RC25645 (WAT1612-010) December 5, 2016
Computer Science

IBM Research Report

Scheduling for Clustered Vector Processors Near Memory

Arpith C. Jacob, Zehra Sura, Tong Chen, Carlo Bertolli, Samuel Antao, Olivier Sallenave, Kevin O'Brien, 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

Scheduling for Clustered Vector Processors Near Memory

Arpith C. Jacob Zehra Sura Tong Chen Carlo Bertolli Samuel Antao Olivier Sallenave
Kevin O' Brien 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,nair,brunhe,jacobp,rosnbrg,yoonho,alexe,kimchang}@us.ibm.com

Abstract

The Active Memory Cube is a processing-in-memory device that achieves high power efficiency with a carefully designed microarchitecture eliminating much of the complexity of conventional cores. To deliver high performance it requires a sophisticated compiler, which we present in this work.

We propose a novel clustering algorithm for distributed functional units and a scheduling algorithm for temporal vector operations within the cluster. We propose an abstract machine model that precisely captures vector functional units, vector register file sharing at the element level, communication between clusters, and interaction between the core and the memory subsystem. Unlike prior approaches our clustering algorithm operates at the scope of an entire procedure to make globally optimal assignments.

We achieve high computational efficiency and linear performance scaling on memory- and compute-bound kernels using standard, portable pragmas and no accelerator-specific program code. Our work is an important step toward building next-generation, power-efficient computing systems.

1. Introduction

Crossing the Exascale threshold is proving to be challenging. Dennard scaling can no longer provide improvements in clock frequency at a constant power density. To achieve continued performance improvements it has become necessary to tackle the inefficiencies of general-purpose systems.

We analyzed the power consumption of a BlueGene/Q node running an optimized version of double precision matrix-matrix multiply (DGEMM). Just 14% of total energy is spent executing floating-point operations. Overheads include energy spent within integer pipelines, fetch and decode units, and the issue logic. Access to external memory and on-chip caches account for 45% of total consumption.

The *Active Memory Cube* (AMC) [13] is a processing-in-memory accelerator designed for low power consumption that places light-weight vector processing lanes directly under a 3D stack of DRAM. Nair et al [13] report that a hand-optimized version of DGEMM is expected to consume just 10 W on one AMC.

The fraction of power spent on floating-point operations in an AMC's lanes increases from 14% to 36%. One reason for the AMC's energy efficiency is due to the on-chip, direct path from memory to a vector register file in lieu of power-hungry caches. However, the resulting tradeoff is that a programmer is exposed to the latency of a DRAM access.

We also studied the energy consumption of the various components within a lane. About 73% of total energy is consumed by the arithmetic pipelines and the register files. Instruction fetch and decode units achieve high power efficiency due to an instruction set architecture that operates on vectors of up to 32 elements. The lane does not perform dynamic scheduling or issue of instructions in hardware instead exposing all pipelines to the programmer. A lane's register files are partitioned and distributed across the functional units into clusters to improve power and performance characteristics. Cross cluster communication requires explicit copy instructions via a low-bandwidth bus. Therefore, a programmer must balance instructions across clusters while minimizing register transfers in the critical path.

The AMC's efficiency is achieved by shifting many of the typical processor functions that induce overhead to the programmer. While several hand-coded kernels have been demonstrated on the AMC [13], in this work we investigate a compiler for the AMC. In particular, we tackle the scheduling challenges due to the near-memory organization and the various computing paradigms exhibited in the microarchitecture. We make the following contributions:

1. Functional units within a lane operate element-wise on vectors, whereas a lane is a VLIW core. This interaction of the vector personality with the VLIW nature leads to a unique scheduling challenge. In contrast to SIMD units and many embedded vector processors, the AMC's vector operations are temporal, not spatial. The scheduler must

issue a mix of single-issue scalar and multiple-issue vector instructions to ensure high utilization of the functional units. We first introduce an abstract machine model for such a vector-VLIW core that also captures the clustered organization of functional units. The memory subsystem in the context of near-memory processing with a direct path from core to memory is modeled. We use this model in a Modulo Scheduling algorithm to exploit vector instruction level parallelism through software pipelining. As far as we are aware this is the first work to apply Modulo Scheduling in the context of a temporal-vector-VLIW core.

2. Existing clustering algorithms operate on a per basic block basis [1–3, 14] and make locally optimal decisions. On a procedure with scalar, vector and residue code, local clustering leads to poor schedules or increased register spills, which is detrimental for performance on a processor near memory. We propose a clustering algorithm that can distinguish load gain on individual basic blocks, enabling us to make globally optimal decisions.

The rest of the paper is organized as follows. We summarize the AMC hardware in Section 2 with background on our compiler in Section 3. Section 4 introduces the abstract machine model we use for clustering and scheduling. Our novel global clustering algorithm is introduced in Section 5 and the Modulo Scheduling algorithm is described in Section 6. Section 7 describes experimental results with related work in Section 8. Section 9 concludes with a discussion on optimizations for a near memory processor.

