Memory Access Scheduling to Reduce Thread Migrations

Sana Damani
Georgia Institute of Technology, USA
sdamani@gatech.edu

Prithayan Barua
Georgia Institute of Technology, USA
prithayan@gatech.edu

Vivek Sarkar
Georgia Institute of Technology, USA
vsarkar@gatech.edu

Abstract

It has been widely observed that data movement is emerging as the primary bottleneck to scalability and energy efficiency in future hardware, especially for applications and algorithms that are not cache-friendly and achieve below 1% of peak performance on today's systems. The idea of "moving compute to data" has been suggested as one approach to address this challenge. While there are approaches that can achieve this migration in software, hardware support is a promising direction from the perspectives of lower overheads and programmer productivity. *Migratory thread architectures* migrate lightweight hardware thread contexts to the location of the data instead of transferring data to the requesting processor. However, while transporting thread contexts is cheaper than moving data, thread migrations still incur energy and bandwidth overheads and can be particularly expensive if threads frequently migrate in a ping-pong manner between processors due to poor locality of access.

In this paper, we propose Memory Access Scheduling, a new compiler optimization that aims to reduce the number of overall thread migrations when executing a program on migratory thread architectures. Our experiments show performance improvements with a geometric mean speedup of 1.23× for a set of 7 explicitly-parallelized kernels, and of 1.10× for a set of 15 automatically-parallelized kernels. We believe that memory access scheduling will also be an important optimization for other locality-centric architectures that benefit from software thread migrations, such as multi-threaded NUMA architectures.

CCS Concepts: • **Computer systems organization → Distributed architectures**; *Multicore architectures.*

CC '22, April 02–03, 2022, Seoul, South Korea

ACM ISBN 978-1-4503-9183-2/22/04…$15.00
https://doi.org/10.1145/3497776.3517768

Keywords: Compilers, Emu Architecture, Instruction Scheduling, Integer Linear Programming (ILP), Sequential Ordering Problem, Dataflow Analysis, Thread Migration

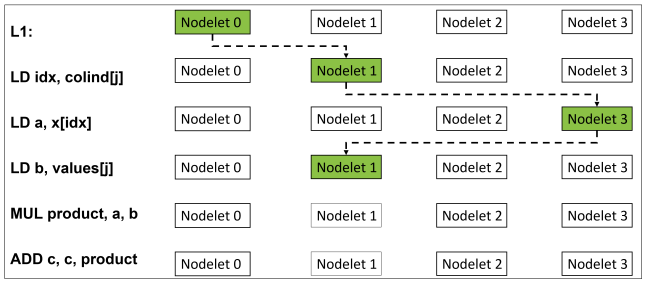
ACM Reference Format:
Sana Damani, Prithayan Barua, and Vivek Sarkar. 2022. Memory Access Scheduling to Reduce Thread Migrations. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC '22), April 02–03, 2022, Seoul, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3497776.3517768

1 Introduction

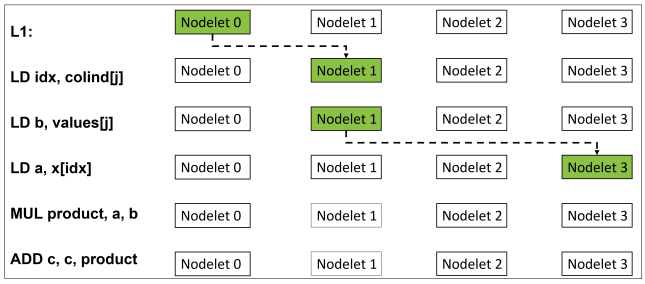
Migratory thread architectures such as Emu [4] reduce memory bandwidth, latency, and energy utilization by migrating lightweight thread contexts to lightweight near-memory processors (called "nodelets") instead of transferring blocks of data to a requesting processor. When a thread encounters a memory read, the hardware checks if the data is available on the current nodelet. If the data is not on the current nodelet, the hardware migrates the thread to the nodelet that holds the requested address. This approach works well for data-intensive applications with weak locality [13], such as graph analysis, data analytics, and machine learning. However, in some cases, a thread may request data from nodelets in an alternating manner, resulting in a thread migration on every access. To study the cost of thread migration, we wrote two microbenchmarks with a single thread that reads data from (1) a replicated array, and (2) a distributed array. We found that the first microbenchmark with 0 migrations ran 15.7× faster on an Emu processor than the second microbenchmark with 100,000 migrations. The cost per-migration is approximately 272 clock cycles, which can be a significant overhead if incurred for a large number of read operations.

```
1       // allocate arrays
2       long *values = mw_malloc1dlong(N);
3       long *x = mw_malloc1dlong(N);
4       long *colind = mw_malloc1dlong(N);
5       ...
6       for (i = 1 to N) {
7         sum = 0;
8         for (j = rowptr[i] to rowptr[i+1]) {
9             sum += values[j] * x[colind[j]];
10        }
11        y[i] = sum;
12      }
```

Listing 1. Sparse Matrix Vector Multiply



(a) Baseline: 3 migrations per iteration



(b) Memory Access Scheduling: 2 migrations per iteration

Figure 1. Thread migrations in SpMV

In this paper, we introduce a new compiler optimization for migratory thread architectures called *Memory Access Scheduling*. Our optimization statically identifies co-located memory accesses and groups them together to reduce the number of thread migrations.

We use the sparse matrix-vector multiplication (SpMV) code from Listing 1 as a running example. The function *mw_malloc1dlong* distributes arrays across nodelets in a round-robin manner starting from *Nodelet 0* (see Figure 3). Hence, *values[j]* and *colind[j]* are always on the same nodelet. The location of *x[colind[j]]* cannot be statically determined, so we conservatively assume that it will result in a migration. Data layouts are further described in Section 2.3.

Table 1. SPMV codegen with Memory Access Scheduling

(a) Baseline	(b) Optimized

`// migrate to nodelet 0`
`S1: R0 = load colind[j];`
`// migrate to nodelet 3`
`S2: R1 = load x[R0];`
`// migrate to nodelet 0`
`S3: R2 = load values[j];`
`S4: R3 = mul R2, R1;`
`S5: sum = add sum, R3;`
	`// migrate to nodelet 0`
	`S1: R0 = load colind[j];`
	`S3: R2 = load values[j];`
	`// migrate to nodelet 3`
	`S2: R1 = load x[R0];`
	`S4: R3 = mul R2, R1;`
	`S5: sum = add sum, R3;`

Table 1(a) shows the code generated for line 9 and Figure 1(a) illustrates the execution of a single iteration of the SpMV loop. We observe that in the base case, there are (at most) three migrations in each iteration of the inner loop resulting in $3N^2$ total migrations. Table 1(b) shows the code generated after memory access scheduling, which groups together accesses from the same nodelet as much as possible, before executing a migration-inducing instruction. The modified schedule has a maximum of two migrations per iteration of the inner loop resulting in $2N^2$ total migrations, a reduction of 33% (see Figure 1(b)). Simulator studies show that total migrations for the SpMV application reduced from $2.2M$ to $1.3M$ migrations with the modified schedule. The remainder of this paper describes how our scheduler identifies and reorders co-located memory accesses.

