Register Optimizations for Stencils on GPUs

Prashant Singh Rawat
The Ohio State University
rawat.15@osu.edu

Aravind Sukumaran-Rajam
The Ohio State University
sukumaranrajam.1@osu.edu

Atanas Rountev
The Ohio State University
rountev.1@osu.edu

Fabrice Rastello
INRIA
fabrice.rastello@inria.fr

Louis-Noël Pouchet
Colorado State University
pouchet@colostate.edu

P. Sadayappan
The Ohio State University
sadayappan.1@osu.edu

Abstract

The recent advent of compute-intensive GPU architecture has allowed application developers to explore high-order 3D stencils for better computational accuracy. A common optimization strategy for such stencils is to expose sufficient data reuse by means such as loop unrolling, with the expectation of register-level reuse. However, the resulting code is often highly constrained by register pressure. While current state-of-the-art register allocators are satisfactory for most applications, they are unable to effectively manage register pressure for such complex high-order stencils, resulting in sub-optimal code with a large number of register spills. In this paper, we develop a statement reordering framework that models stencil computations as a DAG of trees with shared leaves, and adapts an optimal scheduling algorithm for minimizing register usage for expression trees. The effectiveness of the approach is demonstrated through experimental results on a range of stencils extracted from application codes.

CCS Concepts • **Software and its engineering → Compilers**;

1 Introduction

Stencil computations are an important computational motif in many scientific applications. Typically, a simple stencil computation updates elements of one or more output arrays using elements in the spatial neighborhood from one or more input arrays. The footprint of a stencil is determined by its *order*, which is the number of input elements from the center read along each dimension. In many scientific applications, the stencil order determines the computational accuracy. For this reason, high-order stencils have been gaining popularity. However, the inherent data reuse within or across statements in such high-order stencils exposes performance challenges that are not addressed by current stencil optimizers.
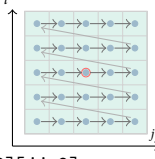
A significant focus in optimizing stencil computations has been to fuse operations across time steps or across a sequence of stencils in a pipeline [5, 21, 22, 36, 44, 54, 59]. With high-order stencils, the operational intensity is sufficiently high so that even with just a simple spatial tiling, the computation should theoretically not be memory-bandwidth bound. Consider a GPU with around 300 GBytes/sec global memory bandwidth and a peak double-precision performance of around 1.5 TFLOPS. The required operational intensity to be compute-bound and not memory-bandwidth bound is around 5 FLOPs/byte or 40 FLOPs per double-word. Many high-order stencil computations have much higher arithmetic intensities than 40. For such stencils, achieving a high degree of reuse in cache is very feasible, but high performance is not realized on GPUs. *The main hindrance to performance is the high register pressure with such codes, resulting in excessive register spilling and a subsequent loss of performance.* As we elaborate in the next section, existing register management techniques in production compilers are not well equipped to address the problem with register pressure for high-order stencils. Addressing this problem in context of GPUs is even more challenging, since most of the widely used GPU compilers like NVCC [38] are closed-source. Even the recent effort by Google (gpucc [57]) only exposes the front-end to the user, and uses the NVCC backend as a black box to perform instruction scheduling and register allocation.

In this paper, we develop an effective pattern-driven global optimization strategy for instruction reordering to address this problem. The key idea behind the instruction reordering approach is to model reuse in high-order stencil computations by using an abstraction of a DAG of trees with shared nodes/leaves, and exploit the fact that optimal scheduling to

```
for (i=2; i<N-2; i++)
  for (j=2; j<N-2; j++) {
    out[i][j] = 0;
    for (ii=-2; ii<=2; ii++)
      for (jj=-2; jj<=2; jj++)
        out[i][j] += in[i+ii][j+jj] * w[ii+2][jj+2];
}
```