2. The Active Memory Cube

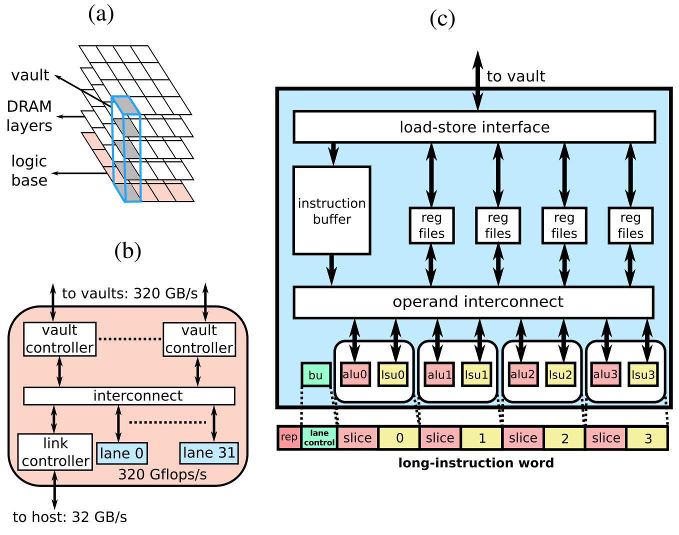

Figure 1: The Active Memory Cube: (a) Logic and memory layers (16 lanes and vaults depicted), (b) Logic base, (c) Lane microarchitecture.

The AMC [13] is a proposed near memory processor that stacks 8 GB of DRAM layers on a logic base of 32 *processing lanes* (see Figure 1a). A cross section of stacked memory elements on top of one lane is termed a *vault* and a group of eight neighboring vaults form a *quadrant*. Any processing lane may access any vault through an interconnect (Figure 1b), but the latency of access is lower if lanes access vaults within their local quadrant. Operating system support facilitates data allocation within a particular quadrant [19].

Each AMC also serves as an external memory chip and memory is kept coherent with the host's data caches. This shared memory architecture greatly simplifies and improves efficiency of communication between the two compute chips. We refer the reader to the literature for complete details on the hardware [13].

2.1 Processing Lane

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

A lane instruction is coded in long-instruction form (VLIW) with sub-instructions designated for the four slices and the branch unit. A lane instruction holds an iteration count (repeat field) that specifies the number of vector elements to be processed by each slice operation. The lane stores up to 512 long-instructions in a lane instruction buffer.

The functional units on a lane fully expose their pipelines to the compiler; hence, sufficient delays must be inserted in the instruction stream to avoid data and structural hazards. 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.

A lane's vector, scalar, and mask register files are clustered within slices to improve performance and power characteristics. Each slice has 16 32-element vector registers, 4 32-element mask registers, and 32 scalar registers.

Scalar and mask register files are partitioned and distributed across slices. Within a slice, functional units have high bandwidth access to their register files while communication across slices is with copy instructions via a low-bandwidth result bus. While the vector register file is also partitioned across slices, the four ports of each register file may be directly accessed by any slice through an interconnect. Simultaneous access to the same vector register by multiple slices is permitted, including on the same port, provided they access the same vector element.

3. The AMC Compiler

The AMC compiler is built on an industrial-strength C, C++ and Fortran compiler. The compiler offloads structured blocks identified by an OpenMP 4.0 [15] *target* pragma to the AMC. Standard OpenMP worksharing constructs are used to parallelize sequences of loops on AMC lanes. The host and AMC procedures follow a compilation path customized for each device and the respective object codes are linked to produce a single fat binary.

A cluster assignment pass maps every instruction in a procedure to a slice. This is followed by instruction scheduling. In the following sections we first introduce a machine model for a processing lane. We then propose our clustering and modulo scheduling algorithms.

4. Machine Model

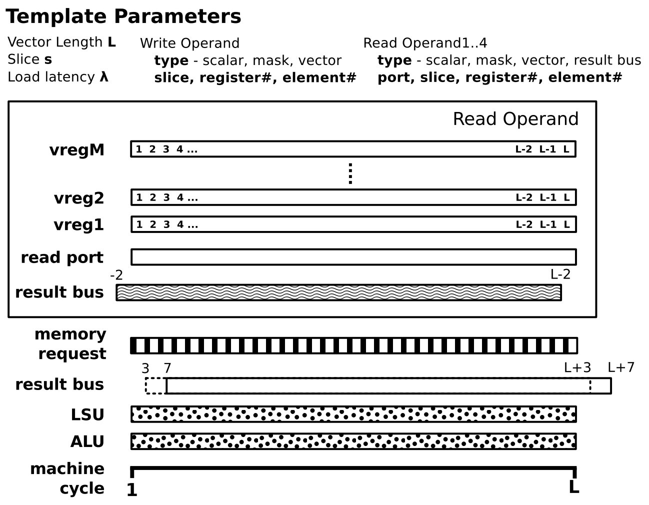

Figure 2: A machine model for a vector-VLIW core.

Before we describe the software pipelining optimization we first introduce a machine model that captures the essence of a vector-VLIW core without being tied down to its implementation. An abstract but high fidelity model allows us to precisely define and efficiently solve the scheduling problem such that its solution is feasible and maximizes performance in the real hardware. The model captures both the scalar and vector personality of a slice, the clustered organization of a lane, the various functional units in a slice, and the partitioned vector register file with its shared read ports. The essence of the memory subsystem in the context of near-memory processing with a direct path from core to memory is also modeled. Our model is not limited to the AMC and the concepts we introduce are transferrable to other microarchitectures that apply similar computing paradigms.

The machine model, depicted in Figure 2, defines a parameterized template for each instruction that describes its effect over time on the microarchitecture. An instruction template is parameterized by a positive integral vector length L (a value of 1 represents a scalar instruction), the slice s onto which it is mapped, the load latency λ if a load instruction, and up to four read and one write operand.

