Associative Instruction Reordering to Alleviate Register Pressure

Prashant Singh Rawat[†], Aravind Sukumaran-Rajam[†], Atanas Rountev[†], Fabrice Rastello[‡],
Louis-Noël Pouchet[§], P. Sadayappan[†]
[†] The Ohio State University, USA
{rawat.15, sukumaranrajam.1, rountev.1, sadayappan.1}@osu.edu
[‡] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
fabrice.rastello@inria.fr
[§] Colorado State University, USA
pouchet@colostate.edu

***Abstract*—Register allocation is generally considered a practically solved problem. For most applications, the register allocation strategies in production compilers are very effective in controlling the number of loads/stores and register spills. However, existing register allocation strategies are not effective and result in excessive register spilling for computation patterns with a high degree of many-to-many data reuse, e.g., high-order stencils and tensor contractions. We develop a source-to-source instruction reordering strategy that exploits the flexibility of reordering associative operations to alleviate register pressure. The developed transformation module implements an adaptable strategy that can appropriately control the degree of instruction-level parallelism, while relieving register pressure. The effectiveness of the approach is demonstrated through experimental results using multiple production compilers (GCC, Clang/LLVM) and target platforms (Intel Xeon Phi, and Intel x86 multi-core).**

***Index Terms*—Compilers, register pressure, associative reordering, domain-specific optimization**

I. Introduction

As we approach the end of Moore's law scaling, there will be increasing emphasis on maximizing single-node efficiency, in addition to achieving good scalability across the nodes of a supercomputer. Compilers play a very critical role in enabling high performance and efficiency within a node. In this paper, we revisit a compiler problem that is generally considered to be solved for all practical purposes: register allocation. For most programs, the register allocation and instruction scheduling strategies in production compilers are very satisfactory, and the number of register spills is well controlled. However, this is not the case for many compute-intensive array-based applications in computational science and machine-learning/data-science that feature multiple inter-related reuses. With such computations, a number of variables in a basic block of instructions exhibit many coupled reuses, with different groups of instructions referencing different combinations of the set of reused variables. As clearly demonstrated with quantitative data later in the paper, existing register management strategies in production compilers are unable to effectively control the number of register spills for such computations. These many-to-many reuse patterns typically involve associative accumulation of multiple contributions into destination variables. In this paper, we develop an effective instruction reordering strategy that exploits the flexibility offered by associative operations. We demonstrate significant alleviation of register pressure and spilling, and corresponding enhancement of performance.

We use a high-order "box" stencil computation to elaborate on the addressed problem. First, let us consider a simple 2D 9-point Jacobi computation:

```
for(i=1;i<N-1;i++)
   for(j=1;j<N-1;j++)
      A[i][j] = c*(B[i-1][j-1] + B[i-1][j] + B[i-1][j+1] +
               B[i][j-1] + B[i][j] + B[i][j+1] +
               B[i+1][j-1] + B[i+1][j] + B[i+1][j+1]);
```

For computing each element $A[i][j]$, the corresponding element $B[i][j]$ and all 8 neighboring elements are read. For two adjacent result elements, $A[i][j]$ and $A[i][j+1]$, six of the needed elements from B are common. By explicitly unrolling the inner j loop, these potential reuses are exposed in the resulting basic block of code in the unrolled loop. However, for a 4-way unrolled version of an 81-point convolution stencil, with GCC-7.2.0 as the base compiler on an Intel i7-6700K processor, we observe the number of memory accesses per iteration increases from 269 to 664 in the generated assembly code, and the performance drops from 72 GFLOPS to 70 GFLOPS when going from the original non-unrolled form to the unrolled version.

For such high-order stencils, the problem has been recognized in prior work and a solution provided: exploit associativity to reorder operations and create a different equivalent stencil pattern [1]. With this changed access pattern, it has been shown [1] that the maximum number of concurrently live registers reduces from $O(k^2)$ to just $O(k)$ for an order-k box stencil. Rearranging the contributions for the 81-point convolution stencil, such that the contribution from an input value is "scattered" to different output points, brings down the memory accesses per iteration in the unrolled code to 206, and the performance increases from 70 GFLOPS to 137 GFLOPS.

In this paper we develop a general solution for the associative reordering problem, without using any specialized abstractions such as stencil patterns as was used by Stock et al. [1]. We devise an instruction reordering strategy that simul-

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

taneously considers the flexibility with associative reordering of accumulations and the impact of instruction order on the maximum number of live values.

We develop a List-based Adaptive Register-reuse-driven Scheduler (LARS) that uses multiple criteria, including affinities of non-live variables to those live in registers, as well as potentials of variables to fire operations and to release registers. We chose to implement the instruction scheduler as a source-to-source pass outside the compiler, to be executed before the standard compiler passes. By doing so, we are able to evaluate its impact with two production compilers, LLVM and GCC. We present experimental results on a large collection of benchmarks that exhibit significant potential register-level reuse for array elements. We demonstrate significant benefits from the use of LARS.

The paper makes the following contributions:

- It develops a framework to reduce register pressure by exploiting the flexibility of associative reordering for computational kernels with multiple inter-related reuses.
- It develops a flexible multi-criteria instruction reordering heuristic that can be adapted across architectures.
- It demonstrates the effectiveness of the proposed framework for a number of scientific kernels when compiled with different compilers and on multiple architectures.

II. Background and Motivation

Register Allocation and Instruction Scheduling: Register allocation assigns physical registers to the variables used in the intermediate representation (IR). All variables that are live at a given program point must be assigned to distinct registers if register spills are to be avoided. We denote as MAXLIVE the maximum number of variables that are simultaneously live at any program point through the execution of the program. A *data reuse* refers to multiple accesses to the same variable. *Spill* instructions are generated when the allocator runs out of physical registers: some registers are freed after their contents are stored into memory or simply discarded. The spilled contents must be reloaded into registers before their subsequent use. Since memory accesses have high latency, minimizing spills is important for performance.

Several techniques have been proposed to perform register allocation. Most compilers use versions of graph coloring [2] or linear scan [3]. Both these techniques perform register allocation on a fixed instruction schedule that is obtained after performing instruction scheduling on the IR. The instruction scheduler orders the instructions to minimize the schedule length and simultaneously increase the instruction-level parallelism (ILP), so that the functional units and pipelines of the underlying processor are effectively utilized. Clearly, the objectives of instruction scheduling and register allocation can be antagonistic: instruction scheduling may prefer independent instructions scheduled in proximity to increase ILP, whereas register allocation may prefer data-dependent instructions to be scheduled in proximity to shorten the live ranges. This problem is more pronounced for applications that exhibit complex, many-to-many data reuse pattern: the instruction scheduler does not consider the reuse pattern while generating the initial instruction ordering, resulting in increased live ranges for variables.

Listing 1: Unrolled input

```
for (int j=2; j<N-2; j++)
   for (int i=2; i<N-2; i++) {
      A[j][i] = a*B[j-2][i] + b*B[j-1][i] + c*B[j][i] +
                d*B[j+1][i] + B[j][i]*B[j+2][i];
      A[j+1][i] = p*B[j-1][i] + q*B[j][i] + r*B[j+1][i] +
                s*B[j+2][i] + B[j+1][i]*B[j+3][i];
   }
```

Listing 2: A reordering that leverages associativity of +

```
for (int j=2; j<N-2; j++)
     for (int i=2; i<N-2; i++) {
          A[j][i]    = a*B[j-2][i];
          A[j][i]   += c*B[j][i] + B[j][i]*B[j+2][i];
          A[j+1][i]  = s*B[j+2][i] + q*B[j][i];
          A[j][i]   += b*B[j-1][i] + d*B[j+1][i];
          A[j+1][i] += p*B[j-1][i] + r*B[j+1][i];
          A[j+1][i] += B[j+1][i]*B[j+3][i];
     }
```

Associative Instruction Reordering: Sometimes, a better instruction reordering can be achieved if one leverages associativity of operations. Although floating-point additions are not strictly associative, it is generally acceptable to perform associative reordering of accumulations in most applications that do not rely on rounding behavior. In fact, many scientific computations are compiled using *-ffast-math* flag [4], which allows a compiler to exploit associativity of floating-point operations to improve performance at the expense of IEEE compliance. Many recent efforts have leveraged operator associativity to drive code optimization strategies [1], [5], [6].

Consider the input program of Listing 1. The two statements read from four common input values. If the computation of the first statement entirely precedes the second statement, then these four values must be kept alive in registers. However, one can leverage the associativity of addition to reorder the computation as shown in Listing 2. In the reordered computation, the evaluation of the two output points is interleaved, so that all the uses of an input value are brought closer, and consequently, its live range is shortened. However, most production compilers perform instruction scheduling and register allocation on an IR like Register Transfer Language (RTL), which is low-level, and much closer to the machine mnemonics. Most compiler frontends operate on one or more high-level IRs. For example, GCC has two high-level IRs: GIMPLE and Static Single Assignment (SSA). GIMPLE is closer in spirit to the input, and captures operator associativity naturally. When GIMPLE is lowered to SSA, the accumulation operations are converted to a *use-def* chain of contributions. SSA is further lowered to RTL by applying a sequence of intra and interprocedural optimizations. Leveraging operator associativity at RTL would involve lifting the complex RTL to a higher GIMPLE-like abstraction that naturally expresses operator associativity, which is difficult. Therefore, most compilers fail to fully utilize operator associativity to relieve register pressure.