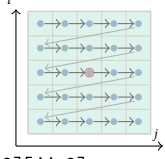**(a)** Stencil with lexicographical sweeps

```
for (i=2; i<N-2; i++)
  for (j=2; j<N-2; j++) {
    out[i][j] = 0;
    for (ii=2; ii>=-2; ii--)
      for (jj=-2; jj<=2; jj++)
        out[i][j] += in[i+ii][j+jj] * w[ii+2][jj+2];
}
```

(b) Stencil with reverse-lexicographical sweeps

Figure 1. Comparing same stencil computation with different sweeping order

minimize registers for a single tree with distinct operands at the leaves is well known [47]. We thus devise a statement reordering strategy for a DAG of trees with shared nodes that enables reduction of register pressure to improve performance. The paper makes the following contributions:

- It proposes a framework for multi-statement stencils that reduces register pressure by reordering instructions across statements.
- It describes novel heuristics to schedule a DAG of trees that reuse data using a minimal number of registers.
- It demonstrates the effectiveness of the proposed framework on a number of register-constrained stencil kernels.

2   Background and Motivation

Register Allocation and Instruction Scheduling   A compiler has several optimization passes, register allocation and instruction scheduling being two of them. Passes before register allocation manipulate an intermediate representation with an unbounded number of *temporary* variables. The goal of register allocation is to assign those temporaries to physical storage locations, favoring the few but fast registers to the slower but larger memory.

For a fixed schedule, a common approach to perform register allocation is to build an interference graph of the program, which captures the intersection of the live-ranges of temporaries at any program point. Register assignment is then reduced to coloring the interference graph, where each color represents a distinct register [10, 11]. Interfering nodes in the interference graph are assigned different colors due to their adjacency. The number of registers needed by the coloring algorithm is lower-bounded by the maximum number of intersecting live-ranges at any program point (MAXLIVE). If MAXLIVE is more than the number of physical registers, *spilling* of registers and the consequent load/store operations from/to memory are unavoidable.

Register pressure can sometimes be alleviated by reordering the schedule of dependent instructions to reduce the MAXLIVE. Reordering independent instructions is often used to enhance the amount of instruction-level parallelism (ILP), for hiding memory access latency. Thus, there is a complex interplay between instruction scheduling and register allocation, affecting instruction-level parallelism and register pressure, and the associated optimization problem

is highly combinatorial. Production compilers generally use heuristics for increasing ILP, with a best-effort greedy control on register pressure. For typical application codes, the negative effect on register pressure is not very significant. However, for high-order stencil codes with a large number of operations and a lot of potential register-level reuse, the impact can be very high, as illustrated by an example below.

Illustrative Example   Consider an unrolled version of the double-precision 2D Jacobi stencil computation (Figure 1a) from [50]. NVCC interleaves the contribution from each input point to different output points to increase instruction level parallelism (ILP). The interleaving performed to increase ILP also has the serendipitous effect of reducing the live range of the register data, and a consequent reduction in register pressure. Nvprof [39] profiling data on a Tesla K40c device shows that under maximum occupancy, this version performs 3.73E+06 spill transactions, achieving 467 GFLOPS.

Figure 1b shows the same stencil computation after changing the order of accumulation. Exactly the same contributions are made to each result array element, but the order of the contributions has been reversed. With this access pattern for the code in Figure 1b, NVCC fails to perform the same interleaving despite allowing reassociation via appropriate compilation flags. In fact, the register pressure is now exacerbated by the consecutive scheduling of independent operations to increase ILP. For this version, 1.58E+08 spill transactions were measured, with performance dropping to 51 GFLOPS.

This example illustrates a problem with register allocation when the computation has a specific reuse pattern, characteristic of high-order stencil computations. The problem stems from the fact that for most compliers the register allocation and instruction scheduling algorithms that operate at a basic-block level have a peephole view of the computation – they make scheduling/allocation decisions without a global perspective, and thus sometimes work antagonistically. Meanwhile, stencil computations typically have a very regular access pattern. With a better understanding of the pattern, and a global perspective on the computation, it is feasible to devise an instruction reordering strategy to alleviate register pressure.