We use a two dimensional table to represent resources accessed in time offset from an instruction's first issue cycle. Time is modeled at the granularity of an individual machine cycle rather than a coarser unit such as a vector length of cycles. This higher fidelity is computationally expensive but also more expressive, allowing the model to represent both scalar and vector instructions.

Functional Units. The template models the effect of the fully pipelined Arithmetic, Load/Store, and Branch units by reserving them for one cycle per vector element on the slice specified by the template parameter s. We define the *load* of an instruction I using the functional unit I_f as its vector length I_L, i.e., the number of cycles the instruction occupies a particular functional unit.

Register Files. On most modern compilers scheduling is a distinct pass prior to register allocation. Scheduling typically operates on instructions in Static Single Assignment (SSA) form that define and access virtual registers. Therefore our model for register files assumes infinite capacity, with an individual register specified by a tuple whose elements are the register type, slice, and a non-negative integer identifier. In addition, we use a read pointer for vector register accesses to identify an element within a vector that is being read.

An instruction writes to a shared result bus on slice s 3 and 7 cycles after issue to the Integer and Floating point pipelines respectively. Modeling this write result bus avoids collisions between integer and floating-point instructions. Note that the exact register including the elements of any vector register defined by the instruction is recorded on the result bus. The reason for this will be evident shortly.

Read Operands. A read from a register file is modeled by recording the register accessed at the specific read port every cycle. This allows the concurrent scheduling of instructions that read the same register via the same port. In addition, the constraints of a vector register access require careful modeling. Recall that a vector register may be concurrently accessed from any one of the four slices via any one of a register file's four ports so long as they read the same vector element. To model this constraint we represent a vector register file of a sufficiently large size M with a corresponding number of rows in the resource table. Each slot in the row records the vector element accessed at a particular machine cycle, which is ordered sequentially from $1 \ldots L$ starting with the first issue cycle. To check for a feasible schedule, it is sufficient to ensure that a slot of a vector register is non conflicting with concurrently scheduled templates.

An operand may bypass the register file to directly read the output of a dependent instruction via its slice's result bus. The model records the operand register accessed in a read result bus row of the resource table. To activate the bypass, the read result bus of the template must align exactly in time and match elements on the write result bus of the dependent instruction's template.

4.1 Modeling the Memory Subsystem

The direct path from near-memory cores to DRAM exposes the latency of a memory access to the scheduler. Unless load latency is hidden during scheduling, a program will spend most of its execution time stalled on memory. To accurately represent a near-memory processor we extend our machine model to include the memory subsystem.

The latency of an access depends on the quadrant where data is stored and on whether a load is sequential or random. In *Quadrant wide* mode lanes access data at low latency from within vaults of their quadrant, while in *AMC wide* mode a lane accesses data at higher latency from all vaults. In this section we characterize the latency of load operations. The scheduler may use this information to hide latency of an access by exploiting both the vector issue and instruction level parallelism to maximize bandwidth efficiency.

Characterizing Latency. We design a microbenchmark that issues one load per cycle and uses the result d cycles after issue in order to measure the load-to-use latency. Our strategy consists of inserting a series of loads to hide the latency to first use of the data, increasing the number of inserted loads until there are no execution stalls.

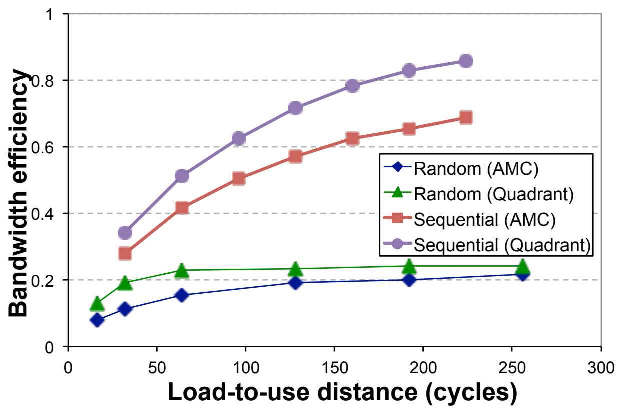

Figure 3: Bandwidth utilization for linear 32 byte and random 8 byte accesses as a function of load-to-use distance. All 32 slices are active.

When the load-to-use distance d is small (32-64 cycles) load latency cannot be hidden, resulting in stalls on the lane and under utilized bandwidth (Figure 3). As d increases there are more requests in the system to hide the latency and the utilization of bandwidth increases. However, as bandwidth utilization increases across lanes there are more conflicts in the memory interconnect and this makes it difficult to approach peak bandwidth. This is evident from the curves for the sequential access pattern where hiding 160 and 192 cycles achieves about 80% and 60% of peak bandwidth for the Quadrant-wide and AMC-wide cases respectively. This is close to the maximum achievable bandwidth given the overheads in the memory controller. In the case of the random access pattern we reach saturation at a lower load-to-use distance of 128 cycles.

We can use the results of this experiment to generate better schedules. First, to achieve high bandwidth utilization the scheduler must separate a load from its first use by the expected latency of access and issue independent instructions during this interval. It is difficult to hide the 100 cycle latency using scalar instructions alone. We achieve this through the temporal issue of up to 32 iterations of a vector operation and by exploiting vector instruction level parallelism through Modulo Scheduling. Second, the maximum achievable bandwidth is attained by treating linear load and gather instructions as having access latencies (parameter λ of the model) between 128 and 192 cycles.