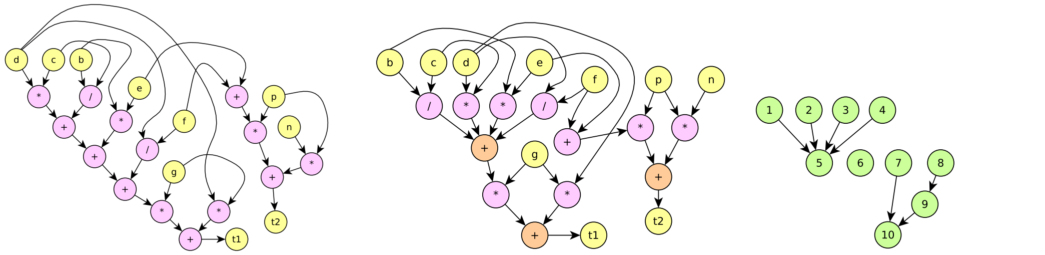
Solution Approach: Many prior efforts have studied the implications of phase ordering between register allocation

```
t1 = ((d*c) + (b/c) + (b*e) + (d/f)) * g + d * g;
t2 = (n*p) + ((f+e) * p);
```

(a) Illustrative computation

```
1.  a = d * c;
2.  a += b / c;
3.  a += b * e;
4.  a += d / f;
5.  t1 = a * g;
6.  t1 += d * g;
7.  m = n * p;
8.  q = f + e;
9.  r = q * p;
10. t2 = m + r;
```

(b) Initial schedule



(c) Input CDAG

(d) CDAG with accumulations

(e) Dependence graph

Fig. 1: Example: statements lowered down to sequence of instructions, and corresponding CDAG

Listing 3: Input in a representative DSL

```
1  void j3d7pt (float *out, float *in, int a, int b, int c) {
2   ...
3   for (int k=0; k<N; k+=2)
4     for (int j=0; j<N; j+=2)
5       for (int i=0; i<N; i++) {
6         #pragma dsl begin iterator k,j,i unroll k=2,j=2
7         out[k][j][i] = a*(in[k+1][j][i]) + b*(in[k][j-1][i]
8              + in[k][j][i-1] + in[k][j][i] + in[k][j][i+1]
9                       + in[k][j+1][i]) + c*(in[k-1][j][i]);
10        #pragma dsl end
11      }
12 }
```

and instruction scheduling on the generated code [7], [8], [9]. These works proposed integrated register allocation and instruction scheduler for VLIW or in-order issue superscalar processors. However, none of these approaches leverage properties of the operators involved in the computations to improve the instruction reordering. We show in Section IV that for many scientific applications executed on out-of-order (OoO) processors, register spills are a performance bottleneck. To this end, we propose **LARS** (**L**ist-based, **A**daptive, **R**egister-reuse-driven **S**cheduler)—a greedy instruction reordering framework for straight-line code, that a) operates at source level to fully exploit the associativity of operations, b) has a much better global perspective of the computational reuse pattern, and c) reorders the instructions minimize MAXLIVE.

III. Instruction Reordering With LARS

A. Preprocessing Steps

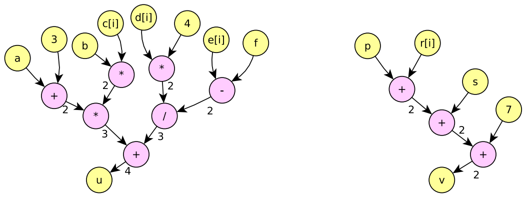
LARS parses statements within a computational loop nest that are expressed in a subset of C with the following restrictions: (1) the loop iterators and the program parameters must be immutable in the statements; (2) the right-hand side expression of a statement must be side-effect free; and (3) the array index expressions in each statement must be an affine function of the loop iterators, program parameters, and literals.

Listing 3 shows an example of preparing an input C code for reordering using LARS. LARS currently uses a pragma-based approach, where the straight-line code of interest is demarcated using `dsl` pragmas. All the auxiliary information is supplied as arguments to the `dsl` pragma. This makes it convenient to use LARS with domain-specific languages that are similar in flavor to C/C++ [10]. Lines 6 and 10 mark the beginning and end, respectively, of the optimization region. The arguments in line 6 indicate that i, j, and k are the loop iterators, and that the demarcated statements need to be unrolled by an unrolling factor of 2 along the dimensions corresponding to iterators k and j.

A preprocessing pass performs loop unrolling by the specified unrolling factors, and then lowers each statement in the optimization region into a sequence of instructions using operator associativity and/or distributivity. The instructions are somewhat similar in spirit to the three-address GIMPLE IR of GCC, where the right-hand side of an instruction has at most two operands, and the operator is either an assignment or an accumulation. In terms of assembly language, these instructions are synonymous with the register-register instructions ($r_1 \leftarrow r_2$ *op* r_3 where r_1 and r_2/r_3 can be the same register).

Figure 1a shows an illustrative computation which is lowered into the instructions shown in Figure 1b using the associativity and distributivity of $+$ and $*$. Note that even though all the operands in the illustrative example are scalars, in practice, the operands can be a mix of array accesses, scalars, and literals. An abstraction commonly used to represent such computations is a computational DAG (CDAG) [11], [12], where the leaf nodes represent the storage locations read, the root represents the output, and the internal nodes represent the operators. For example, Figure 1c shows the CDAG corresponding to the computation of Figure 1b, whereas Figure 1d shows the CDAG corresponding to the instructions after converting the additive contributions in the original DAG to accumulations, represented in the figure by orange "accumulation +" nodes. Throughout the text, we will shift from an instruction sequence to its CDAG abstraction for ease of explanation.

In order to reduce register pressure, LARS must gauge the data reuse between the instructions in the original schedule. To recognize the common uses of a value, the accessed storage locations are assigned a *label*. All the accesses to the same storage location within a statement will have the same label. Across statements, the accesses to a storage location M will be identified by the same label if there is no write to M in between their execution.



(a) Tree of S_1 (b) Tree of S_2

Fig. 2: Expression tree of two statements

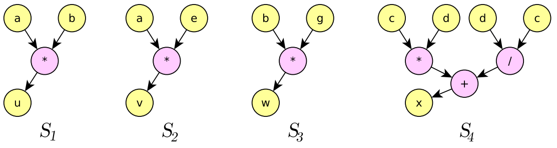


Fig. 3: Evaluating S_4 before $S_{1,2,3}$ reduces register pressure

B. Creating Multiple Initial Schedules

The time taken to reorder instructions is significantly lower with a greedy heuristic than with techniques like dynamic programming [13] or integer linear programming [14]. We use this to our advantage, by reordering multiple versions of the input program instead of just one, and choosing the reordering with the best performance.

Figure 2 shows the expression tree of two statements, the register requirement for which can be computed by the classic Sethi-Ullman algorithm [12]. The numbers next to each node in Figure 2 shows the number of registers required to evaluate that node. There can be two evaluation scenarios: (1) evaluating S_1 before S_2 will require 4 registers, since the 3 registers released after evaluating S_1 can be used to entirely evaluate S_2; (2) evaluating S_2 before S_1 will require 5 registers, since one register will be engaged in holding the result of S_2. This simple example demonstrates that if one rewrites the pragma-demarcated computation as different dependence-preserving permutations of input statements, each permutation can possibly produce a reordering schedule with a different register requirement. Determining the best statement permutations becomes more complicated with an increase in the number of statements, and inter-statement data reuse [15]. For example, each statement of Figure 3, in isolation, would require 3 registers for evaluation: 2 to store the operands, and one to store the result. A register assignment when evaluating the statements in the order shown in Figure 3 is $\{a \rightarrow r_1, b \rightarrow r_2, u \rightarrow r_3, e \rightarrow r_4, v \rightarrow r_1, g \rightarrow r_4, w \rightarrow r_2, c \rightarrow r_4, d \rightarrow r_5, x \rightarrow r_6\}$. However, the permutations that evaluate S_4 first will yield more register-optimal schedules than other permutations. One possible register assignment when S_4 is evaluated first is $\{c \rightarrow r_1, d \rightarrow r_2, x \rightarrow r_3, a \rightarrow r_1, b \rightarrow r_2, u \rightarrow r_4, e \rightarrow r_5, v \rightarrow r_1, g \rightarrow r_5, w \rightarrow r_2\}$, which uses 5 registers instead of 6.

In such situations, a brute-force approach to determine the best permutation will examine all dependence-preserving permutations of the input statements, and apply LARS to each permutation. However, for a program with n independent statements, this would imply exploring $n!$ statement sequences, a formidable task as n increases [15]. Instead, we use the simple clustering algorithm from [16] to generate a few different permutations that cluster the statements with reuses together, and then apply LARS to only these few statement permutations. Once we have the permutations, we choose one permutation at a time, apply the preprocessing step described in Section III-A to it, and then use LARS to reorder the version. The final reordered version is the one that is more efficient in execution. For the rest of the discussion, we will assume that the input to LARS is a valid permutation P.

C. Overview of the Reordering Strategy

Given the initial schedule P, the main objective of LARS is to compute a reordered schedule that preserves the dependences, and reduces register pressure while maintaining sufficient ILP. We use the initial schedule of Figure 1b as an example to give a brief overview of the reordering strategy implemented in LARS. Prior to reordering, we construct a dependence graph (DG) for the initial schedule. Since we allow the contributions to an accumulation node to be in arbitrary order, we do not include the true/output dependences on the accumulation node in DG. Figure 1e shows the dependence graph for the schedule of Figure 1b.