Solution Approach   In this paper, we circumvent the complexity of the general optimization problem of instruction reordering and register allocation by devising a pattern-specific

optimization strategy. Stencil computations involve accumulation of contributions from array data elements in a small neighborhood around each element. The additive contributions to a data element may be viewed as an expression tree. Thus, for multi-statement stencils, we have a DAG of expression trees. Due to the fact that an element may contribute to several result elements, the trees within the DAG can have many shared leaves.

Given a single tree without any shared leaves, it is well known [47] how to schedule its operations in order to minimize the number of registers needed. We use this as the basis for developing heuristics to schedule the operations from the DAG of trees with shared leaves. In contrast to the problem of reordering an arbitrary sequence of instructions to minimize register pressure, a structured approach of adapting the optimal schedule for isolated trees to the case of DAG of trees with shared leaves results in an efficient and effective algorithm that we develop in the next two sections.

3   Scheduling DAG of Expression Trees

Stencil computations are often succinctly represented using a domain-specific language (DSL). Listing 1 shows a 7-point Jacobi stencil expressed in an illustrative DSL, similar in spirit to stencil computation DSLs such as SDSL [25] and Forma [43]. The core computation is shown in lines 2–4. As with similar DSLs, the user can specify unroll factors for loop iterators (line 9). Loop unrolling, or thread coarsening on GPUs, is often used to exploit register-level reuse in the code. The computation is automatically unrolled as a preprocessing step, before the code is generated and optimized.

It is important to note that using a DSL is not a prerequisite for using the scheduling techniques proposed in this work. As described shortly, our approach works on a DAG of expression trees. This DAG can be automatically extracted either from the DSL representation or from C/Fortran code.

A stencil statement can be defined by the stencil shape (as in lines 2–4) and the input/output data (as in line 8). Each such stencil statement can be represented by a labeled expression tree. For example, the tree corresponding to the computation in Listing 1 has array element $out[k, j, i]$ as its root, scalars a, b, c and accesses to elements of array in as its leaves, and arithmetic operators $*$ and $+$ as inner nodes.

An expression tree for a stencil computation has three types of nodes: (1) nodes $n \in N_{mem}$ representing accesses to memory locations, (2) nodes $n \in N_{op}$ representing binary/unary arithmetic operators, and (3) leaf nodes representing constants. All leaf nodes in N_{mem} correspond to reads of array elements (e.g., $in[k + 1, j, i]$) or scalars. The root of the expression tree is also in N_{mem} and corresponds to a write to an array element (e.g., $out[k, j, i]$) or a scalar. We associate a unique label with each read/written memory location, and assign to each node in N_{mem} the corresponding label.

Listing 1. The input representation in the DSL
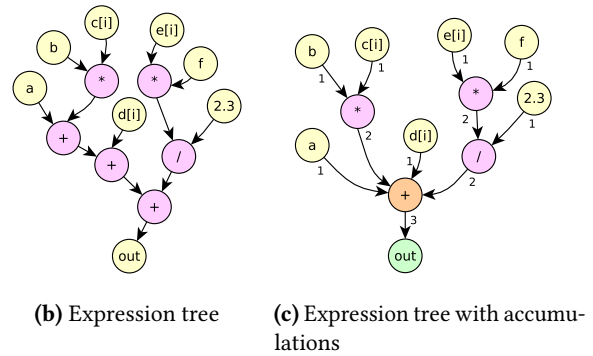
```
1  function j3d7pt (out, in, a, b, c) {
2    out[k][j][i] = a*(in[k+1][j][i]) + b*(in[k][j-1][i] +
3        in[k][j][i-1] + in[k][j][i] + in[k][j][i+1] +
4            in[k][j+1][i]) + c*(in[k-1][j][i]);
5  }
6  parameter L, M, N;
7  iterator k, j, i;
8  double in[L][M][N], out[L][M][N], a, b, c;
9  unroll k=2, j=2;
10 j3d7pt (out, in, a, b, c);
11 return out;
```

```
out = a + (b * c[i]) + d[i] + ((e[i] * f) / 2.3);
```

(a) Illustrative stencil statement

(b) Expression tree

(c) Expression tree with accumulations

Figure 2. Expression tree example

The remaining tree nodes are in N_{op}. Figure 2b shows the expression tree for an illustrative expression.