Our contributions include:

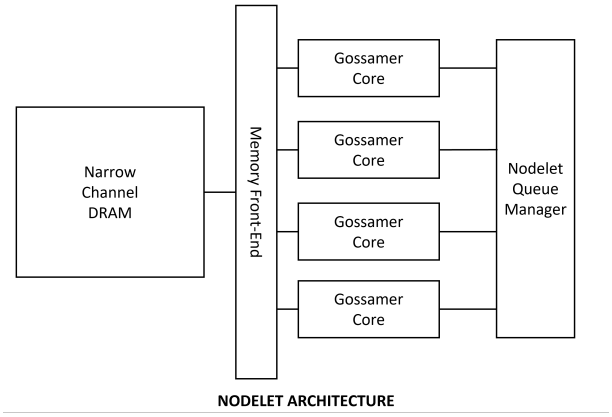
- A novel static analysis, which tracks data layouts and identifies co-located memory accesses.
- Formalization of memory access scheduling as a sequential ordering problem.
- An ILP-based scheduler that reorders instructions to minimize the number of thread migrations.
- An alternate heuristic-based greedy scheduler with lower compile-time overhead relative to the ILP-based scheduler.
- An implementation of our approach in LLVM and experimental results that measure the impact of our schedulers in terms of thread migrations and performance improvement on Emu hardware.

2 Background

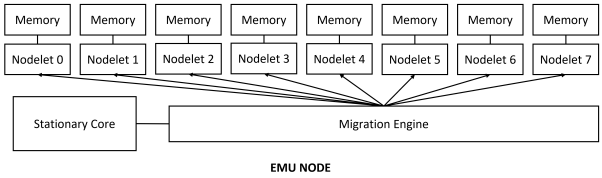
2.1 Emu Architecture

The basic unit of processing in the Emu architecture is a *nodelet* (shown in Figure 2a) [4]. A nodelet consists of multiple *gossamer cores*, general-purpose pipelined processors that execute threads, a *nodelet queue manager* that handles thread scheduling and migration, a memory front-end that handles memory transactions, and a portion of the global address space. A group of 8 nodelets along with a *migration engine*, a cross-bar that migrates threads from the source nodelet to destination nodelet, and a *stationary core* forms a *node* (see Figure 2b). An Emu system may have multiple connected nodes. The architecture has no memory hierarchy and all accesses reference a global memory address space.

When a thread running on a nodelet executes a memory fetch instruction, the hardware first checks if the requested data is present on the current nodelet. If not, the hardware migrates the thread context to the nodelet that holds the requested data. Hence, threads migrate automatically without programmer intervention. A thread context is compact, consisting of a thread status word and address and data registers [4]. The program itself is maintained in replicated memory and can be accessed from any nodelet, and does not need to be transferred with the thread context. Hence, thread migration on the Emu is a cheap operation, but as described in section 1, it can still become a performance bottleneck.



(a) Emu Nodelet



(b) Emu Node

Figure 2. Emu Architecture [4]

2.2 Emu Programming Model

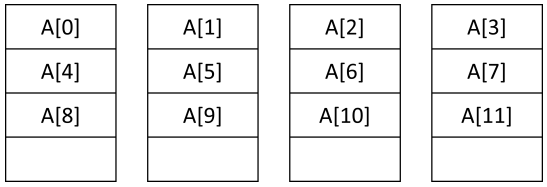
Emu uses the Cilk programming model with the following extensions to C [9]:

- *cilk_spawn* creates a new child thread that can execute in parallel with the parent. A parent can optionally spawn a child on a remote nodelet with a migrate hint.
- *cilk_sync* forces the parent to wait for all its spawned children to finish execution before proceeding.
- *cilk_for* spawns new threads for each iteration of a for loop, which can execute in parallel, and then waits for all the iterations to complete.

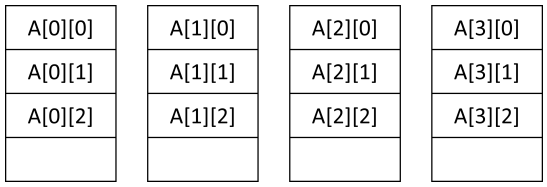
2.3 Data Allocation

Emu-Cilk provides the following functions for dynamic allocation of data structures distributed across nodelets [6].

- *mw_localmalloc(N, ptr)* allocates memory of size N on the same nodelet as ptr.
- *mw_mallocrepl(N)* allocates memory of size N on each nodelet [21]. A copy of a replicated data structure can be found on each nodelet.
- *mw_malloc1dlong(N)* allocates a 1D array of N long elements striped across nodelets in a round-robin manner starting at nodelet 0 (see Figure 3a).
- *mw_malloc2d(B, N)* allocates a 2D array of B blocks of N elements each. The blocks are striped across nodelets in a round-robin manner starting at nodelet 0, whereas elements within a block are co-located on a single nodelet (see Figure 3b for an example with B=4).



(a) mw_malloc1dlong()



(b) mw_malloc2d()

Figure 3. Data Allocation.

2.4 Other Migratory Thread Systems

In Non-Uniform Memory-Access (NUMA) systems, the data access latency depends on where the data is located relative to where the access is being performed. NUMA-aware operating systems aim to co-locate data with the threads that access them as much as possible, so as to minimize this latency. A NUMA variant that migrates threads on remote accesses was shown by Li et al. to be up to 2x faster than data shuffling in past work [11].

Earlier, Rogers et al. proposed using thread migration as an enabler for automatic parallelization of programs that use dynamic data structures on distributed-memory multiprocessors, so as to handle the problem of ensuring that all thread accesses are local when the layout is statically unknown [16]. Contemporaneously, Jenks and Gaudiot proposed the Nomadic Threads run-time system [8]. Similar to Emu threads, a Nomadic thread transfers to the processor that contains data that it requires instead of fetching data from the remote processor with the goal of reducing messages and exploiting locality of access within the target processor.

Our approach to Memory Access Scheduling should be applicable to all such migratory thread approaches, in addition to the Emu system studied in this paper.

3 Approach

We define *co-located* memory accesses as accesses to addresses that are statically known to reside on the same nodelet and would not cause a thread migration when issued consecutively. This includes (1) accesses to replicated memory which never result in a migration, (2) accesses to multiple elements of the same block of a 2D array allocated by mw malloc2d(), and (3) accesses to a 1D array allocated by mw malloc1dlong() with a stride that equals the number of nodelets. Co-located memory accesses represent a different form of locality than traditional cache-based spatial or temporal locality. Co-located accesses may not reside in the same block of memory and may never be reused. However, grouping

together co-located data accesses reduces thread migration. Co-location is a symmetric relation because the order of the memory references does not impact the result.

Our goal then is to identify co-located memory accesses and reorder them to minimize the number of thread migrations while respecting data and control dependencies. We call this the *Memory Access Scheduling* problem. We propose a three-step solution to this problem. First, we identify and propagate data layout information for each memory access through a dataflow analysis we call *Layout Analysis* described in Section 3.1. Next, we use data layout and array index information to identify co-located memory accesses by a process we call *Stride Analysis* (see Section 3.2. Finally, we reorder instructions to minimize thread migrations by grouping together co-located accesses using an ILP-based (Section 3.3) and a heuristic-based instruction scheduler (Section 3.4).