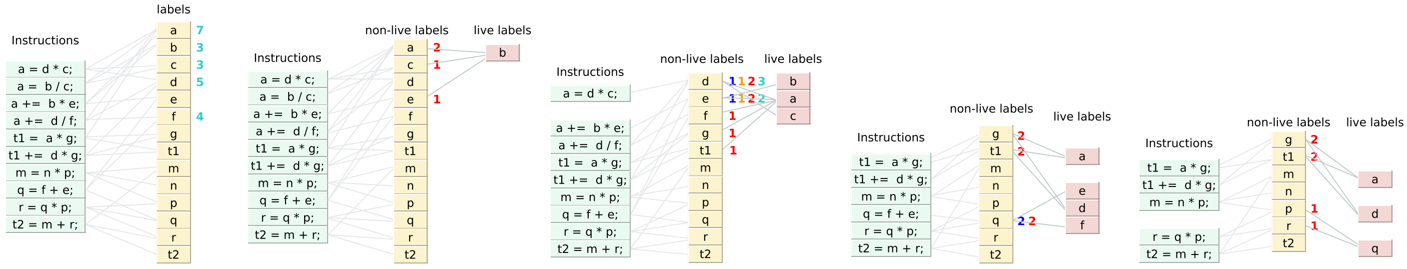
Next, we compute the labels occurring in the initial schedule. Figure 4a shows the mapping from the labels to the instructions in which they appear. We maintain a set L of labels that are currently live in pseudo registers, assuming a *spill-free*[1] model of computation. An instruction I_j in the schedule can fall in one of the following three categories:

- *Blocked:* If I_j has a true dependence on another instruction I_k in the DG, and I_k has not yet been fired
- *Unblocked:* If I_j is not blocked
- *Fireable:* If I_j is unblocked, and all the labels of I_j are in L.

Initially, only the instructions with no incoming dependence edges in DG are unblocked (i.e., instructions 1, 2, 3, 4, 6, 7, and 8 in the schedule of Figure 1b), and none of the labels are live. In a nutshell, the reordering algorithm can be summed up as two iterative steps: make the labels live in some order, so that instructions become fireable (Section III-E), and append the fired instructions into the final reordered schedule (Section III-D). The objective is to minimize the size of L at any given point, since the live range of labels simultaneously live in L are in interference. At the same time, there must be sufficient ILP to tolerate the access/execution latency. The order in which the labels are made live and the fired instructions are appended to the reordered schedule will affect the register pressure and ILP; the heuristic used to determine this order is explained in greater details in this section.

1) Reordering the computational DAG of Figure 1b: In order to determine the first *seed* label that becomes live, we compute an initial priority metric for each label; the computation is described in details in Section III-E1. Simply

[1] A value once loaded in a register will remain so for all its def/uses



(a) Initial schedule (b) Step 1 (c) Step 2 (d) Step 3 (e) Step 4

Fig. 4: Applying LARS reordering strategy to the input of Figure 1b. The digits next to a label are color-coded, and represent metrics that are explained in Section III-E2. green digit: number of non-live labels the label interacts with, red digit: cumulative primary affinity, orange digit: fire potential, blue digit: release potential.

put, the labels occurring in instructions that are closer to the source of the longest (critical) path in the CDAG are assigned a higher initial priority. This is done so that the source instructions in the critical paths are not unnecessarily delayed, resulting in longer schedule length. This also conforms to the priority assignment in the instruction scheduling phase of most production compilers, like GCC. For example, the critical path in the schedule of Figure 1b involves accumulations into a. Since division operation has the highest latency, we assign high initial priority to labels $\{a,b,c,d,f\}$. Among these, LARS chooses b as the seed label, since it interacts with the least number of non-live labels (Figure 4b); the rationale behind this choice is discussed in Section III-E2.

Once a label is made live, we check if any unblocked instruction becomes fireable. If so, the algorithm fires/executes it, and appends it to the final reordered schedule. Otherwise, a cost tuple $\langle t_1, t_2, \ldots, t_k\rangle$ is constructed for each non-live label, the elements of which represent the interaction of the label with the already-live labels, and the effect of making that label live on the current unblocked statements. Element t_i has higher priority than t_j if t_i appears before t_j in the tuple. $t_i(T)$ denotes the value of element t_i in the cost tuple T. Similarly, $t_i(l)$ denotes the value of element t_i in the cost tuple of label l. Cost tuple comparison is based on the lexicographical ordering, i.e., for two tuples T_a and T_b,

$$T_a > T_b \Leftrightarrow \overset{k}{\underset{i=1}{\exists}} \big(\overset{i-1}{\underset{j=1}{\forall}} (t_j(T_a) = t_j(T_b)) \wedge t_i(T_a) > t_i(T_b) \big)$$

Section III-E2 describes the cost tuple in detail. The labels with most profitable (i.e., greatest) cost tuple are assigned highest priority, and are iteratively made live till one or more of the unblocked instructions becomes fireable. In our example, none of the unblocked instructions become fireable when the seed label b is made live, and more labels need to be made live in order to fire any instruction. Figure 4b shows the interaction between the live label b, and the non-live labels $\{a,c,e\}$ that appear with b in instructions $\{2,3\}$, i.e., their live range interferes with b. This interaction is captured by one of the elements, say t_p, in the cost tuple: $t_p(a) = 2$ since the live range of a interferes with b in two instructions, whereas $t_p(c) = t_p(e) = 1$, since the live ranges of c and e interfere with b in only one instruction. Other elements in all the cost tuples being equal, labels a,c,e will have higher priority over other non-live labels due to t_p, and consequently one of these three labels will be added to the live set next; LARS picks a to be added to the live set, since $t_p(a) > t_p(c,e)$.

Once a is live, instruction 2 can become fireable if c is made live next. When recomputing the cost tuple for the non-live labels, this *firing potential* of c is accounted for in its cost tuple by one of the elements, say t_f. Element t_f is positioned in the cost tuple so that it has higher priority than t_p, giving c higher priority over other non-live labels. Therefore c is made live next, and instruction 2 is fired (Figure 4c). At this point, making either label d or e live will result in the firing of exactly one instruction. LARS prioritizes making e live, since it interacts with fewer non-live labels. Note that by doing so, LARS has separated the live range of b from that of d.

Once the computation reaches the state of Figure 4d, making label q live will enable firing of instruction 8. Instruction 8 has the last uses of labels f and e: once fired, the pseudo-register assigned to these two labels can be reused to store other labels. This *release potential* of q is accounted for by an element in its cost tuple, say, t_r. Element t_r is assigned higher priority than t_f in the cost tuple. This results in q getting higher priority over other non-live labels, and becoming live in the next step (Figure 4e). When a label is released, its live range is separated from the non-live labels. LARS thus uses cost tuples to effectively manage the live set so that the live set size (and consequently the live range interference) is reduced.

After an instruction is fired, its dependence edges are removed from DG, which can possibly unblock some instructions. The cost tuple for each non-live label is recomputed with a change in the set of live labels, or the set of unblocked statements. The algorithm terminates when all instructions in the initial schedule have been fired. The final reordered schedule is then printed out using appropriate intrinsics for multi-core CPUs. Algorithm 1 sketches out the high-level reordering algorithm.

D. Adding Instructions to Reordered Schedule

We say that an instruction I_a interlocks with a previously fired instruction I_b, if there is a true dependence from I_b to I_a, or both I_b and I_a contribute to the same accumulator [11]. At any step, if there are several fireable instructions, priority is

```
1.  a   = b / c;
2.  a   += b * e;
3.  a   += d * c;
4.  a   += d / f;
5.  q   = f + e;
6.  t1  = d * g;
7.  t1  += a * g;
8.  r =   q * p;
9.  m =   n * p;
10. t2 = m * r;
```

(a) Final schedule

```
1.  a   = b / c;
2.  a  += b * e;
3.  a1 += d * c;
4.  a1 += d / f;
5.  q   = f + e;
7.  t1  = d * g;
6.  t1 += (a + a1) * g;
8.  r =   q * p;
9.  m =   n * p;
10. t2 = m * r;
```

(b) Final schedule with shadows

Fig. 5: Example: Using shadows to avoid interlocks

given to the one that has minimum interlocks with the recently scheduled instructions. For example, at the stage of Figure 4e, if *t1* and *g* are made live, instructions 5 and 6 become fireable simultaneously. However, the previously fired instruction 4 interlocks with 5 on label *a*. Therefore, 6 is scheduled before 5 (Figure 5a). If all fireable instructions have the same degree of interlock, we fire them in their order in the original schedule. This provides the scheduling phase of the backend compiler with opportunities to exploit ILP [11].

If instructions I_a and I_b interlock due to the same accumulation output, we can resolve the interlock by register renaming [17], a technique often used by both compilers and the hardware to remove false dependences and improve ILP. When firing the instruction I_a, we check if it interlocks due to the accumulation output with any of the k previously fired instructions, where k is a code generator parameter, which we term *interlock window size*. If it does, then we replace the accumulator in I_a with its *shadow*, which will act as a partial accumulator. The shadow accumulator is appropriately initialized to the identity of the accumulation operator (e.g., 0 for addition/subtraction, 1 for multiplication), and its value will be added to the actual accumulator before the subsequent use of the accumulation output. There will be at most k shadows per accumulator, after which they will be cyclically reused. By register renaming, we remove the dependence between instructions that are less that k hops apart in the final schedule, thereby improving ILP. For example, in the reordered schedule of Figure 5a, the true dependences due to the accumulation output *a* in instructions 1–4 limits the achieved ILP. We can increase ILP by creating a shadow accumulator *a1*, and collecting partial accumulations from alternate instructions into it, as shown in Figure 5b. The partial contribution from the shadow *a1* is added to *a* before its subsequent use in instruction 6.