In a preprocessing step, we introduce k-ary nodes for associative operators. For example, for the tree in Figure 2b, the chain of + nodes is replaced with a single "accumulation" + node. Figure 2c shows the resulting expression tree; the numbers on the nodes will be described shortly. The semantics of an accumulation node is as expected: the value is initialized as appropriate (e.g., 0 for +, 1 for $*$) and the contributions of the children are accumulated in arbitrary order.

We often consider a sequence of stencil computations—for example, in image processing pipelines [43]. Each computation in the sequence will be represented by a separate expression tree. Similarly, unrolling will result in distinct expression trees for each unrolled instance. For example, after unrolling along dimensions k and j in Listing 1, there will be a sequence of four expression trees. In some cases the output of a tree is used as an input to a later tree in the sequence. In such a case, there is a flow dependence: the root of the producer tree has the same label as some leaf node of the consumer tree (without an in-between tree that writes to that label). In the input to our analysis, this flow is represented by a dependence edge from the root node to the leaf node. Thus, the entire computation is represented as a DAG of expression trees.

Throughout the paper, we make the following two assumptions: (1) the assembly instructions generated for the DAG of trees after register allocation are of the form $r_1 \leftarrow r_2$ *op* r_3,

where r_1 and r_2 can be the same; (2) each operand/result requires exactly one register. This condition is only enforced to simplify the presentation of the next two sections, and can be very easily relaxed [4]. Our objective is to schedule the computations in the DAG so that the register pressure is reduced.

3.1 Sethi-Ullman Scheduling

We will use "data sharing" to refer to cases where the same memory location is accessed at multiple places. There are two types of data sharing: (1) within a tree: several nodes from N_{mem} have the same label; and (2) across trees: in a DAG of trees, nodes from distinct trees have the same label.

A classic result, due to Sethi and Ullman [47], applies to a *single* expression tree *without* data sharing (i.e., each $n \in N_{mem}$ has a unique label), and with binary/unary operators. They present a scheduling algorithm that minimizes the number of registers needed to evaluate such an expression tree under a spill-free model.[1] Each tree node n is assigned an Ershov number [1]; we will refer to them as "Sethi-Ullman numbers" and denote them by $su(n)$. They are defined as

$$su(n) = \begin{cases} 1 & n \text{ is a leaf} \\ su(n_1) & n \text{ has one child } n_1 \\ max(su(n_1), su(n_2)) & su(n_1) \neq su(n_2) \\ 1 + su(n_1) & su(n_1) = su(n_2) \end{cases} \quad (1)$$

The last two cases apply to a binary op node n with children n_1 and n_2. Intuitively, $su(n)$ is the smallest possible number of registers used for the evaluation of the subtree rooted at n. The first two cases are self-explanatory. For a binary op node n, if one child n' has a higher register requirement (case 3), this "big" child should be evaluated first. The result of n' will be stored in a register, which will be alive while the second ("small") child is being evaluated. The remaining $su(n') - 1$ registers used for n' are available (and enough) to evaluate the small child. Finally, the register of n' can be used to store the result for n, meaning that $su(n)$ is equal to $su(n')$. If the order of evaluation were reversed, the result of the small child would have to be kept in a register while n' is being evaluated, which would lead to sub-optimal $su(n) = 1 + su(n')$. In the last case in equation (1), both children have the same register needs; thus, their relative order of evaluation is irrelevant and one extra register is needed for n. Of course, under the definitions in equation (1), $su(n)$ is the same as MAXLIVE for the tree rooted at n.

It is straightforward to generalize Sethi-Ullman numbering to trees containing accumulation nodes (as in Figure 2c). Each such accumulation node n has children n_i for $1 \leq i \leq k$. Let $mx = max_i\{su(n_i)\}$. If there is a single child n_j with $su(n_j) = mx$, this child is scheduled for evaluation first, and therefore $su(n) = mx$. If two or more children n_j have

---
[1]In a spill-free model of the computation, a data element is loaded only once into a register for all its uses/defs.