3.1 Layout Analysis

The *Layout Analysis* pass determines how consecutive elements of an array are allocated across the nodelets. For our analysis, we consider the following kinds of data allocation:

- **Local**: The array is allocated on the local memory of the nodelet where the requesting thread is running.
- **Co-located**: The array is allocated on the same nodelet as another pointer using `mw_localmalloc()`.
- **1D**: The array is striped across nodelets.
- **2D**: The array is block striped across nodelets.
- **Replicated**: A copy of the data element resides on each nodelet.

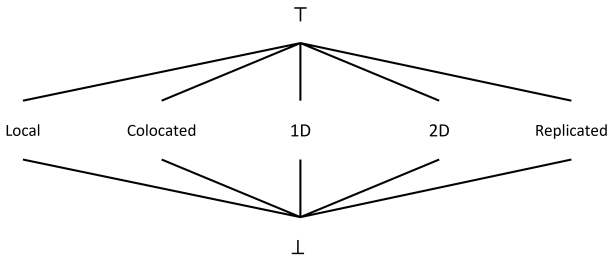


Figure 4. Lattice for layout analysis

The *Layout Analysis* pass is an interprocedural dataflow analysis pass that first traverses all functions in the module to propagate the layout of each pointer or array variable within the function and then propagates the layouts across functions. Algorithm 1 describes our dataflow analysis over the lattice shown in Fig 4.

Once we know the layout of each variable based on the intraprocedural analysis, we propagate this information across function calls using a flow-sensitive and context-insensitive interprocedural analysis. At each of the call sites, if the layout of the arguments in the caller function is determined by the intraprocedural analysis, then we propagate the layout

Algorithm 1: Intraprocedural Layout Analysis: DFA

1 **Domain:** $\{local, colocated, 1D, 2D, replicated\}$
2 **Direction:** $forward$
3 $\top$ = Memory layout not yet determined. This is the initial value, i.e., the empty set {}.
4 $\bot$ = Memory layout cannot be determined. This is the universal set.
5 **Meet Operation:** $\wedge$ on the Lattice as shown in Figure 4
6 **Transfer function:** given instruction
$dst = Instr(op1, op2)$,
$Layout(dst) = Layout(op1) \wedge Layout(op2)$
7 $\forall x$
- $x \wedge \top = x$
- $x \wedge \bot = \bot$
- $c1 \wedge c2 = \bot$, if $c1 \neq c2$

to the callee function parameters, by applying the Meet($\wedge$) operator, $Layout(param) = Layout(param) \wedge Layout(arg)$. On the other hand, if only the layout of the parameters of the callee function is determined by the intraprocedural analysis and the argument layout is unknown, then $Layout(arg) = Layout(param)$. The analysis also propagates the layout of the return value is across the def-use chain of the result of the call statement in the caller function. The interprocedural analysis iterates until convergence, that is, until there are no more updates to the layout information. Convergence is guaranteed by the monotonicity of the transfer function. Finally the layout analysis infers the memory layout for each variable in the program.

3.2 Stride Analysis

Next, we use layout and array index information to determine whether a pair of migration-inducing memory accesses is co-located. For our purposes, we define a migration-inducing memory access as one that could potentially require a thread migration. This includes memory reads and atomics from non-replicated data structures. We propose *Stride Analysis*, a pass that determines the distance between two memory accesses depending on the distribution of the array in memory and the offset into the array. Our analysis relies on the basic assumption that the base address of an array is on nodelet 0 in node 0. The offset from the base address, therefore, decides the nodelet of each element.

Given two memory references, we define *stride* as the relative distance between two migration-inducing memory accesses in terms of nodelets.

- **Case 1**: Both accesses have a 1D layout.
Consecutive elements are located on adjacent nodelets and striped across all nodelets in a round-robin fashion. Hence, the stride between the two accesses is simply the absolute difference between their offsets:
$$stride(A[i_1], B[i_2]) = |i_1 - i_2|$$
- **Case 2**: Both accesses have a 2D layout.

Consecutive rows are on adjacent nodelets so that all elements of a row are on the same nodelet. Hence, the stride is computed using the difference between the row offsets and is independent of the column offset:

$$stride(A[i_1][j_1], B[i_2][j_2]) = |i_1 - i_2|$$

- **Case 3**: A has a 1D layout and B has a 2D layout.

$$stride(A[i_1], B[i_2][j_2]) = |i_1 - i_2|$$

- **Case 4**: They were allocated on the same nodelet using *mw_localmalloc()*.

$$stride(a, b) = 0$$

- **Otherwise**:

$$stride(a, b) = unknown$$

Finally, we use stride information to determine if any pair of memory references $\langle M_1, M_2 \rangle$ is *co-located*. Two non-replicated memory accesses $\langle M_1, M_2 \rangle$ are co-located if their addresses are separated by some multiple of total nodelets, i.e.,

$$stride(M_1, M_2) = num\_nodelets * k$$

for some $k \in Z$.

3.3 ILP-Based Scheduler

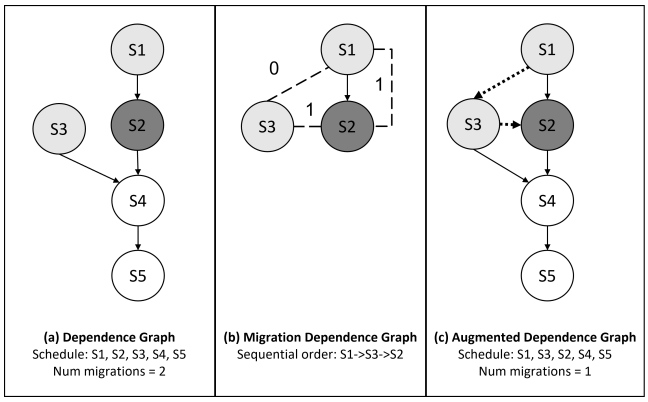


Figure 5. SpMV: Memory Access Scheduling as Sequential Ordering Problem. Vertices shaded the same color are co-located memory accesses. Uncolored vertices represent non migration-inducing instructions. The solid arrows represent data dependences while the dotted arrows represent augmented dependences added by the ILP solver. The undirected weighted edges illustrated by dashed lines represent the cost of migration between nodes.

Given a directed acyclic graph $G = (V, E)$ with a set of nodes V, edges E that represent precedence relations between the nodes, and a distance matrix, a sequential ordering of the nodes is a minimum length path that visits each node exactly once and does not violate precedence constraints [5].