E. Adding Labels to Live Set

1) Assigning initial priority to labels: No labels are initially live. To start the reordering, we have to choose the first non-live *seed label*, and make it live. In order to determine the seed label, we assign an initial priority to all labels. This priority is based on the concept of *earliest starting time*. For each instruction n in the initial schedule, the earliest starting time, $EST(n)$, can be computed based on the pipeline latencies of the underlying architecture [18].

$$EST(n) = \max_i(EST(p_i) + latency(p_i)) \qquad (1)$$

Algorithm 1: Reorder (P, k)

```
Input : $P$: Initial schedule, $k$: Interlock window size
Output: $R$: Reordered schedule
$DG \leftarrow$ dependence graph for $P$;
$L \leftarrow \emptyset$; // set of live labels
$S_{ub} \leftarrow$ instructions that have no incoming dependences in DG;
$S_{fire} \leftarrow \emptyset$; // set of fireable instructions
compute $EST(n_i)$ for each instruction $n_i$ using equation 1;
for each label $l$ in $P$ do
    compute initial priority of $l$ using $EST$ of instructions (Sec. III-E1)
while not all instructions in $P$ are fired do
    $M \leftarrow \emptyset$; // list of cost tuple for each non-live label
    for each label $l \notin L$ such that $l$ occurs in an instruction in $S_{ub}$ do
        $M \leftarrow M \cup$ Create-Tuple $(l, L, S_{ub})$; (Algo. 2)
    Adaptive-Custom-Sort $(M)$ (Sec. III-F);
    add the most profitable label in sorted $M$ to $L$;
    for each instruction $s$ in $S_{ub}$ do
        if all the labels of $s$ are in $L$ then
            $S_{fire} \leftarrow S_{fire} \cup s$;
            $S_{ub} \leftarrow S_{ub} \setminus s$;
    while $S_{fire} \neq \emptyset$ do
        remove an instruction $s \in S_{fire}$ that has the least interlocks with the
        instructions in $R$;
        for each label $l$ in $s$ do
            if this is the last use of $l$, remove $l$ from $L$;
        if $s$ contributes to an accumulation node $t$ then
            create a shadow accumulation output if any of the last $k$
            instructions in $R$ also contribute to $t$;
        append $s$ to $R$;
        remove the dependence edges from $s$ in $DG$, add instructions with no
        incoming dependences in $DG$ to $S_{ub}$;
```

where p_i is the i^{th} predecessor of n in DG, and $latency(p_i)$ is the latency of the dependence edge from p_i to n.

A source in the DG is a node with no predecessors, and a sink is a node with no successors. For a path p in DG starting at source node n_s and ending at sink node n_t, we define the path execution time, $PET(p) = EST(n_t)$. A *critical path* in DG corresponding to V is defined as the source-to-sink path with the highest PET. There can be more than one critical paths in the program. In most instruction scheduling algorithms, the first scheduled instruction (*seed instruction*) is the source instruction in the critical path [18], [11].

If an instruction n occurs in k source-to-sink paths $\{p_1, p_2, \ldots, p_k\}$ in the DG, we can compute a cumulative cost for n, similar to the one defined in [11], as $\max_{1 \leq r \leq k}(PET(p_r) - EST(n))$. Thus, the source in the critical path will have the highest cumulative cost, and the instructions closer to the source instruction in relatively longer paths in the DG will have higher cumulative costs than other instructions. The labels occurring in an instruction with higher cumulative cost must be assigned a higher priority, so that such instructions are scheduled with urgency. The initial priority P_l of a label l is therefore set to $\max_i\{cumulative\ cost(n_i)\}$, where n_i represents all instructions in which label l appears.

2) Creating a cost tuple for each label: To model the profitability of making a non-live label l live, we capture its impact on the current set of unblocked instructions and its interaction with the live labels $\in L$ using the following metrics:

– Fire potential (F_{pot}): The fire potential of label l is the number of instructions that become fireable upon making l live. If there are k unblocked instructions that only need l to become live in order to become fireable, then the fire

Algorithm 2: Create-Tuple (l, L, S_{ub})

```
Input  : $L$: Set of live labels, $l$: a label $\notin L$, $S_{ub}$: Unblocked instructions
Output: $T$: Cost tuple for $l$
$R_{pot} \leftarrow 0$; $F_{pot} \leftarrow 0$; $N_{lk} \leftarrow 0$;
for each $s \in S_{ub}$ that becomes fireable when $l \in L$ do
    increment $F_{pot}$;
    for each $t$ in $L$ that has the last use in $s$ do
        increment $R_{pot}$;
$P_l \leftarrow$ the label priority of $l$ based on its initial priority, and the current leading
statement;
$N_{npaff} \leftarrow$ number of labels $\notin L$ that have non-zero primary affinity with $l$;
$P_{aff} \leftarrow$ cumulative primary affinity of $l$ to the labels $\in L$;
$S_{aff} \leftarrow$ cumulative secondary affinity of $l$ to the labels $\in L$;
$T \leftarrow \langle R_{pot}, F_{pot}, P_{aff}, S_{aff}, -N_{npaff}, P_l \rangle$;
return $T$;
```

potential of l is k.

- Release potential (R_{pot}): The release potential of l refers to the number of labels that have their last uses in the instructions that become fireable when l is made live.
- Cumulative Primary affinity (P_{aff}): Label l has primary affinity of strength p with an already live label t if both l and t occur simultaneously as the labels in exactly p instructions. The cumulative primary affinity of l is the sum of its primary affinity with all live labels.
- Cumulative Secondary affinity (S_{aff}): Label l has secondary affinity of strength s with an already live label t if they both have a non-zero primary affinity with exactly s common labels. The cumulative secondary affinity of l is the sum of its secondary affinity with all live labels.

Rationale: The release potential of a label reflects its ability to reduce register pressure directly. Therefore, the label with highest release potential must always be made live before others. Making a label with high fire potential live can provide the compiler with independent instructions to schedule. Also, even if the fired instructions do not release registers, they may still be a step in decreasing the remaining uses of their operands, and towards a consequent release of the registers occupied by their operands. If a label l has a non-zero primary affinity with an already-live label t, then the live range of l and t definitely interfere, and t cannot be released till l becomes live. Therefore, if label l has a high cumulative primary affinity with the already-live labels, then we eagerly make l live, so that we get a step closer to releasing l, and the labels it has primary affinity with. Similarly, if a label l has a non-zero secondary affinity with an already-live label t due to a label m, then the live ranges of both l and t will interfere with m. Making l live will shrink the live range of m and bring us one step closer to releasing m.

To help us determine the seed label, we had initialized the initial priority P_l for each label based on the cumulative cost of the instructions. Multiple labels along disjoint critical paths could be initialized with the same priority if those paths have the same PET. However, when an instruction n is fired by LARS, the priority of the labels in the instructions dependent on n must be increased to favor depth-first traversal of the CDAG containing n. Otherwise, the instructions along different CDAGs may get unnecessarily interleaved, increasing the register pressure [12]. Whenever an instruction is fired, the priority of each label is updated based on the current *leading statement*. A statement S_i is currently leading if maximum number of the instructions that it was lowered down to have been fired. All the non-live labels occurring in the unblocked instructions corresponding to S_i are assigned a higher P_l than other non-live labels.

The primary affinity of a non-live label l to other non-live labels is also an important metric. Suppose the label l has a non-zero primary affinity with r non-live labels. Then, if l is made live at the current step, it needs to be live until all the r labels are live as well. For each label l, $N_{npaff}(l)$ denotes the number of non-live labels with which l has non-zero primary affinity. A label with higher N_{npaff} must be penalized, as making it live may increase live-range interference in future.

The resultant combined metric for each label can be thought of as a 6-tuple: $\langle R_{pot}, F_{pot}, P_{aff}, S_{aff}, P_l, -N_{npaff} \rangle$. The collection of tuples for all labels is then lexicographically sorted in descending order to rank-order the labels based on their profitability to be made live. The label that is deemed most profitable after the rank-ordering is chosen to be added to the set L. Algorithm 2 depicts the creation of the cost tuple.

F. Adaptivity in LARS

The relative order of the metrics in the tuple affects the reordering objective. For example, assigning the highest priority to R_{pot} followed by P_l would allow an interleaved execution of two statements only when the interleaving results in release of registers. This provides us the flexibility to define multiple sort functions, each acting as an objective function to determine the sequence in which the labels are added to set L. In the implementation, we choose the objective function based on the nature of the computation. For the computations where the intra-statement reuse is less than half the inter-statement reuse (e.g., computation of Figure 6e), we order the metrics as $\langle R_{pot}, F_{pot}, P_{aff}, S_{aff}, -N_{npaff}, P_l \rangle$. This facilitates interleaving across the statements to reduce register pressure. However, if the inter-statement reuse is less than one-third the intra-statement reuse, we order the metrics as $\langle R_{pot}, P_l, F_{pot}, P_{aff}, S_{aff}, -N_{npaff} \rangle$, so that the interleaving across CDAGs is minimized.

G. Putting It All Together

Algorithm 3 recapitulates the broad steps applied by LARS to reorder an input program M. The unrolling factors are applied to M to obtain an unrolled version M'. The dependence graph is then computed for M' (line 4) using which, multiple permutations $\{V\}$ of M' are created (line 5). The reordering heuristic is then applied to each of these versions (line 7). The end goal is to find the fastest amongst all versions, which is then returned as the final reordered code (lines 8–11). Appendix A gives a detailed example of using LARS to reorder the 2D 9-point stencil.