Modeling the Memory Interface. Table 1 shows the execution profile of vectorized, hand-optimized *Determinant* kernel with loads scheduled as early as possible to separate them from their first use. The lane is stalled a quarter of the execution time on a full load/store queue (LSQ) due to eager scheduling of up to four memory operations per cycle.

Lane Execution	LSQ Full Stall	Memory Stall
67.95%	24.37%	7.68%

Table 1: Execution profile of Determinant on one lane.

Slices may issue up to four memory requests a cycle but a lane can only service one per cycle. Moreover, a vector LSU instruction such as a scatter issues up to 32 requests. These are buffered in a 192 entry LSQ until satisfied by the AMC's memory subsystem. An LSU instruction stalls the lane if it is issued when the LSQ is full.

A simple yet effective way to model and control the behavior of the lane's memory subsystem is to represent the number of memory requests initiated from a lane in a row of the reservation table. For a memory bound kernel a scheduler may issue at most one per cycle while for more compute bound kernels up to four may be issued per cycle. This approach can exploit the LSQ as a buffer for the variable latencies of load requests as long as the average latency is below 192 cycles.

5. Global Cluster Assignment

In this section we introduce our novel clustering algorithm. We decouple slice assignment from scheduling to achieve *globally* optimized assignments. After slice assignment, data transfer instructions are inserted as required and the scheduler respects the preselected assignment.

There are two key features of our algorithm. Slice assignment examines an entire procedure to make globally optimal decisions when mapping instructions to slices, i.e., data transfers are minimized across the entire procedure, even for producers and consumers that span multiple basic blocks. On the other hand, load across slices is balanced on a basic block basis because this is the granularity of our scheduler.

5.1 Multilevel Graph Partitioning

Our slice partitioning algorithm is based on the efficient and effective multilevel strategy [8] and operates on the program dataflow graph. In a *coarsening* step pairs of related nodes, for example, a producer of a scalar or mask value and its consumer, are coalesced together to treat them as a single unit for the purposes of a subsequent partitioning step. Coalesced nodes are replaced by a macro node, edges incident to the pair of nodes are now made incident to the macro node, and a new multigraph is formed. Coarsening continues on the newly derived graph until we are left with a last-level graph of at most four nodes.

In the *refinement* step each of the four nodes of the last-level graph is assigned a distinct partition (one for each slice) and this information is propagated to nodes in the graph at the next lower level. At this stage a refinement process perturbs the initial slice assignment, moving macro nodes to other slices in an effort to minimize data transfers while balancing load, until there is convergence to an optimal partitioning. Refinement continues through the subsequent levels of graphs until the original dataflow graph is partitioned.

Since the algorithm treats globally related nodes as a unit it is able to consider mappings that may be sub-optimal within a basic block but beneficial for the entire procedure.

5.2 Node and Edge Annotations

To evaluate the costs and benefits of various partitions we annotate nodes and edges of the program dataflow graph.

5.2.1 Node Weight

A (macro) node is annotated with a weight that reflects the functional units used by its component operations on a particular slice. The weight of a node θ is a two dimensional variable $W_\theta(1 \ldots n_b, 1 \ldots n_f)$, where n_b is the number of basic blocks in the procedure and n_f the number of functional units.

If a node is a single instruction I in basic block I_b using the functional unit I_f, its weight is the instruction load as defined in the machine model, i.e., its vector length I_L.

$$W_I(b,f) = \begin{cases} I_L & \text{if } b = I_b \wedge f = I_f, \\ 0 & \text{otherwise.} \end{cases}$$

After coalescing a pair of nodes, a macro node contains multiple instructions from arbitrary basic blocks. Its weight is calculated recursively as $W_\theta(b,f) = \sum_{\theta' \in \theta} W_{\theta'}(b,f)$.

5.2.2 Edge Weight

An edge annotation captures the cost of a data transfer if its nodes are placed on two different slices. An edge e is weighted by cw, a measure of the criticality of the edge. An edge is considered critical if it is a scalar or mask value transfer and **a)** it is on a critical path within some basic block, or, **b)** it is a transfer to or from a loop that is to be pipelined. Regardless, an edge is never considered critical if it has a high fan out (greater than 4 in our algorithm), in which case we prefer exploiting instruction level parallelism.

Let $\mathbb{1}_L$ be an indicator variable that is 1 if any node of the edge is within a loop that is to be pipelined and 0 otherwise. The critical weight of an edge is calculated as:

$$cw = \begin{cases} 1 + \mathbb{1}_L & \text{if edge is critical,} \\ 0 & \text{otherwise.} \end{cases}$$

5.3 Coarsening

Consider a program dataflow graph with an edge e annotated by weight $w_e = cw$. A *matching* on a graph is a set of its edges with no two edges sharing a common vertex. If the weight of a set of edges is the sum of its individual edge weights, a *maximum weighted matching* is a matching of maximum weight.

To accurately model the cost of data transfers we must not only identify producer-consumer relationships across basic blocks but also the type of data transfer. In our version of coarsening we coalesce the graph by data transfer type and the location of the edge. The maximum weighted matching of edges with a scalar or mask producer and with both vertices in the same basic block is first computed and the graph is coarsened (possibly through multiple levels). Coarsening continues on the resulting graph by coalescing edges of scalar or mask inter basic block type, vector intra basic block, vector inter basic block, and finally AMC input type until we are left with a graph of four or fewer nodes.