We represent the memory access scheduling problem as a variation of the NP-hard sequential ordering problem [5] on

Algorithm 2: ILP Formulation

Parameters

- n = number of migration-inducing instructions
- $Distance_{ij} = \begin{cases} 0, \text{if i, j are on the same nodelet} \\ 1, \text{otherwise} \end{cases}$
- $Dependence_{ij} = \begin{cases} 1, \text{if j depends on i} \\ 0, \text{otherwise} \end{cases}$

Decision Variables

- T_i, time at which instruction i is scheduled

Objective Minimize number of migrations,

$$\sum_{i=0}^{n} \sum_{j=0}^{n} (T_i = T_j - 1) * Distance_{ij}$$

Constraints

- All instructions must be scheduled, i.e.,

$$\forall i, 1 \leq T_i \leq n$$

- Data dependences must be maintained, i.e.,

$$Dependence_{ij} = 1 \implies T_i \leq T_j$$

- Only one instruction may be scheduled at a time, i.e.,

$$i \neq j \implies T_i \neq T_j$$

the *migratory dependence graph* of a program, where nodes represent migration-inducing memory access instructions, directed edges represent data and control dependencies between instructions, and the distance between each pair of nodes, I_1 and I_2, is given by the migration cost of scheduling I_1 and I_2 consecutively:

$$Distance(I_1, I_2) = \begin{cases} 0, I_1 \text{ and } I_2 \text{ are co-located} \\ 1, I_1 \text{ and } I_2 \text{ are non-co-located} \end{cases}$$

A sequential ordering of the resultant graph, called the Migration Dependence Graph, generates a schedule that minimizes thread migrations and maintains data and control dependences. Figure 5(b) shows the migration dependence graph corresponding to the SpMV example from Listing 1. Algorithm 2 shows our proposed ILP formulation. Our formulation has $O(n^2)$ variables and $O(n^2)$ constraints.

We now describe the overall instruction scheduling algorithm to minimize migrations using the sequential ordering problem (see Algorithm 3).

As a first step, we build a *Migration Dependence Graph* with nodes representing migration-inducing instructions and directed edges between nodes representing direct or indirect dependencies. Next, we use layout and stride analysis (described in Sections 3.1 and 3.2) to determine distances between each pair of nodes in the Migration Dependence Graph. We then pass this Migration Dependence Graph as an input to an ILP solver for the sequential ordering problem described in Algorithm 3. The solver returns an optimal

ordering for the memory access instructions in the Migration Dependence Graph. To maintain the sequential ordering generated by the solver in the final schedule, we augment the original dependence graph of the program with *false dependencies*. If instruction I_1 was scheduled before I_2 in the sequential ordering, we add an edge from $I_1 \longrightarrow I_2$ in the *Augmented Dependence Graph*. Finally, we perform a topological sort on the instructions in the augmented dependence graph, which optimally orders memory accesses while maintaining original dependencies in the program. Figure 5(c) shows the augmented dependence graph for SpMV generated using the ILPScheduler() detailed in Algorithm 3. A topological ordering of the graph in Figure 5(c) results in the optimized schedule from 1(b).

Correctness: Our instruction scheduler does not move instructions across sync/lock instructions. Further, the topological ordering of instructions maintains all data dependences. Hence, reordering instructions within a thread does not introduce data race conditions or affect the correctness of the program.

Theorem. *The augmented dependence graph is a DAG, i.e., there are no cycles in the augmented dependence graph, and the program has a valid schedule.*

Proof. We know that the original dependence graph has no cycles. Assume that the augmented graph does contain a cycle due to a newly added edge from node A to B. For a cycle to exist, there must be a path from node B to A. This path was either an actual dependence from B to A or an augmented edge introduced by the ILP solver.

If there were a dependence from B to A in the original dependence graph, the sequential ordering would respect precedence and schedule B before A in the final schedule by definition. If, on the other hand, B to A contains an augmented edge, then the sequential ordering contains a cycle. In either case, a cycle results in a contradiction. ■

Algorithm 3: ILPScheduler

Input: NodeletMap, DependenceGraph
Output: Schedule of instructions

```
1 Dependencies ← CalcMemDeps(DependenceGraph);
2 Distances ← CalcMemDistances(NodeletMap);
3 T ← SequentialOrdering(Dependencies, Distances);
4 AugmentDependenceGraph(T, DependenceGraph);
5 TopologicalSort(DependenceGraph);
```

3.4 Heuristic Scheduler

We now describe a heuristic-based list scheduling algorithm (algorithm 4) that uses a greedy approach to reduce the number of migrations in the program in polynomial time.

Algorithm 4: Heuristic Scheduler: A list scheduler that groups co-located memory access instructions.

Input: NodeletMap, DependenceGraph
Output: Schedule of instructions

```
1 current ← −1;
2 while some instructions not scheduled do
3     scheduled ← False;
      // schedule non-migrating instructions
4     for inst in all instructions do
5         if inst.ready() ∧
            (¬inst.memoryAccess() ∨ inst.nodelet = current)
          then
6             schedule(inst);
7             scheduled ← True;
      // migrate to new nodelet
8     if ¬scheduled then
9         current ← Migrate();
```

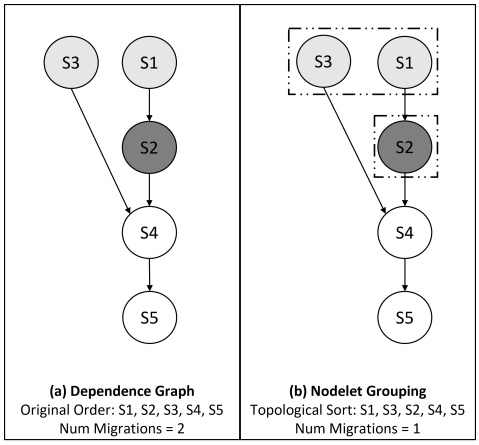


Figure 6. SPMV: Heuristic-based Scheduling.

As a first step, we build a dependence graph for the program where nodes represent instructions and edges represent dependencies. Next, we use layout and stride analysis (described in Sections 3.1 and 3.2) to group memory access instructions on the same nodelet into supernodes, thereby forming a *Dependence Supergraph*. Finally, we perform a topological sort of the dependence supergraph, which tries to schedule all instructions from one nodelet before migrating to a new nodelet. Arithmetic instructions and remote or replicated memory accesses do not induce migrations and may be scheduled whenever their operands are ready. Note that this supergraph may have cycles if none of the nodelet groups have all instructions ready to schedule, in which case we select a ready instruction at random.

Figure 6 shows how the instructions in our SPMV program from Listing 1 are scheduled using the HeuristicScheduler() detailed in algorithm 4. Once again, it can be verified that a

topological ordering of the supergraph in Figure 6(b) results in the minimal-migration schedule from 1(b).

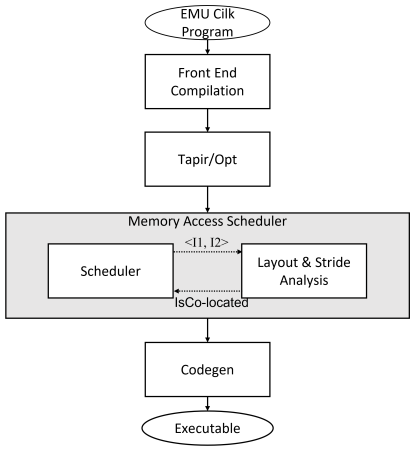


Figure 7. Emu Cilk Compiler Stack.