IV. Experimental Evaluation

Experimental Setup: The experimental results presented here were obtained on an Intel Xeon Phi and a Skylake i7-6770K processor. The hardware details are shown in Table

Algorithm 3: End-to-end Algorithm

```
Input : $M$: Input program, $uf$: Unrolling factors
Output: $R$: Reordered output
$T_m \leftarrow \infty$;
$k \leftarrow$ user-specified interlock window size;
$M' \leftarrow$ Unroll ($M, uf$);
$G' \leftarrow$ Dependence-Graph ($M'$);
$\{V\} \leftarrow$ Create-Permutations ($M', G'$); (Section III-B)
for $P \in \{V\}$ do
    $r \leftarrow$ Reorder ($P, k$);
    $t \leftarrow$ execution time of $r$;
    $T_m \leftarrow$ min ($T_m, t$);
    $R \leftarrow$ faster version between $r$ and the one corresponding to $T_m$;
return $R$;
```

Resource	Details
Intel multi-core CPU	Intel core i7-6700K (4 cores, 2 threads/core 4.00 GHz, 8192K L3 cache)
Intel Xeon phi	Intel Xeon Phi 7250 (68 cores, 4 threads/core 1.40GHz, 1024K L2 cache)

TABLE I: Benchmarking hardware

Compiler	Flags
gcc	-Ofast/Os -fopenmp -ffast-math -fstrict-aliasing -march=core-avx2/{knl -mavx512f -mavx512er -mavx512cd -mavx512pf} -mfma -f(no)schedule-insns -fschedule-insns2 -fsched-pressure -ffp-contract=fast
llvm	-Ofast/Os -fopenmp=libomp -mfma -ffast-math -fstrict-aliasing -march=core-avx2/knl -mllvm -ffp-contract=fast -pre-RA-sched="source/list-ilp/list-hybrid/list-burr"

TABLE II: Compilation flags for multi-core and Xeon Phi

I. All benchmark codes are compiled with GCC-7.2.0, and the version of LLVM from the svn (llvm/trunk 311831). The compilation flags for both compilers are listed in Table II. Both GCC and LLVM provide some fine-grain control over the instruction scheduling passes via the compilation flags: GCC allows the user to enable or disable the prepass and postpass instruction scheduler, whereas LLVM provides multiple implementations for the prepass scheduler. With the fine-grain access to the compiler passes, we can control the effect of the scheduling passes on the LARS reordered schedule.

Benchmarks: We evaluate the efficacy of LARS on a wide variety of benchmarks, which can be grouped into five sets. The first set comprises generalized versions of computations typically used in iterative processes such as solving partial differential equations and convolutions [19]. The second set contains smoothers used in HPGMG benchmark suite [20]. The third set includes high-order stencil computations from the ExpCNS Compressible Navier-Stokes mini application from DoE [21] and the Geodynamics Seismic Wave SW4 application code [22]. The fourth set comprises the kernels from Cloverleaf benchmark suite [23]. The final set contains the tensor contraction kernels CCSD(T) from the NWChem suite [24]. These benchmarks are listed in Table III. All benchmarks are double-precision. Note that we evaluate all compute kernels from Cloverleaf benchmark suite, excluding the trivial kernels that copy data from one array to another.

Code Generation: The original version (*Original*) for each benchmark is as written by application developers, parallelized and vectorized by adding appropriate OpenMP pragma to the outermost parallel loop, and SIMD pragma to the innermost vectorizable loop, but without any explicit loop unrolling.

Benchmark	N	FPP	R	Benchmark	N	FPP	R
2d25pt	8192^2	49	2	cell-advec 3D	256^3	49	16
2d49pt	8192^2	97	2	mom-advec 3D	256^3	55	15
2d64pt	8192^2	127	2	acceleration 3D	256^3	57	13
2d81pt	8192^2	161	2	ideal-gas 3D	256^3	12	4
2d121pt	8192^2	241	2	PdV 3D	256^3	98	16
3d27pt	512^3	53	2	calc-dt 3D	256^3	67	11
3d125pt	512^3	249	2	fluxes 3D	256^3	30	12
chebyshev	512^3	39	6	viscosity 3D	256^3	139	9
7-point	512^3	11	2	cell-advec 2D	4096^2	47	14
poisson	512^3	21	2	mom-advec 2D	4096^2	41	13
helmholtz-v2	512^3	22	7	acceleration 2D	4096^2	38	10
helmholtz-v4	512^3	115	7	ideal-gas 2D	4096^2	12	4
27-point	512^3	30	2	PdV 2D	4096^2	51	13
hypterm	300^3	358	13	calc-dt 2D	4096^2	36	9
diffterm	300^3	415	11	fluxes 2D	4096^2	12	8
rhs4th3fort	300^3	687	11	viscosity 2D	4096^2	58	7
derivative	300^3	486	10	sd-t-d1-{1:9}	24^8	2	3

N: Domain Size, FPP: FLOPs per Point R: Arrays Accessed

TABLE III: Benchmark characteristics

We generate multiple unrolled versions (*Unrolled*) for each benchmark by explicitly unrolling all but the innermost loop by powers of 2, restricting the maximum unroll factor to 8. For each unrolled version, we generate an accumulation version with all the additive contributions converted to accumulations (*Accumulation*). Corresponding to each unrolled version, we create a LARS-optimized version (*Reordered*) as well. At present, LARS does not autotune for optimal unrolling factors. The user only needs to specify the unrolling factors in the pragma (line 6 of Listing 3), and appropriately modify the increment expression of the computational loop nest (lines 3–5 of Listing 3). The unrolling of the statements is done by the preprocessing stage described in Section III-A, usually in under less than a second. All benchmarks are control-flow-free and conform to the restrictions described in Section III-A, and hence LARS is able to parse them without any code modification. Whenever possible, we set the unrolling factor along the fastest-varying dimension to 1 to allow efficient vectorization by the underlying compiler. Except for the CCSD(T) kernels, LARS generates a reordered schedule with appropriate SIMD intrinsics for multi-core CPU and Xeon Phi. We do not generate accumulation version for CCSD(T) benchmarks, since their unrolled versions are already in accumulation form. No other optimization (e.g. tiling) is applied to the reordered code, in order to get a fair performance comparison with respect to the unrolled versions. For code generation, the interlock window size, k, is set to 2. For all benchmarks, the reordered versions were generated under 15 seconds by LARS. We check the correctness of each reordered code against the original version. For all benchmarks, we consider the theoretical floating-point operations performed by the original version to compute the GFLOPS for all versions.

Performance Results: Table IV plots the performance of the original, unrolled, accumulation, and reordered code for the different benchmark sets. For the convolution kernels, the reordered version significantly outperforms the original and unrolled versions over all compilers/architectures, especially when the *order* of the computation increases. The order here refers to the extent of elements read from the center. For