Recall that refinement, i.e., the process of mapping nodes to slices operates in the reverse direction so early on vector register sharing within a basic block is minimized but in later stages priority is given to scalar or mask transfers since copying these registers across slices is most expensive. Note also that critical edges, including those incident on a node in a loop, are given highest priority.

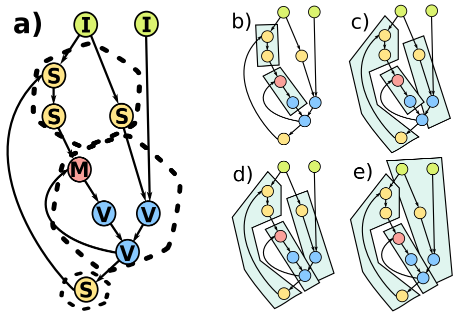

Figure 4: An illustration of coalescing on the graph in **a)** with scalar (**S**), mask (**M**), vector (**V**), and input (**I**) nodes. Nodes in a basic block are enclosed in a dashed hexagon.

Our algorithm is illustrated on the example graph in Figure 4a with the coarsening steps shown in Figure 4b-e. Figure 4b shows the graph after coalescing scalar and mask edges within a basic block. In Figure 4c scalar edges that cross basic block boundaries are coalesced, while Figure 4d shows the graph after all edges with producers within the AMC are considered. Finally, input edges are coalesced to get a graph of three nodes.

5.4 Refinement

Given an initial mapping of nodes to slices, refinement is the process of perturbing the initial assignment to reach an optimal partitioning. Each node is considered in turn and moved to the slice that maximizes a gain function measured as a function of node weights and edge cuts. This process iterates until convergence.

Let $\Phi : V \rightarrow \{0, 1, 2, 3\}$ be an assignment function that maps a node $\theta \in V$ to one of four slices. A good mapping balances the load of slice functional units in each basic block as one objective. The load $L(s, b, f)$ of functional unit f on slice s is computed on a per-basic-block level as

$$L(s, b, f) = \sum_{\theta} W_\theta(b, f) \quad \forall\, \Phi(\theta) = s.$$

Assume a node θ is mapped to slice s_1. The load of two slices can be compared using a metric such as the *Manhattan (L1)* distance. We calculate the distance D between two slices s_1 and s_2 on a basic block b as

$$D(s_1, s_2, b : \Phi(\theta) = s_1) = \sum_{f} |L(s_1, b, f) - L(s_2, b, f)|.$$

The load gain of moving node θ from slice s_1 to s_2 is

$$G^L_\theta(s_1 \rightarrow s_2) = \sum_{b} (D(s_1, s_2, b : \Phi(\theta) = s_1) - D(s_1, s_2, b : \Phi(\theta) = s_2)) \times \beta(b),$$

where the function β maps a basic block to an integer, and can be used to prioritize vectorized loops over residue and scalar code. The node is moved from slice s_1 to s_2 if the gain is positive. Note that it is the multidimensional formulation of node weights that enables load balancing by basic block.

We similarly calculate the effect of moving node θ from slice s_1 to s_2 on critical edge data transfers. Let an edge e be given by its two incident nodes (θ_1, θ_2). The weighted critical edges when node θ is mapped to slice s is given by

$$CE_\theta(s) = \sum_{e=(\theta,\theta')} cw_e + \sum_{e=(\theta',\theta)} cw_e \quad | \; \Phi(\theta) = s \wedge \Phi(\theta') \neq s.$$

Due to the definition of critical weight, priority is given to edges incident on a node within a loop. The gain of moving a node from slice s_1 to s_2 is

$$G^C_\theta(s_1 \rightarrow s_2) = CE_\theta(s_1) - CE_\theta(s_2).$$

Finally, the overall gain function is computed as

$$G_\theta(s_1 \rightarrow s_2) = G^L_\theta(s_1 \rightarrow s_2) + SCALE \times G^C_\theta(s_1 \rightarrow s_2).$$

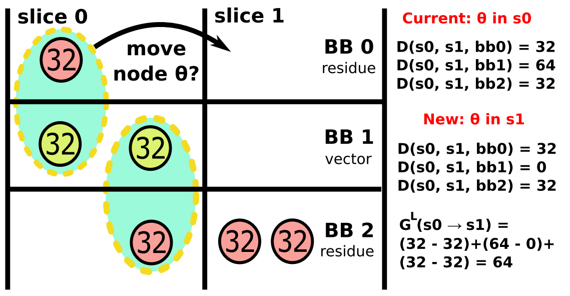

Figure 5: Example of load gain calculation (ALU only) to evaluate if node θ should be moved to slice 1. Our algorithm does so to improve performance of the vectorized loop.

Figure 5 shows the advantage of being able to distinguish load by basic block, where our technique moves node θ to slice 1 to balance load in the vectorized loop. A simple scheme that maintains a unidimensional load value for the node weight does not attain a positive gain.

6. Software Pipelining

Software pipelining [10] is an optimization that improves flop efficiency of loops by partially overlapping successive loop iterations to exploit instruction level parallelism. Successive loop iterations are triggered at a constant time period termed the *initiation interval* (II). Loop execution time is proportional to II so the goal is to minimize it; but the problem of finding a schedule with minimum II is NP-complete.