4 Implementation Details

Figure 7 shows the LLVM-based compilation flow for Emu Cilk based on the OpenCilk compiler framework [19], extended with the memory access scheduler introduced in this paper. The Clang front-end compiler first translates the program to LLVM IR. The next stage performs optimizations across threads [20] before the program passes to the Emu code generation stage, which generates the final executable. The compiler does not have an instruction scheduler and the code generation phase largely maintains the original order of instructions. We implemented all the analysis and transformation passes described in section 3 in LLVM where array access information is still available, and at the end of all LLVM passes to avoid reordering of instructions after memory access scheduling.

Layout Analysis. The Layout analysis determines the memory layout of each variable using the algorithm described in Section 3.1. During a breadth-first traversal over basic blocks, the intraprocedural layout analysis, a *Function Pass* in LLVM, computes the data layout of each variable based on the allocation function used. Next, it propagates the layout information across blocks using the transfer function defined in steps 6 and 7 of Algorithm 1. Finally, the interprocedural layout analysis pass, an LLVM *Module Pass*, propagates the layout information of parameters across function calls.

We illustrate our layout analysis using Listing 2. The analysis assigns a data layout to every LLVM Value. As a first step, we initialize the layout of all values to $\top$ (unknown). For each call to *malloc()*, we set the layout of the result to one of Local, Co-located, 1D, 2D or Replicated based on the type of malloc function called. In Listing 2, we set the layout of pointer p to 1D on line 2, and the layout of *arg2* to 2D on line 14 and Replicated on line 12. LLVM inserts a ϕ−node to join the values of *arg2* from lines 12 and 14. We use the meet operation at the ϕ−node to obtain $Layout(arg2) = 2D \wedge Replicated = \bot$. For each store instruction, we set the layout of the pointer operand to the layout of the value operand. For all other instructions, the pass computes a join over all input operands of the instruction to obtain the layout of the result.

The layout of all other pointers is still $\top$. In the first iteration of the interprocedural analysis, the callee function, *init()*, propagates the layout of the parameter p to the corresponding argument *arg1* in the caller. In the second iteration, the caller, *emuLaunch()*, propagates the layouts of *arg1* to the callee's parameter, *p1* and *arg2* to *p2*. Finally, the analysis propagates the layout of *p1* within *kernel()* across its def-use chain to *array* (line 5). At the end of the layout analysis pass, the layout of *p,arg1,p1,array* is 1D and the layout of *p2,arg2* is undetermined, or $\bot$.

```
1  void init(long **p) {
2    *p = mw_malloc1dlong(n); // p:1D
3  }
4  void kernel(long *p1, long p2) { // p1:1D,p2:⊥
5    long *array = p1; // array:1D
6    ...
7  }
8  void emuLaunch() {
9    long *arg1, *arg2; // arg1,arg2:⊤
10   init(&arg1);       // arg1:1D
11   if (flag) {
12     arg2 = mw_mallocrepl(n); // arg2:Replicated
13   } else {
14     arg2 = mw_malloc1Dlong(n); // arg2:1D
15   }
16   // ϕ-node arg2: ⊥
17   kernel(arg1, arg2);
18 }
```

Listing 2. Layout Analysis: An Example

Stride Analysis. The stride analysis pass uses layout information to assign an *equivalence class* to each variable. Variables in the same equivalence class are co-located, i.e., there is no migration between two memory accesses in the same equivalence class. The stride analysis pass (section 3.2) identifies canonical patterns of 1D and 2D array accesses in the LLVM IR. The pass does a breadth-first-traversal over all instructions and assigns an equivalence class to each pointer variable. Every pointer is initially assigned a unique equivalence class. In Listing 3, lines 3 and 6 correspond to the array accesses $Array1D[index]$ and $Array1D[index + k]$ respectively for a 1D array. If k is divisible by the number of nodelets, i.e., $k \,\%\, num\_nodelets() = 0$, then $EquivalenceClass(l2) = EquivalenceClass(l1)$, that is, the two array accesses are on the same nodelet.

Lines 11 and 16 represent accesses $Array2d[i][j1]$ and $Array2d[i+k][j2]$, where $Layout(Array2D) = 2D$. Since the location of a 2D array access depends only on the row, l4 and temp1 belong to the same equivalence class and l6 and temp2

belong to the same equivalence class. Hence, a3 and a4 belong to the same equivalence class if k % $num\_nodelets() = 0$ (case 2 from section 3.2). Finally, our analysis decides that accesses $Array1D[index]$ and $Array2D[i][j1]$ belong to the same equivalence class if $|index - i|\%num\_nodelets() = 0$ using case 4 from section 3.2.

```
1   // Array1D[index]
2   l1 = Array1D + index;
3   a1 = load l1;
4   // Array1D[index + k]
5   l2 = l1 + k;
6   a2 = load l2;
7   // Array2D[i][j1]
8   l3 = Array2D + i;
9   temp1 = load l3;
10  l4 = temp1 + j1;
11  a3 = load l4;
12  // Array2D[i+k][j2]
13  l5 = l3 + k;
14  temp2 = load l5;
15  l6 = temp2 + j2;
16  a4 = load l6;
```

Listing 3. Stride Analysis: An Example

ILP Scheduler Pass. Once we have computed co-location information for all LLVM values in the program, the ILP scheduler traverses all basic blocks in a function and identifies instructions that may induce a thread migration, including non-replicated memory loads and atomic updates. Replicated accesses and memory stores do not induce migrations and are therefore ignored by the scheduler.

Next, we build a per-basic block dependence matrix for migration-inducing instructions by extending the LLVM dependence analysis pass such that $Dep_{ij} = 1$ if $Instruction_j$ has a direct or indirect dependence on $Instruction_i$. We further build a per-block distance matrix where $Distance_{ij} = 1$ if scheduling $Instruction_j$ after $Instruction_i$ will result in a thread migration i.e. if the stride analysis assigned different equivalence classes to the two instructions.

We feed the Distance and Dependence matrices to the ILP Solver described in 2 which we implemented using IBM's IloCplex solver [3]. The solver returns the optimal schedule of migration inducing memory access instructions in the basic block so that the number of thread migrations are minimized. We use the output of the ILP solver to build the augmented dependence graph as described in 3.3.

Finally, we perform a topological sort on this augmented dependence graph to generate an instruction schedule that preserves the original dependencies and maintains the optimal ordering of memory access instructions as determined by the ILP solver.

5 Evaluation

This section describes the experimental setup and benchmarks used, followed by a detailed analysis of the impact of memory access scheduling on thread migrations, speedup, and compile time. Table 2 summarizes the Emu Chick hardware setup used in our experiments. As mentioned in Section 2.2, this hardware supports the Emu Cilk programming model.

All benchmarks discussed below were ported to Emu Cilk. Each benchmark was compiled with and without memory access scheduling enabled, and profiled on the Emu simulator v20.06 [6] to count the number of thread migrations. We also measured the actual performance impact on real hardware to obtain the average kernel execution time with and without memory access scheduling over three runs.

