2014.

[14] LLNL. Hydrodynamics Challenge Problem. Technical Report LLNL-TR-490254.

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

[16] G. Mitra, E. Stotzer, A. Jayaraj, and A. P. Rendell. Implementation and optimization of the OpenMP accelerator model for the TI Keystone II architecture. In *Workshop on OpenMP*, pages 202–214. 2014.

[17] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Symp. Micro.*, pages 103–114, 1998.

[18] OpenMP ARB. OpenMP version 4.0, May 2013.

[19] V. Porpodas and M. Cintra. CAeSaR: Unified cluster-assignment scheduling and communication reuse for clustered VLIW processors. In *Compilers, Architecture and Synthesis for Embedded Systems*, Sept 2013.

[20] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Symp. on Microarchitecture*, pages 63–74, 1994.

[21] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *International Conference on High Performance Computing for Computational Science*, pages 1–25, 2011.

[22] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, June 2000.

[23] D. H. Woo et al. An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *International Symposium on High Performance Computer Architecture*, 2010.

[24] D. Zhang et al. TOP-PIM: Throughput-oriented programmable processing in memory. In *International Symposium on High-performance Parallel and Distributed Computing*, pages 85–98, 2014.