Modulo Scheduling [17] is a heuristic that attempts to find a feasible pipelined schedule for a chosen II at low computational cost. A pipelined schedule is represented using II rows of a *modulo reservation table* [17] with columns representing hardware resources. Given a dependence graph of the loop the algorithm selects nodes in a well-defined order and places them in slots of the reservation table such that resources are available and no dependencies are violated. If all nodes cannot be placed in feasible slots II is increased and the search repeats. Our algorithm is based on the Swing heuristic [12] extended to a backtracking version [21] that attempts to withdraw out of dead end states by undoing previous scheduling decisions.

Initially a strip mining pass blocks the vectorizable loop by up to 32 iterations resulting in an outer loop iterating over blocks and an inner loop that is then replaced by vector operations. We pipeline the *vector iterations* of the outer loop and use a predicated residue loop to execute the remainder iterations.

6.1 Tracking Vector & Scalar Instruction Resources

We desire a strategy to track and schedule a loop with a mix of vector and scalar code. One technique suggested by Eisenbeis [5] is to use a two-stage scheduler. In the first, coarse-grained stage, vector instructions are scheduled in reservation table slots that represent a block of machine cycles—typically the vector length—known as a granule. Resources required by the instruction are reserved for the entirety of a granule (or multiple granules). A second fine-grained stage expands the granule-level table into machine cycles to schedule scalar instructions.

The advantage of tracking resources at the granule level is reduced compilation time and generation of compact code. There are two serious disadvantages. First, a resource must be reserved for whole granules, i.e., entire multiples of machine cycles, even if it is only used for a fraction of the granule. Second, distinct vector and scalar scheduling introduces a phase-ordering problem. There is no guarantee that the second stage is feasible given a fixed vector schedule.

We propose a joint vector-scalar scheduling strategy for Modulo Scheduling, with slots in the reservation table representing individual machine cycles. Vector instructions are treated as an atomic block for scheduling, and slots in the resource table are reserved according to the template of Section 4. This approach increases the runtime of the scheduler by a factor proportional to the vector length but we can now pipeline instructions in functional units and effectively schedule a mix of scalar and vector instructions. We have found the corresponding increase in runtime to be tolerable.

6.2 Applying the Machine Model to Modulo Scheduling

The abstract machine model is incorporated in our scheduling algorithm by the procedure *SearchInInterval*. The procedure accepts a preselected slice and an interval of time slots for its search. To check if an instruction can be placed at a particular time the resource table in its template is compared, and if compatible, mapped to the modulo reservation table. Comparison occurs in order of time, and resource. Starting at the first time slot of the interval we scan and expand the comparison until it covers a window of cycles of length equal to the size of the template. If expansion fails, we skip to the failure point and restart the expansion.

Note that in general it may not be possible to completely map the template, for example, if predecessor nodes have not yet been scheduled. In such cases we defer processing of that component of the template until the dependencies have been resolved.

6.3 The Modulo Scheduling Algorithm

We briefly outline our Modulo Scheduling algorithm as shown in Figure 6. Nodes are scheduled in an order that minimizes register pressure and prioritizes those in the critical path. An interval of time is selected based on an instruction's predecessors and successors as well as past trials. The algorithm searches this interval within a preselected slice onto which the instruction has been mapped by the cluster assignment pass (Section 5).

The algorithm uses the machine model of Section 4 to check for a feasible matching of the instruction in the reservation table. If unsuccessful, a forced matching on the assigned slice at the earliest time in the search interval is selected. Conflicting instructions are ejected from the reservation table and rescheduled. The algorithm also reschedules instructions that increase the number of live registers at any cycle above a user-defined threshold. We store a history of past search attempts for each node and avoid these slots when rescheduling to ensure forward progress.

```
// Find a schedule for nodes in G at a given II
def ModuloSchedule(Graph G, integer II):
   PriorityQueue Q = OrderedNodes(G)
   integer Budget = Factor * |G|
   while (!Q.IsEmpty() && Budget > 0):
      Node n = Q.pop()
      Slice as = GetAssignedSlice(n)
      Interval iv = GetSearchInterval(n)
      (Slice s, Cycle c) = SearchInInterval(n, as, iv)
      if (c == -1):
         (s, c) = ForceSearchInInterval(n, as, iv)
         EjectConflictingNodes(n, s, c)
      EjectNodesExceedingMaxLive(n, s, c)
      Schedule(n, s, c)
      --Budget
// Failed to schedule if Q is not empty
```

Figure 6: The Modulo Scheduling algorithm for a lane.

7. Experimental Results

In this section we evaluate our techniques 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 the host part of an application. The AMC is modeled by a cycle accurate simulator that precisely represents lanes, the memory interconnect, vault controllers, and DRAM [13]. 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. The domain targeted by the AMC is high performance computing so we experiment on kernels that are of interest to the supercomputing community. In addition, the AMC is an accelerator so we focus on kernels within its scope, i.e., vectorizable codes (see Table 2). DAXPY is a memory-bound kernel that stresses the memory subsystem and DGEMM is a double-precision matrix-matrix multiply kernel that tests the ability of the compiler to exploit vector registers for data reuse to achieve high floating-point utilization. They are traditionally used to benchmark the feed and speed capabilities respectively of HPC machines. DET is a compute-bound kernel that calculates the elemental volume in 3D force modeling assuming that its nodal positions are in contiguous memory, while MESH [11] gathers these nodal co-ordinates through an unstructured mesh.