Table 2. Hardware Setup: Emu Chick Specifications [4]

Configuration	1 Emu Node (8 Nodelets)
Memory	64 GB
Per-Thread Registers	16
Storage	1TB Solid State Disks
Compute	12 Gossamer Cores 1 Stationary Core
Gossamer Core Clock Speed	150 MHz
System Interconnect	Serial RapidIO (SRIO)

Table 3. Explicitly-parallelized kernels with sources

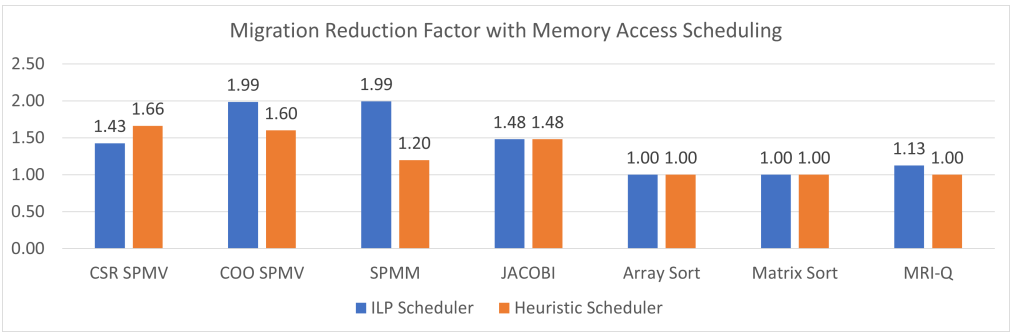
Benchmark	Description
SpMV CSR	CSR Sparse Matrix-Vector Multiply [14]
SpMV COO	COO Sparse Matrix-Vector Multiply [14]
SpMM	Sparse Matrix-Matrix Multiply [26]
Jacobi	Simulation of thermal transmission [12]
Array Sort	Quick Sort an array of elements [10]
Matrix Sort	Sort elements in sparse matrix [10]
MRI-Q	3D MRI reconstruction [22]

Table 4. Compile Time

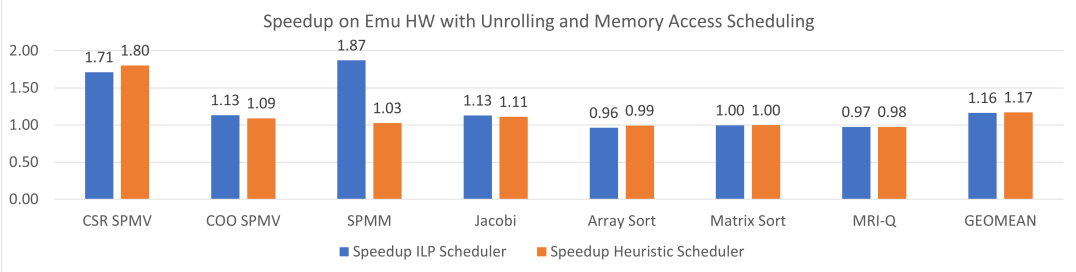
Benchmark	Base (s)	ILP (s)	Heuristic (s)
SpMV CSR	1.11	1.47	1.437
SpMV COO	1.141	9.647	1.39
SpMM	1.265	1.336	1.303
Jacobi	1.230	1.533	1.271
Array Sort	1.247	1.356	1.284
Matrix Sort	1.768	1.983	1.804
MRI-Q	2.244	28.893	2.282

5.1 Performance Results

Table 3 lists the 7 explicitly-parallel kernels used in our evaluation, along with their sources. These kernels used all three Emu Cilk primitives listed in Section 2.2: *cilk_spawn*,



(a) Migration Reduction Factors



(b) Speedup

Figure 8. Results: Explicitly-parallelized kernels.

cilk_sync, and *cilk_for*. Figure 8(a) shows the migration reduction factor[1] for each of the benchmarks, measured on the simulator for both the ILP scheduler and the heuristic scheduler. Figure 8(b) shows the corresponding speedup on actual hardware with a single Emu node. The baseline for all our experiments is the Emu LLVM-based compiler, using the default compilation flags. We see that memory access scheduling reduces migrations by up to 1.99× based on the simulator, and resulted in a speedup on real hardware of up to 1.87× with a geometric mean speedup of 1.23×. (GEOMEAN MAX refers to the geometric mean of the maximum speedup obtained by the ILP and heuristic schedulers, across all benchmarks.)

We also evaluated our approach on a collection of 15 automatically-parallelized benchmarks from the Polybench suite [15] using the standard dataset as input. Automatic parallelization was performed using the PPCG compiler [24] to obtain OpenMP code, which was converted to Emu Cilk by replacing OpenMP parallel for loops by *cilk_for* loops. We excluded benchmarks for which PPCG failed to parallelize the loops. We used the loop unroll pragma to enable loop unrolling by a factor of 2 in kernels that had an inner loop over columns of a 2D array. The 15 benchmarks chosen were those for which PPCG successfully generated parallel code in our evaluation.

[1]The migration reduction factor is the ratio of the original number of migrations to the number of migrations after memory access scheduling.

Figure 9 shows the speedup for the automatically parallelized Polybench benchmarks using memory access scheduling on a single node configuration of the Emu hardware. Our results show that memory access scheduling improves runtime by up to 1.43× for these applications with a geometric mean of 1.10×.

5.2 Analysis

Impact of Register Pressure. Some benchmarks showed unexpected slowdowns, or lack of improvement, with memory access scheduling enabled. Upon further investigation, a major contributor to this anomaly was the impact of instruction reordering on register pressure, which in some cases resulted in increased execution time due to spill related migrations. Note that a register spill that is inserted between co-located memory accesses can result in two additional migrations (to the spill location and back) when there would be none in the absence of a spill. The examples with the largest slowdowns were *lu* and *syr2k* in Figure 9, with ILP Scheduler speedups of 0.87× and 0.94× respectively. Looking at the generated code, we saw a 40% and 6% increase in static spill-related instructions after memory access scheduling respectively. We also studied the dynamic number of spill related migrations for *lu* and found that the percentage of spill related migrations increased from 2% to 40% with memory access scheduling. For *jacobi*, even though the number of migrations decreased by a large factor, the overall speedup wasn't as high because of an increase in the number of memory access instructions. These results show that memory