Benchmark	GCC on i7-6700K				LLVM on i7-6700K				GCC on Xeon Phi 7250				LLVM on Xeon Phi 7250			
	Org	Unr	Acc	Reo	Org	Unr	Acc	Reo	Org	Unr	Acc	Reo	Org	Unr	Acc	Reo
2d25pt	53.74	50.65	54.18	55.11	55.16	51.82	51.42	55.85	68.73	90.19	91.17	95.35	64.55	74.59	87.62	95.71
2d49pt	70.90	70.71	67.96	100.58	75.37	20.09	72.91	99.08	89.79	146.69	144.48	170.76	87.16	114.97	128.73	151.20
2d64pt	66.65	69.20	66.34	112.12	71.67	19.16	64.88	120.62	104.67	167.75	171.94	184.56	92.18	106.47	146.14	191.74
2d81pt	72.18	69.97	66.23	137.36	68.15	18.28	17.90	137.43	109.23	222.13	231.01	278.38	85.74	116.12	195.79	293.93
2d121pt	72.33	68.31	64.98	163.90	25.58	23.48	65.25	175.13	77.42	102.81	293.17	392.13	94.99	106.01	223.98	367.03
3d27pt	37.12	45.71	43.31	52.33	37.66	19.95	42.28	51.38	79.60	104.08	106.78	118.08	87.90	101.23	116.90	147.48
3d125pt	53.01	44.85	55.55	109.02	17.90	17.07	46.99	115.15	146.18	200.65	235.35	354.90	100.03	97.88	217.36	364.21
chebyshev	16.32	17.15	18.13	19.54	16.62	18.08	18.44	19.56	30.54	42.17	40.94	46.70	26.81	45.35	46.54	52.13
7-point	7.35	10.26	10.19	10.46	7.49	10.81	10.53	11.61	18.28	24.11	24.16	24.78	18.37	27.78	27.13	28.88
poisson	14.31	19.09	19.38	19.97	14.56	16.87	17.91	19.80	31.05	41.05	41.54	45.48	28.28	42.41	50.49	57.15
helmholtz-v2	6.67	8.30	8.30	8.64	6.82	7.76	8.13	8.65	15.89	18.46	18.96	19.98	14.36	18.55	19.76	21.81
helmholtz-v4	23.40	24.92	26.27	27.80	24.13	19.05	25.91	27.95	39.61	56.12	50.21	60.87	37.36	33.34	58.44	75.87
27-point	20.24	25.36	25.30	28.75	20.89	25.59	22.13	27.71	42.77	58.41	54.78	62.05	39.57	68.13	69.16	83.39
hypterm	8.77	-	11.86	14.37	8.92	-	11.09	14.72	24.63	-	32.73	42.88	29.36	-	33.38	41.14
diffterm	9.61	-	11.58	14.60	4.79	-	9.90	14.64	32.29	-	39.21	43.73	30.94	-	36.11	44.26
rhs4th3fort	43.63	-	45.22	56.64	32.66	-	40.43	57.83	115.23	-	129.01	159.61	126.28	-	130.21	166.02
derivative	20.14	-	22.95	26.84	24.96	-	25.72	28.26	82.49	-	88.90	101.54	71.93	-	78.17	87.94
cell-advec 3D	4.24	4.24	4.13	4.36	3.95	4.24	4.21	4.36	10.85	9.28	9.17	12.46	8.79	9.67	10.33	13.28
mom-advec 3D	5.42	5.56	5.58	5.84	5.63	5.82	5.77	5.88	14.05	12.66	12.78	16.74	11.88	11.60	13.71	17.81
accelerate 3D	7.82	8.51	8.19	9.63	8.08	8.08	8.13	9.21	15.70	16.58	18.77	22.79	13.77	13.79	19.65	27.83
ideal-gas 3D	6.49	6.34	6.37	6.64	6.58	6.46	6.41	6.65	11.58	12.45	13.95	19.65	8.56	9.44	13.19	19.72
PdV 3D	15.08	15.69	14.91	15.84	15.28	14.85	14.71	15.86	35.44	32.70	35.31	41.56	29.85	36.29	41.91	50.39
calc-dt 3D	14.10	14.98	14.71	15.30	14.31	15.11	15.23	15.45	29.30	30.69	30.69	43.08	28.68	29.42	33.27	47.60
fluxes 3D	5.25	5.71	5.84	5.96	5.33	5.53	5.57	5.92	13.98	14.13	13.91	17.06	12.02	12.13	13.31	18.43
viscosity 3D	37.35	44.93	45.77	50.54	37.80	41.76	43.37	56.57	77.31	83.64	90.76	114.80	64.39	89.66	133.81	179.02
cell-advec 2D	6.42	6.24	6.17	6.70	6.60	6.45	6.17	6.70	12.35	12.52	13.71	15.40	10.52	10.11	11.91	15.83
mom-advec 2D	4.95	4.80	4.91	5.03	4.92	4.77	4.97	5.05	10.63	10.81	11.31	13.29	8.74	8.84	9.93	13.35
accelerate 2D	9.84	9.72	9.83	10.22	9.86	8.25	9.13	10.22	14.77	16.38	22.81	26.54	10.39	8.02	17.91	27.52
ideal-gas 2D	6.78	6.46	6.79	6.81	6.65	6.41	6.71	6.80	11.49	11.53	14.43	19.76	8.00	8.25	13.77	19.87
PdV 2D	12.32	12.39	12.10	12.67	12.42	12.18	11.71	12.58	22.97	26.31	27.21	34.59	19.41	20.98	22.63	34.50
calc-dt 2D	11.69	11.52	11.97	12.31	11.62	11.52	11.71	11.90	23.52	24.64	28.14	33.56	14.93	16.06	21.61	30.47
fluxes 2D	3.79	3.94	3.33	4.05	3.87	3.88	3.78	3.96	5.09	6.00	5.91	7.09	5.68	5.27	7.73	11.23
viscosity 2D	31.67	30.85	31.55	32.44	31.26	19.41	21.66	32.46	46.09	49.39	54.17	71.44	31.61	34.98	44.87	72.88
sd-t-d1-1	18.55	28.38	-	40.04	7.46	30.80	-	46.28	42.61	55.04	-	69.24	19.76	97.58	-	140.07
sd-t-d1-2	18.01	32.77	-	35.85	7.52	37.74	-	41.46	45.01	60.04	-	91.41	19.95	95.95	-	138.06
sd-t-d1-3	15.19	22.16	-	41.28	7.45	23.92	-	52.87	35.82	82.21	-	178.44	17.06	83.94	-	190.03
sd-t-d1-4	17.64	37.26	-	43.24	7.27	34.08	-	40.68	43.65	84.65	-	111.85	19.33	115.11	-	142.16
sd-t-d1-5	17.91	47.32	-	64.26	7.06	47.02	-	70.78	45.22	127.96	-	201.84	19.25	156.54	-	211.01
sd-t-d1-6	15.27	22.58	-	41.01	7.39	22.96	-	55.74	35.88	82.42	-	180.56	17.04	81.66	-	191.43
sd-t-d1-7	18.82	39.93	-	46.93	7.50	44.07	-	57.39	43.76	85.71	-	113.19	19.59	148.48	-	216.17
sd-t-d1-8	18.23	47.60	-	71.72	7.45	46.73	-	68.62	45.43	130.55	-	203.06	19.54	146.15	-	214.35
sd-t-d1-9	11.17	25.18	-	36.61	6.79	30.09	-	52.83	34.79	83.88	-	148.35	17.66	99.49	-	190.90

Org: original, Unr: unrolled, Acc: accumulation, Reo: reordered

TABLE IV: Performance of the benchmarks (in GFLOPS) on different architectures

example, the *2d21pt* kernel benefits the most from reordering, and it has the highest order, 5. This observation is consistent with that of Stock et al. [1]. The volume of data read per point usually increases with an increase in the order for most of the dense computations, and this can exacerbate the spills in the unrolled code. LARS exploits associativity to judiciously reorder the instructions by interleaving computations across the unrolled statements, thereby reducing the spill volume. The reordered version also outperforms the accumulation version, indicating that reordering the instructions after rewriting the computation in accumulation form is crucial to performance.

LARS outperforms both original and unrolled versions for the HPGMG smoothers as well, but the performance gains are lower when compared to the convolution kernels due to two reasons: (a) LARS leverages operator distributivity in order to exploit associativity for smoothers, thereby increasing the computation; (b) the increase in arithmetic intensity (defined as the FLOPs relative to the memory accesses) with unrolling is less significant for smoothers, which implies that they are more bandwidth-bound than convolutions.

For Cloverleaf 3D and 2D benchmarks, the performance of LARS-reordered code is almost the same (1.0× – 1.13× for multi-core CPU), or slightly better (1.14× – 1.7× with GCC on Xeon Phi). This can be mainly attributed to the nature of the computations in the Cloverleaf suite. As observed from Table III, each kernel in the Cloverleaf benchmarks reads from a multitude of arrays, but the computations in each kernel is scarce. Thus, there is very little reuse to be exploited via unrolling, and the computation is severely bandwidth-bound. Both these factors reduce the benefits of unrolling and reordering. However, the performance results serve to demonstrate that the reordering done by LARS does not degrade the performance for such benchmarks.

High-order stencils are becoming commonplace in scientific simulations, and optimizing them is the current focus of the HPC community. These high-order stencils can be thought of as a forest of CDAGs, with high data reuse across the CDAGs. This abstraction is particularly challenging for a traditional compiler to optimize, since most of the integrated instruction schedulers are designed to schedule a single CDAG. With LARS, we were able to reduce the total memory accesses per point even with the non-unrolled version, and achieve a

1.22× – 3× speedup over the base version.

A similar performance trend is observed for CCSD(T) tensor contractions. Since each contraction has a loop nest of depth 7 and loop unrolling in tensor contractions is a research problem in itself [25], we fix two loops as the unrolling candidates. Since the computation is an accumulation, the unrolling candidates are chosen to favor the reuse of the output element. Unrolling greatly improves the performance, and we get a 1.09× – 2.26× speedup with simple reordering of the instructions to reduce the memory accesses.

Appendix B presents an analysis of the generated assembly code on Xeon Phi 7250, confirming that the LARS-reordered version incurs far fewer memory accesses than the unrolled version.

V. Related Work

The interplay of register allocation and instruction scheduling has been studied by a body of prior research [26], [7], [8], [9], [27], [11]. Sethi et al. [12] propose an algorithm to translate an expression tree into machine code using optimal number of registers. The algorithm does not extend to CDAGs. Goodman and Hsu [11] present a prepass scheduler that is register-pressure sensitive. Motwani et al. [26] show that integrated register allocation and instruction scheduling is NP-hard, and propose a combined heuristic that provides relative weights for controlling register pressure and instruction parallelism. Other works attempt to combine the data dependence graph on which instruction scheduling is performed and interference graph on which register allocation is performed, and then perform instruction scheduling and register allocation on the unified graph. Berson et al. [9] use register reuse DAGs to identify instructions whose parallel scheduling will require more resources than available, and optimize them to reduce their resource demands. Pinter [7] describes an algorithm that colors a parallel interference graph to obtain a register allocation that does not introduce false dependences, and therefore exploits maximal parallelism. Norris et al. [8] propose an algorithm that constructs an interference graph with all feasible schedules, and then remove interferences for schedules that are to be least likely followed. All these approaches are designed for in-order or VLIW processors, and an experimental evaluation by Valluri et al. [28] show that integrated efforts rarely benefit OoO processors.

In the context of OoO processors, Barany and Krall [27] propose an optimistic integrated approach that performs prepass scheduling, and then rearranges instructions to mitigate register pressure during register allocation. However, the rescheduling is done from the local perspective of a single instruction, and tends to reduce ILP. Silvera et al. [29] propose an instruction reordering framework for dynamic issue hardware that takes the output schedule after prepass scheduling, and reorders the instructions within the instruction window to reduce register pressure. With such constraints on reordering, their method will not be able to do aggressive reordering that reduces register pressure while maintaining ILP, as done by LARS (Appendix A). While Barany and Krall [27] note that their rescheduling may not reduce register pressure as much as a register-pressure-driven scheduler, both they and Silvera et al. [29] emphasize the importance of reducing register spills in OoO processors.