App	LoC	App	LoC	App	LoC
DAXPY (C)	4	DGEMM (C)	12	DET (C)	58
MESH(C)	59	LULESH (C++)	259	NEKB. (F77)	134

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

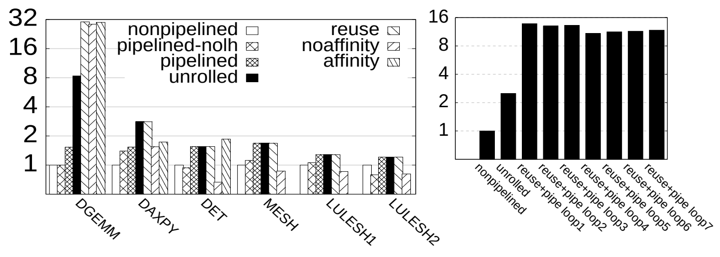

Figure 7: Flop efficiency improvement relative to unpipelined loop due to compiler optimizations for NEKBONE (right) and remaining kernels (left).

LULESH [7] and NEKBONE [6] are two proxy 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. 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 through vector register tiling. We then run on 32 lanes with and without exploiting data affinity within AMC quadrants. Figure 7 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 is expected to worsen significantly as more lanes are activated. Unrolling with pipelining not only provides more vector instructions that help hide latency, but additional instruction level parallelism that can be exploited in the case of DAXPY, DGEMM, and NEKBONE ($1.83\times$, $5.44\times$, and $2.50\times$ 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 through vector register tiling ($3.63\times$ 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 7b showing improvements of 4.34-5.51$\times$ 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 traditional 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 quadrant affinity, when possible, helps negate some of this loss. DET achieves higher performance than the single lane case, indicating that affinity is beneficial even with one lane active.

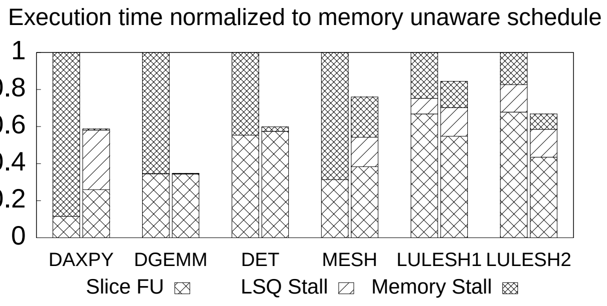

Figure 8: Reduced memory subsystem stalls with a memory aware scheduler.

Performance of the Near-Memory Scheduler. To understand the performance of the scheduling algorithm on the near-memory subsystem, we show in Figure 8 the execution profile with and without memory optimizations. The best performing pipelined kernels from the previous section are run on multiple lanes and execution time is normalized to the memory unaware schedule. The profile compares time when any slice's functional unit is active and doing useful work versus when the entire lane is stalled either due to a full LSQ or on an unsatisfied memory request.

Between 30-90% of execution time is spent stalled on the memory subsystem with the naive schedule. This is true even for the compute bound kernels (66% for DGEMM and 45% for DET). Congestion on the memory interconnect due to traffic from multiple lanes causes the high stall rate. Clearly a Modulo Scheduler that assumes a cache-based system is unsuitable for a processor near memory.

Our memory aware scheduler is always able to reduce execution time (between 15-65%), by either partially hiding latency for memory bound kernels or completely in the case of compute bound kernels. Throttling memory requests reduces LSQ stalls. However, the improved schedule also generates

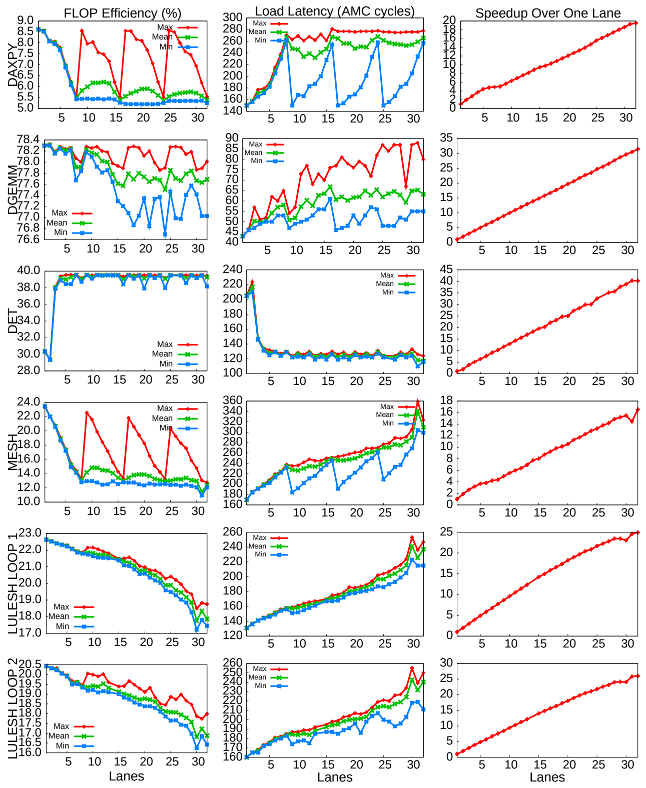

Figure 9: Performance curves for the kernels on the AMC.

more traffic on the memory subsystem which increases load latencies and backs up the LSQ.