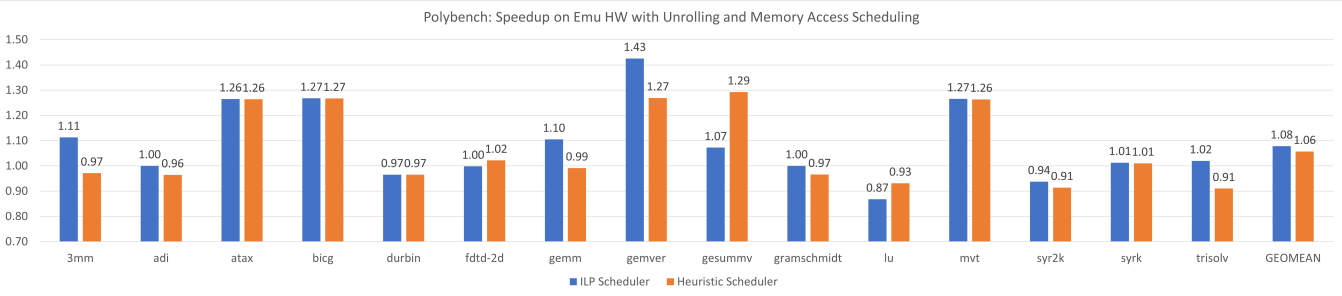


Figure 9. Polybench Speedup

Table 5. Impact of Loop Unrolling on Jacobi 2D

(a) Baseline:$3N^2$ migrations	(b) Unroll:$1.5N^2$ migrations

`Iteration 0:`
`  // migrate`
`  ld a[i][j];`
`  ld a[i][j-1];`
`  ld a[i][j+1];`
`  // migrate`
`  ld a[i-1][j];`
`  // migrate`
`  ld a[i+1][j];`
`Iteration 1:`
`  // migrate`
`  ld a[i][j+1];`
`  ld a[i][j];`
`  ld a[i][j+2];`
`  // migrate`
`  ld a[i-1][j+1];`
`  // migrate`
`  ld a[i+1][j+1];`
	`Iteration 0:`
	`  // migrate`
	`  ld a[i][j];`
	`  ld a[i][j-1];`
	`  ld a[i][j+1];`
	`  ld a[i][j+1];`
	`  ld a[i][j];`
	`  ld a[i][j+2];`
	`  // migrate`
	`  ld a[i-1][j];`
	`  ld a[i-1][j+1];`
	`  // migrate`
	`  ld a[i+1][j];`
	`  ld a[i+1][j+1];`

access scheduling is more effective for programs with low register pressure or architectures with more thread registers. It also motivates future work on integrating memory access scheduling, instruction scheduling and register allocation, following past work on combining instruction scheduling with register allocation.

Impact of Loop Unrolling. To illustrate how loop unrolling can have an impact of memory access scheduling, Table 5 shows the difference between memory access scheduling of the innermost loop of a 2D Jacobi kernel without and with a 2× loop unrolling. The unrolled loop provides a larger window of instructions to the scheduler which is able to group co-located accesses across the two unrolled loop iterations. We see that the unrolled version has approximately $1.5N^2$ migrations compared to the original loop for which the memory access scheduler generated code with $3N^2$ migrations. Loop unrolling by a larger factor would allow for grouping more co-located accesses across loop iterations, thereby reducing the number of migrations further. For our experiments, we empirically selected an unroll factor of 2

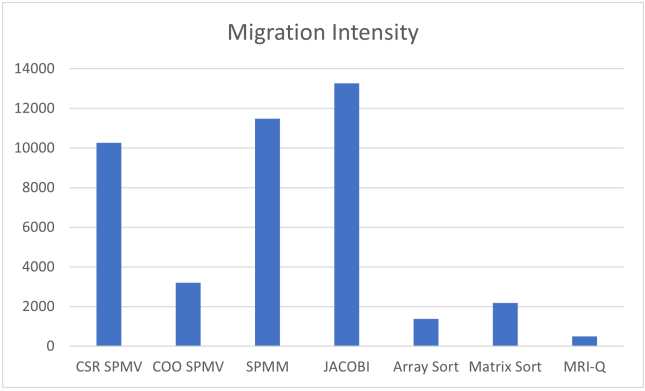


Figure 10. Migration Intensity

for both the baseline and optimized kernels. While a larger unroll factor may expose more opportunity for grouping co-located memory accesses across iterations, this increase in opportunity comes at the cost of increased register pressure resulting in register spills which, as discussed above, can increase thread migrations rather than reducing them.

Heuristic scheduler vs. ILP scheduler. While we expect the ILP scheduler to generate an optimal schedule for memory access instructions within a basic block, there are cases where the heuristic scheduler outperforms the ILP scheduler in practice (e.g., for CSR SPMV and gesummv) because of accidental differences in spill instruction interleaving which are not modeled by either algorithm. This also motivates why we developed two schedulers, and recommend using whichever one delivers better performance for a given application.

Optimization Opportunity. As with other compiler optimizations, the impact of memory access scheduling depends on the extent to which the underlying program is bottlenecked by the overhead targeted by the optimization, viz., thread migrations in this case. To better understand optimization opportunity, we computed the ratio of number of migrations to execution time in the original program as a *migration intensity* metric. Figure 10 shows this metric for the explicitly-parallelized benchmarks. The three benchmarks with the largest migration intensity are Jacobi, SPMM and

CSR SPMV. Among these, memory access scheduling showed large speedups of 1.87× and 1.71× for SPMM and CSR SPMV respectively. However, the improvement for Jacobi was not as large, though still respectable at 1.13×. This is because of an increase in spill instructions in Jacobi due to which a large (1.48×) migration reduction factor did not result in a proportional improvement in execution time.

Compilation Time: We evaluated the compile time impact of our pass by measuring the overall compilation time for each of the benchmarks using the base compiler, the ILP scheduler, and the heuristic scheduler. Table 4 shows that while the ILP scheduler increases compile time by a factor of two for COO SPMV and a factor of seven for the MRI-Q benchmark, the compile-time for the remaining benchmarks is comparable to that of the baseline compiler (within 1.5×). The heuristic scheduler has compile-time within 1.2× of the baseline compiler for all benchmarks we studied. For larger blocks or higher unroll factors, we expect the ILP scheduler to take significantly more time than the heuristic scheduler.

6 Related Work

Belviranli et al. proposed algorithm-level optimizations using improved thread creation strategies and better data distribution at the program level to reduce thread migrations on the Emu architecture [1]. Rolinger et al. addressed the problem of thread migrations and load balancing in SpMV by optimizing data layouts to allocate data blocks across nodelets, so that memory access latency is reduced [17] [18]. Hein et al. studied the impact of replication, remote writes, and data layout transformations on reducing thread migrations [7]. Page proposed a prefiltering mechanism to avoid redundant migrations [13]. These approaches rely on changing the algorithm or data layout. On the other hand, our approach automatically reorders memory accesses and can therefore automatically reduce migrations across applications and can handle multiple access patterns on the same data structures.

Another data layout optimization to reduce thread migrations in bitonic sort was proposed by Velusamy et al. [23]. They use a dynamic data remapping strategy instead of a static data layout to avoid remote memory accesses in their work. However, this run-time remapping operation has an associated runtime cost that memory access scheduling does not incur. Chatarasi and Sarkar used traditional compiler loop transformations to increase memory access locality, such as loop fusion, to reduce thread migrations in graph applications on the Emu architecture [2]. We believe that such loop transformations may also expose more opportunity for memory access scheduling.

7 Conclusions

Near memory and distributed processors pose new challenges for code generation. In particular, we find that thread migrations often become a performance bottleneck for irregular programs with weak locality of access. We propose a new instruction scheduling approach that aims at minimizing redundant thread migrations by grouping together co-located memory accesses within the program. Unlike prior work, our approach attacks the problem of redundant thread migrations automatically within the compiler, without changes to data layouts or to the algorithm. We propose both an optimal ILP-based solution and an approximate greedy scheduler with lower compile time overhead. Further, we combined memory access scheduling with loop unrolling to increase the scheduling window. Experimental evaluation on the Emu hardware shows speedups of up to 1.87× with geometric mean 1.23× for a set of explicitly-parallelized kernels and up to 1.43× and a geometric mean of 1.10× for a set of automatically parallelized kernels. Our findings suggest that kernels bottlenecked on thread migrations and with relatively low register pressure are ideal candidates for memory access scheduling.