Stock et al. [1] leverage retiming to convert a gather-gather stencil into a scatter-gather stencil, thereby relieving register pressure. However, their approach is only applicable to regular stencils. In contrast, our work proposes a generalized reordering strategy that uses affinity between labels and unblocked instructions to reduce register pressure; it is not restricted to the cases that Stock et al. handle. Domagala et al. [30] present a register allocation strategy that is cognizant of loop unrolling and instruction scheduling. However, it does not consider associative reordering to improve register allocation/instruction scheduling. Veras et al. [31] identify instances where manual instruction selection and scheduling can boost the performance of compute-bound numerical kernels like matrix-matrix multiplication, and propose a technique that uses custom macro intrinsics to generate efficient assembly for such numerical codelets. Even though their work, like ours, highlights the same performance issues with instruction scheduling of GCC, their approach is only suitable for numerical kernels that are tiled to expose repeated application of a compact codelet. Recently, Rawat et al. [16] proposed an associative reordering strategy targeting stencil computations on GPU. They represent the stencil computation as a DAG of expression trees, and then generalize the Sethi-Ullman algorithm [12] to schedule the DAG with the objective of minimizing register pressure. In contrast, we present a more generalized scheme that does not rely on any special abstraction of computation, and is adapted to generate efficient reordered code for multi-core CPUs.

VI. Conclusion

Register spills and low performance are a problem when compiling scientific codes with high degree of many-to-many reuse. We present LARS, a list-based, adaptive, register pressure driven source-level scheduler, that leverages operator associativity to reorder instructions to reduce register pressure. The metrics used by LARS are powerful enough to exploit patterns in the computation, and general enough to benefit a variety of input applications. We demonstrate the usability of LARS, and the effectiveness of its reordering heuristics, over several benchmarks using multiple compilers and architectures.

Acknowledgments

We thank the anonymous reviewers for their feedback and suggestions that helped improve the paper. This work was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and by the U.S. National Science Foundation (NSF) through awards 1440749 and 1513120.

References

[1] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, "A framework for enhancing data reuse via associative reordering," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014.

[2] G. J. Chaitin, "Register allocation & spilling via graph coloring," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN '82. New York, NY, USA: ACM, 1982.

[3] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Trans. Program. Lang. Syst.*, Sep. 1999.

[4] "Semantics of Floating Point Math in GCC," 2007. [Online]. Available: https://gcc.gnu.org/wiki/FloatingPointMath

[5] P. Basu, M. Hall, S. Williams, B. V. Straalen, L. Oliker, and P. Colella, "Compiler-directed transformation for higher-order stencils," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, May 2015.

[6] P. Suriana, A. Adams, and S. Kamil, "Parallel associative reductions in halide," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. Piscataway, NJ, USA: IEEE Press, 2017.

[7] S. S. Pinter, "Register allocation with instruction scheduling," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, ser. PLDI '93. New York, NY, USA: ACM, 1993.

[8] C. Norris and L. L. Pollock, "A scheduler-sensitive global register allocator," in *Supercomputing '93. Proceedings*, Nov 1993.

[9] D. A. Berson, R. Gupta, and M. L. Soffa, "Integrated instruction scheduling and register allocation techniques," in *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, ser. LCPC '98. London, UK, UK: Springer-Verlag, 1999.

[10] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Resource conscious reuse-driven tiling for GPUs," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. ACM, 2016.

[11] J. R. Goodman and W.-C. Hsu, "Code scheduling and register allocation in large basic blocks," in *Proceedings of the 2Nd International Conference on Supercomputing*, ser. ICS '88. New York, NY, USA: ACM, 1988.

[12] R. Sethi and J. D. Ullman, "The generation of optimal code for arithmetic expressions," *J. ACM*, Oct. 1970.

[13] L. Gan, H. Fu, W. Xue, Y. Xu, C. Yang, X. Wang, Z. Lv, Y. You, G. Yang, and K. Ou, "Scaling and analyzing the stencil performance on multi-core and many-core architectures," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Dec 2014.

[14] C.-M. Chang, C.-M. Chen, and C.-T. King, "Using integer linear programming for instruction scheduling and register allocation in multi-issue processors," *Computers and Mathematics with Applications*, 1997.

[15] A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code generation for expressions with common subexpressions," *J. ACM*, vol. 24, no. 1, pp. 146–160, 1977.

[16] P. S. Rawat, A. Sukumaran-Rajam, A. Rountev, F. Rastello, L.-N. Pouchet, and P. Sadayappan, "Register optimizations for stencils on GPUs," in *Proceedings of the 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. ACM, 2018.

[17] J. E. Smith and A. R. Pleszkun, "Implementing precise interrupts in pipelined processors," *IEEE Trans. Comput.*, May 1988.

[18] V. Sarkar, M. J. Serrano, and B. B. Simons, "Register-sensitive selection, duplication, and sequencing of instructions," in *Proceedings of the 15th International Conference on Supercomputing*, ser. ICS '01. New York, NY, USA: ACM, 2001.

[19] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. ACM, 2009.

[20] "High-Performance Geometric Multigrid," https://hpgmg.org/, 2016.

[21] "ExaCT: Center for Exascale Simulation of Combustion in Turbulence: Proxy App Software," https://exactcodesign.org/proxy-app-software/, 2013.

[22] "Seismic Wave Modelling (SW4) - Computational Infrastructure for Geodynamics," https://geodynamics.org/cig/software/sw4/, 2014.

[23] A. C. Mallinson, D. A. Beckingsale, W. P. Gaudin, J. A. Herdman, J. M. Levesque, and S. A. Jarvis, "Cloverleaf: Preparing hydrodynamics codes for exascale," 2013.

[24] "NWChem Download," 2017. [Online]. Available: http://www.nwchem-sw.org/index.php/Download

[25] W. Ma, S. Krishnamoorthy, O. Villa, K. Kowalski, and G. Agrawal, "Optimizing tensor contraction expressions for hybrid cpu-gpu execution," *Cluster Computing*, Mar 2013.

[26] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen, "Combining register allocation and instruction scheduling," Stanford, CA, USA, Tech. Rep., 1995.

[27] A. Krall and G. Barany, "Optimistic integrated instruction scheduling and register allocation," ser. CPC '10. John Wiley & Sons, Ltd, 2010.

[28] M. G. Valluri and R. Govindarajan, "Evaluating register allocation and instruction scheduling techniques in out-of-order issue processors," in *1999 International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[29] R. Silvera, J. Wang, G. R. Gao, and R. Govindarajan, "A register pressure sensitive instruction scheduler for dynamic issue processors," in *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques*, Nov 1997.

[30] L. Domagala, D. van Amstel, F. Rastello, and P. Sadayappan, "Register allocation and promotion through combined instruction scheduling and loop unrolling," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: ACM, 2016.

[31] R. Veras, D. T. Popovici, T. M. Low, and F. Franchetti, "Compilers, hands-off my hands-on optimizations," in *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, ser. WPMVP '16. New York, NY, USA: ACM, 2016.

Appendix A
Example: LARS in action

We demonstrate the power of LARS by using it to reorder a Jacobi stencil computation [1]. We assume that the interlock window size is 1. The computation has only inter-statement reuse, and the cost-tuple for labels are sorted as described in Section III-F.

Figure 6e shows a 4-way unrolled version of the 2D 9-point Jacobi stencil. We assume that the coefficients in the stencil are literals, and all the input contributions to outputs are additive. There are 4 expression trees, each corresponding to the computation of a single output point, and there is significant reuse between these trees: two consecutive output points reuse 6 input values, and the output points two hops away reuse 3 input values. For the ease of description, let us assume that each input (output) point in Figure 6e is assigned an integer identifier which starts from 1, and is incremented during the lexicographical scan, going from left-to-right, top-to-bottom. We identify an input or output label as i_v or o_v, respectively, where v is its identifier. All the labels have the same initial priority.

LARS starts by scheduling the solo contributions from the inputs i_1, i_7 and i_{13} that contribute to o_1, since these labels can be immediately released (Figure 6a). Next, i_2 is made live, since it has firing potential of 1, and has the lowest primary affinity to other non-live labels. Now, making o_2 live will release i_2 (Figure 6b). i_{14} is made live next, as it will be released after making its contributions to o_1 and o_2. i_8 will be made live next, forcing o_3 to become live in the next step. At this point, we make i_3 live, and it contributes to three already-live output labels (Figure 6c).

LARS follows this pattern to progress to the schedule shown in Figure 6d, and ultimately that of 6e. At each step, we had at most three output points, that had high secondary affinity to each other, live consecutively. Also, we made exactly one input point live, which contributed to all the live output points, and was immediately released. Therefore, irrespective of the degree of unrolling, the maximum register pressure for the schedule by LARS will be 4.

Obtaining such a schedule is difficult with the existing instruction schedulers. The existing schedulers may interleave the computation across trees to increase ILP. However, among multiple interleavings with the same degree of ILP, there will be only a few that will simultaneously reduce the register pressure. Finding such interleavings requires more than just a greedy decision based on a local perspective of register pressure. LARS is able to find such an interleaving by having a broader perspective of register pressure based on release/fire potential, and cumulative primary/secondary affinities.

Appendix B
Assembly Code Analysis

We perform an analysis of the generated assembly code for the various versions, and count the number memory accesses via loads into the *ymm/zmm* registers in the computational loop. The numbers for Xeon Phi 7250 are presented in Table V. One can observe from the data that the reordered version incurs far lesser memory accesses than the unrolled version, indicating that the reordering with LARS reduces the number of spills and reloads.