Scaling Across Lanes. Next, we study scaling on the 32 lanes of the AMC, charting flop efficiency, average load latency observed at a lane, and speedup relative to one lane in Figure 9. We keep the per lane workload constant (weak scaling), except for MESH and 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, $31.48\times$. We achieve a flop efficiency of 77%, which compares favorably to the 83% of peak performance achieved by careful hand optimization. This is attributed to the excellent schedule, which is able to tightly pack ALU instructions across the floating point units of the four slices in a lane while also hiding memory latency.

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). DET performs an odd number of determinant calculations, however, our scheduler is able to efficiently pipeline the loop across all four slices. Register pressure heuristics while scheduling are critical to balance register use across slices and a less aggressive ALU utilization is automatically selected to avoid spilling.

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.

MESH. MESH achieves only 31% of the efficiency as DET on 32 lanes due to the random access pattern of the unstructured mesh. This drop in performance is explained by a $2.5\times$ increase in load latency observed at the lanes. Latency increases because memory accesses are 8-byte instead of more efficient 32-byte loads as for DET. For programs with indirect accesses we may resolve the indirection by gathering data in advance. This may be beneficial if the application contains several kernels requiring indirection through the same mesh.

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 9 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 early 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.

8. Related Work

Vector software pipelining. Numerous software pipelining models [12, 17, 21] target scalar instructions with a single issue. Our work specifically targets temporal vector operations, including variable length vectors. We build an abstract machine model for Modulo Scheduling that precisely captures the vector functional units, vector register file sharing at the element level, communication between clusters, and interaction between the core and the memory subsystem. As far as we are aware this is the first work to apply Modulo Scheduling in the context of a temporal-vector-VLIW core.

Eisenbeis [5] also builds a model for the CRAY-2 but the work does not exploit software pipelining. They use two stages to schedule vector and scalar instructions. Resources are reserved for entire multiples of machine cycles (the vector length). Both these aspects lead to suboptimal schedules.

Latency hiding. Several cache-based schedulers [9, 18, 20] issue non-critical memory operations early or insert prefetch instructions to prime the cache. Heuristics predict loads that are likely to miss the cache and a latency of tens of cycles can be hidden. In our work we must hide the latency of both critical and non-critical loads, even at the cost of increasing II. Latencies are one order of magnitude higher but we are able to exploit the vector issue to hide close to a hundred cycles. None of these works account for the LSQ so a direct application of these techniques will overload the LSQ.

Clustering. Instruction mapping onto clustered functional units has been studied in the context of cyclic [14] and acyclic codes [4, 16]. Several approaches integrate clustering with scheduling [3, 14] due to their close coupling. We consider this prohibitively expensive for a production compiler. Aletà [1] and Chu [4] use a multilevel graph partitioning algorithm for clustering that is distinct from scheduling.

These and other existing work [2] operates on basic blocks so they make locally optimal decisions. On a procedure with scalar, vector and residue code local clustering leads to poor schedules and/or increased register spills, which is detrimental for performance on a processor near memory. The model we use for node weights can distinguish load gain on individual basic blocks, enabling us to make globally optimal decisions. Our algorithm handles the costs of various types of partitioned register files by, for example, coarsening edges representing scalar data flow before vector transfers in our multilevel graph partitioning algorithm.

9. Conclusions

The carefully designed microarchitecture of the AMC eliminates the complexity of conventional processors but requires a sophisticated compiler to deliver high performance. In this work we have presented an abstract machine model for a vector-VLIW core that we exploited in a Modulo Scheduling algorithm for lanes, and a global clustering algorithm optimized for entire procedures.

We identified three classes of optimizations for high performance near memory: those that hide memory latency, throttle memory bandwidth, and increase compute efficiency. Vectorization (temporally on a slice) instead of traditional SIMD (for example, four-way on slices) was exploited by the scheduler to hide memory latency. The scheduler throttles the issue of memory requests and exploits instruction-level parallelism through co-operative scheduling across slices rather than treating them as independent units to exploit four-way data-level parallelism.

Our results show high computational efficiency and linear performance 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. Our work is an important step toward building a realizable Exascale system.

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.*, 2001.

[2] A. Aletà et al. Exploiting pseudo-schedules to guide data dependence graph partitioning. In *PACT*, 2002.

[3] A. Aletà et al. AGAMOS: A graph-based approach to modulo scheduling for clustered microarchitectures. *IEEE Trans. Comput.*, 2009.

[4] M. Chu et al. Region-based hierarchical operation partitioning for multicluster processors. In *PLDI*, 2003.

[5] C. Eisenbeis et al. Squeezing more CPU performance out of a Cray-2 by vector block scheduling. In *Supercomputing*, 1988.

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

[7] I. Karlin et al. Exploring traditional and emerging parallel programming models using a proxy application. In *IPDPS*, 2013.

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

[9] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *PLDI*, 1993.

[10] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *PLDI*, 1988.

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

[12] J. Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *PACT*, 1996.

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

[14] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Symp. Micro.*, 1998.

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

[16] V. Porpodas and M. Cintra. CAeSaR: Unified cluster-assignment scheduling and communication reuse for clustered VLIW processors. In *CASES*, Sept 2013.

[17] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO*, 1994.

[18] F. J. Sánchez and A. González. Cache sensitive modulo scheduling. In *MICRO*, 1997.

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

[20] S. Winkel et al. Latency-tolerant software pipelining in a production compiler. In *CGO*, 2008.

[21] J. Zalamea et al. Modulo scheduling with integrated register spilling for clustered VLIW architectures. In *MICRO*, 2001.