While static analysis of memory access co-location information sufficed for the benchmarks studied, other programs may benefit from run-time information regarding how often two accesses are co-located. Alternatively, multi-version compilation may help in cases where static information does not accurately determine co-location information. Further, any improvements to dependence analysis and alias analysis will also increase the accuracy of our static analysis pass. Traditional improvements to instruction scheduling, including global scheduling and the introduction of a post-register allocation scheduling pass may improve the overall performance of memory access scheduling by increasing the size of the scheduling window or by reducing the impact of spill-related migrations, respectively. Finally, in out-of-order execution, a hardware instruction scheduler may be able to use more accurate run-time co-location information to reorder instructions to minimize thread migrations.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1822919. We want to thank Tim Dysart and Shannon Kuntz at Lucata for their guidance with the Emu Cilk compiler and simulator and Brian Page and Jeffrey Young for their support with the Emu hardware. We would also like to thank Prasanth Chatarasi for his input on this work. This research was supported in part through research infrastructure and services provided by the Rogues Gallery testbed [25] hosted by the Center for Research into Novel Computing Hierarchies (CRNCH) at Georgia Tech. The Rogues Gallery testbed is primarily supported by the National Science Foundation (NSF) under Grant No. 2016701. This research was also funded in part by the NVIDIA Graduate Fellowship.

References

[1] M. E. Belviranli, S. Lee, and J. S. Vetter. 2018. Designing Algorithms for the EMU Migrating-threads-based Architecture. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. https://doi.org/10.1109/HPEC.2018.8547571

[2] Prasanth Chatarasi and Vivek Sarkar. 2018. A Preliminary Study of Compiler Transformations for Graph Applications on the Emu System. In *Proceedings of the Workshop on Memory Centric High Performance Computing* (Dallas, TX, USA) *(MCHPC'18)*. Association for Computing Machinery, New York, NY, USA, 37–44. https://doi.org/10.1145/3286475.3286481

[3] International Business Machines Corporation. [n.d.]. V12.10.0: User's Manual for CPLEX. ([n. d.]).

[4] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Ruttenberg, M. Ruttenberg, and S. Stein. 2016. Highly Scalable Near Memory Processing with Migrating Threads on the Emu System Architecture. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. https://doi.org/10.1109/IA3.2016.007

[5] L.F. Escudero. 1988. An inexact algorithm for the sequential ordering problem. *European Journal of Operational Research* (1988). https://doi.org/10.1016/0377-2217(88)90333-5

[6] E. Hein, T. Conte, J. Young, S. Eswar, J. Li, P. Lavin, R. Vuduc, and J. Riedy. 2018. An Initial Characterization of the Emu Chick. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. https://doi.org/10.1109/IPDPSW.2018.00097

[7] Eric R. Hein, Srinivas Eswar, Abdurrahman Yaşar, Jiajia Li, Jeffrey S. Young, Thomas M. Conte, Ümit V. Çatalyürek, Richard Vuduc, Jason Riedy, and Bora Uçar. 2020. Programming Strategies for Irregular Algorithms on the Emu Chick. *ACM Trans. Parallel Comput.* 7, 4, Article 25 (Oct. 2020), 25 pages. https://doi.org/10.1145/3418077

[8] S. Jenks and J.-L. Gaudiot. 1996. Nomadic Threads: a migrating multithreaded approach to remote memory accesses in multiprocessors. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique*. 2–11. https://doi.org/10.1109/PACT.1996.554028

[9] P. M. Kogge. 2019. Multi-threading Semantics for Highly Heterogeneous Systems Using Mobile Threads. In *2019 International Conference on High Performance Computing Simulation (HPCS)*. https://doi.org/10.1109/HPCS48598.2019.9188165

[10] Jiajia Li, Yuchen Ma, Xiaolong Wu, Ang Li, and Kevin J. Barker. 2019. PASTA: A Parallel Sparse Tensor Algorithm Benchmark Suite. *CCF Transactions on High Performance Computing* 1, 2 (8 2019). https://doi.org/10.1007/s42514-019-00012-w

[11] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. 2013. NUMA-aware algorithms: the case of data shuffling. In *CIDR*.

[12] Rubén Gran Tejero Marcos Canales Mayo. [n.d.]. jacobi-mpi. https://github.com/mcanalesmayo/jacobi-mpi.

[13] Brian A. Page. [n.d.]. *Scalability of Irregular Problems*. Ph.D. Dissertation. https://doi.org/10.7274/r0-7qfb-6m58

[14] B. A. Page and P. M. Kogge. 2020. Scalability of Sparse Matrix Dense Vector Multiply (SpMV) on a Migrating Thread Architecture. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. https://doi.org/10.1109/IPDPSW50202.2020.00088

[15] Louis-Noel Pouchet and Tomofumi Yuki. 2015. PolyBench/C 4.1. *SourceForge. Available online: http://polybench. sourceforge. net/(accessed on 12 August 2020)* (2015).

[16] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. 1995. Supporting Dynamic Data Structures on Distributed-Memory Machines. *ACM Trans. Program. Lang. Syst.* 17, 2 (mar 1995), 233–263. https://doi.org/10.1145/201059.201065

[17] T. Rolinger, C. Krieger, and A. Sussman. [n.d.]. Optimizing Data Layouts for Irregular Applications on a Migratory Thread Architecture. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. https://doi.org/10.1109/MCHPC49590.2019.00009

[18] T. B. Rolinger and C. D. Krieger. 2018. Impact of Traditional Sparse Optimizations on a Migratory Thread Architecture. In *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. https://doi.org/10.1109/IA3.2018.00013

[19] Tao B Schardl, I-Ting Angelina Lee, and Charles E Leiserson. 2018. Brief announcement: Open cilk. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. 351–353.

[20] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2017. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. Association for Computing Machinery. https://doi.org/10.1145/3018743.3018758

[21] Paul L. Springer, Thomas Schibler, Géraud Krawezik, Jack Lightholder, and Peter M. Kogge. 2020. Machine Learning Algorithm Performance on the Lucata Computer. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. https://doi.org/10.1109/HPEC43674.2020.9286158

[22] John A. Stratton, Christopher Rodrigrues, I-Jui Sung, Nady Obeid, Liwen Chang, Geng Liu, and Wen-Mei W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report IMPACT-12-01. University of Illinois at Urbana-Champaign, Urbana.

[23] K. Velusamy, T. B. Rolinger, and J. McMahon. 2020. Performance Strategies for Parallel Bitonic Sort on a Migratory Thread Architecture. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. https://doi.org/10.1109/HPEC43674.2020.9286172

[24] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* (2013). https://doi.org/10.1145/2400682.2400713

[25] Jeffrey S. Young, Jason Riedy, Thomas M. Conte, Vivek Sarkar, Prasanth Chatarasi, and Sriseshan Srikanth. 2019. Experimental Insights from the Rogues Gallery. In *2019 IEEE International Conference on Rebooting Computing (ICRC)*. 1–8. https://doi.org/10.1109/ICRC.2019.8914707

[26] Tong Zhou. [n.d.]. sparse-tiling. https://gitlab.com/tongzhou/sparse-tiling.