Appendix C
Artifact Description

A. Abstract

The artifact comprises LARS, an automated framework to perform reordering optimization on straight-line codes; the algorithmic details of the framework are described in the SC'18 paper *Associative Instruction Reordering to Alleviate Register Pressure*. The artifact will be publicly available for download from github. The downloaded package comes with

- The source code for the framework
- The benchmarks in the `examples/` directory
- Documentation on how to add a new stencil benchmark in the `docs/` directory
- Makefile to compile LARS, and shell scripts to run the benchmarks and verify the results reported in the paper

B. Description

1) Check-list (artifact meta information):

- **Algorithm:** Reordering framework for straight-line codes on CPUs.
- **Program:** C/C++ input.
- **Compilation:** g++ with c++11 support (GCC 4.9.2 and 5.3.0 tested).
- **Transformations:** The framework extracts the statements from the pragma-demarcated input C/C++ file, performs lowering transformations on the statements, and then reorders the lowered statements to enhance data reuse and simultaneously reduce register pressure.
- **Binary:** Makefile is included in the package to generate the executable. Reordered versions generated by the framework are included for all the benchmarks; the scripts used to generate the reordered versions are also included.
- **Data set:** Included in the `examples/` directory.
- **Run-time environment:** Tested on Ubuntu 16.04, and Red Hat Enterprise Linux Server release 6.7 operating system.
- **Hardware:** We recommend a linux platform
- **Output:** GFLOPS for all the input benchmarks.
- **Publicly available?:** Yes.

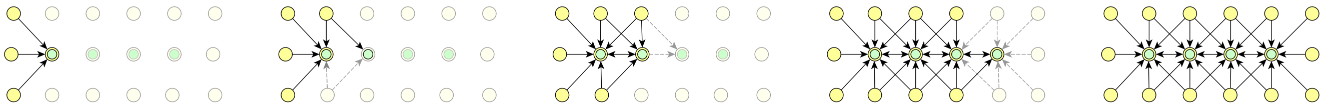
2) How software can be obtained (if available): The framework is open-source, and will be available for download from the git repository https://github.com/pssrawat/LARS. The downloaded package comprises the source code, the benchmarks, and the evaluation instructions and scripts. All the files in the repository are licensed to The Ohio State University.

3) Hardware dependencies: The framework has been tested on Ubuntu 16.04 and Red Hat Enterprise Linux Server release 6.7.

4) Software dependencies:

- flex version $\geq$ 2.6.0 (2.6.0 tested)
- bison version $\geq$ 3.0.4 (3.0.4 tested)
- cmake version $\geq$ 3.8 for gpucc (3.8 tested)
- Boost version $\geq$ 1.58 (1.58 tested)
- GCC version $\geq$4.8.1 with c++11 support to compile the framework (4.9.2 and 5.3.0 tested)
- GCC version $\geq$ 7.2.0 for benchmarking
- LLVM version $\geq$ 6.0.0 for benchmarking

5) Datasets: All the benchmarks that are evaluated in the paper are packaged in the `examples/` directory. Additionally, Makefiles and scripts are included for easy evaluation.



(a) step 1 (b) step 2 (c) step 3 (d) step 4 (e) step 5

Fig. 6: Steps in instruction reordering with LARS for a 4-way unrolled 2D 9-point Jacobi stencil

Bench.	GCC					LLVM				
	Og	Unrolled		LARS		Og	Unrolled		LARS	
	A	A	UF	A	UF	A	A	UF	A	UF
2d25pt	55	77	{2,1}	58	{2,1}	67	207	{4,1}	70	{4,1}
2d49pt	141	344	{4,1}	144	{4,1}	153	399	{4,1}	138	{4,1}
2d64pt	201	494	{4,1}	174	{4,1}	213	519	{4,1}	165	{4,1}
2d81pt	269	664	{4,1}	206	{4,1}	281	655	{4,1}	194	{4,1}
2d121pt	429	1064	{4,1}	294	{4,1}	286	2056	{8,1}	763	{8,1}
3d27pt	65	143	{8,1,1}	93	{8,1,1}	71	223	{4,1,1}	88	{4,1,1}
3d125pt	468	1133	{2,2,1}	361	{2,2,1}	277	1019	{4,1,1}	497	{4,1,1}
chebyshev	42	131	{2,2,1}	73	{2,2,1}	50	297	{4,2,1}	127	{4,2,1}
7-point	15	67	{8,1,1}	62	{8,1,1}	26	34	{4,1,1}	32	{4,1,1}
poisson	27	43	{2,1,1}	35	{2,1,1}	36	166	{8,1,1}	102	{8,1,1}
helmholtz-v2	21	62	{4,1,1}	60	{4,1,1}	33	75	{4,1,1}	60	{4,1,1}
helmholtz-v4	74	151	{2,1,1}	111	{2,1,1}	76	266	{4,1,1}	233	{4,1,1}
27-point	36	228	{4,2,1}	99	{4,2,1}	48	131	{4,1,1}	67	{4,1,1}
hypterm	317	-	-	268	{1,1,1}	307	-	-	214	{1,1,1}
diffterm	345	-	-	328	{1,1,1}	440	-	-	386	{1,1,1}
rhs4th3fort	632	-	-	426	{1,1,1}	327	-	-	287	{1,1,1}
derivative	299	-	-	223	{1,1,1}	311	-	-	290	{1,1,1}
cell-advec 3D	49	92	{2,1,1}	84	{2,1,1}	50	94	{2,1,1}	79	{2,1,1}
mom-advec 3D	61	114	{2,1,1}	60	{1,1,1}	84	118	{2,1,1}	55	{1,1,1}
accelerate 3D	76	150	{2,1,1}	75	{1,1,1}	75	156	{2,1,1}	51	{1,1,1}
ideal-gas 3D	7	12	{2,1,1}	5	{1,1,1}	6	13	{2,1,1}	6	{1,1,1}
PdV 3D	47	86	{2,1,1}	46	{1,1,1}	41	86	{2,1,1}	70	{2,1,1}
calc-dt 3D	44	82	{2,1,1}	71	{2,1,1}	42	77	{2,1,1}	58	{2,1,1}
fluxes 3D	33	120	{2,2,1}	95	{2,2,1}	32	120	{2,2,1}	92	{2,2,1}
viscosity 3D	47	84	{2,1,1}	80	{2,1,1}	50	100	{2,1,1}	80	{2,1,1}
cell-advec 2D	76	143	{2,1}	132	{2,1}	72	139	{2,1}	127	{2,1}
mom-advec 2D	47	164	{4,1}	141	{4,1}	24	163	{4,1}	85	{4,1}
accelerate 2D	42	162	{4,1}	99	{4,1}	40	170	{4,1}	97	{4,1}
ideal-gas 2D	7	12	{2,1}	7	{1,1}	6	13	{2,1}	6	{1,1}
PdV 2D	28	48	{2,1}	42	{2,1}	22	42	{2,1}	38	{2,1}
calc-dt 2D	23	42	{2,1}	39	{2,1}	24	43	{2,1}	34	{2,1}
fluxes 2D	16	50	{4,1}	44	{4,1}	14	50	{4,1}	43	{4,1}
viscosity 2D	23	42	{2,1}	38	{2,1}	28	47	{2,1}	36	{2,1}
sd-t-d1-1	13	76	{4,8}	48	{4,8}	13	76	{4,8}	50	{4,8}
sd-t-d1-2	13	76	{4,8}	48	{4,8}	13	76	{4,8}	48	{4,8}
sd-t-d1-3	15	76	{4,8}	50	{4,8}	16	69	{4,8}	57	{4,8}
sd-t-d1-4	13	82	{4,8}	54	{4,8}	13	76	{4,8}	50	{4,8}
sd-t-d1-5	13	126	{4,8}	93	{4,8}	13	104	{4,8}	62	{4,8}
sd-t-d1-6	15	141	{4,8}	72	{4,8}	16	106	{4,8}	57	{4,8}
sd-t-d1-7	13	82	{4,8}	60	{4,8}	13	76	{4,8}	48	{4,8}
sd-t-d1-8	13	126	{4,8}	93	{4,8}	13	104	{4,8}	73	{4,8}
sd-t-d1-9	12	121	{4,8}	97	{4,8}	18	126	{4,8}	71	{4,8}

A: total memory accesses, UF: unrolling factor along loop dimensions {k,j,i}

TABLE V: Assembly code analysis on Xeon Phi 7250

C. Installation

First clone the artifact source to a local machine:

```
$. git clone https://github.com/pssrawat/LARS
```

Then compile the source code to create the executables:

```
$. cd LARS
$. make all
```

To run the benchmarks:

```
$. cd examples
```

Edit `run-benchmarks.sh` in `examples/` to select appropriate target architecture, and set up paths to the benchmarking compilers. The default has been set for the Skylake i7-6770K processor.

Execute the benchmarking script:

```
$. ./run-benchmarks.sh
```

The computed GFLOPS for all the benchmarks will be redirected to the output file `output.txt` in the `examples/` directory.

D. Evaluation and expected result

The performance for each stencil benchmark in GFLOPS will be in `output.txt` after the evaluation script successfully finishes.

E. Experiment customization

The unrolling factors can be changed in the input C/C++ file, and the reordered versions regenerated by using the `reorder.sh` script provided with each benchmark.

The documentation in `docs/` folder provides additional details about adding new benchmarks, and optimizing them LARS.