$su(n_j) = mx$, one of them is scheduled first; however, in this case $su(n) = 1 + mx$. In both cases, the order of evaluation of the remaining children is irrelevant. Figure 2c shows the Sethi-Ullman numbers for the sample expression tree.

Note that the schedules produced by this approach perform *atomic evaluation* of subexpressions: one of the children is evaluated completely before the other ones are considered. For a tree without data sharing, this restriction does not affect the optimality of the result. In the presence of data sharing, atomic evaluation may not be optimal.

Since stencils read values from a limited spatial neighborhood, data sharing often manifests in the DAG of expression trees. For example, in Listing 1, $in[k][j][i]$ will be an input to all four expression trees corresponding to the unrolled stencil statements. One can also find other nodes in Listing 1 that will be shared across multiple expression trees. For such DAGs, the Sethi-Ullman algorithm cannot be directly applied to obtain an optimal schedule. In Section 3.2, we present an approach to compute an optimal atomic schedule for a DAG of expression trees *with* data sharing. In cases when finding an optimal evaluation can be prohibitively expensive, Section 3.3 presents heuristics to trade off optimality in favor of pruning the exploration space. Finally, restricting the evaluation to be atomic can generate sub-optimal schedules. Section 4 presents a remedial slice-and-interleave algorithm that performs interleaving on the output schedule generated by the approach presented in Section 3.2.

3.2 Scheduling a Tree with Data Sharing

Figure 3a shows an expression tree with data sharing. For illustration, nodes with the same label are connected in the figure. Recall that we assume a spill-free model, therefore a shared label loaded once into a register will remain live for all its uses. With data sharing, there is a possibility that (1) a label is already live before we begin the recursive evaluation of a subtree that has its subsequent use, and/or (2) a label must remain live even after the evaluation of the subtree in which it is used. The optimal schedule of a subtree is affected by the labels that are live before and after the evaluation of the subtree. Therefore, we need to add live-in/out states as parameters to the computation of the optimal schedule of a subtree. In this section, we present an approach to optimally schedule a tree with data sharing, under the model of atomic evaluation of children; we defer the interleaving of computation across subtrees to Section 4.

For a node n, let $uses(n)$ be the set of labels used in the subtree rooted at n. Figure 3a shows $uses(n)$ for each internal node n. The live-in set for a node n, denoted by $in(n)$, contains all labels that are live before the subtree rooted at n is evaluated. The live-out set is

$$out(n) = (in(n) \cup uses(n)) \setminus kill(n) \quad (2)$$

where $kill(n)$ is the set of labels that have their last uses in the subtree rooted at n. Note that $kill(n)$ is context-dependent,

(a) Tree with data sharing   **(b)** Scheduling cost
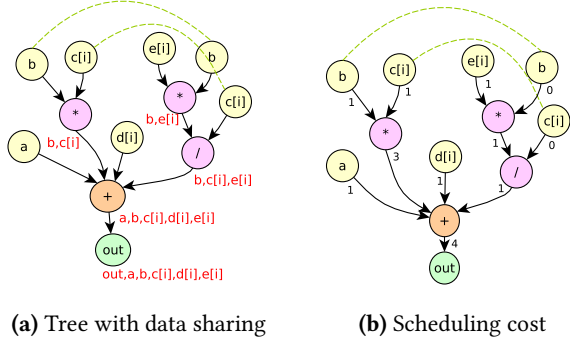
Figure 3. Scheduling a tree with data sharing

(a) Tree with context at root   **(b)** Optimal schedule

Figure 4. Example: computing $su'(\pi)$

i.e., the set will vary depending on the order in which the node is evaluated. The kill sets can be computed on the fly by maintaining the number of occurrences of each label l in the current schedule, and comparing it with the total number of occurrences of l in the entire DAG.

We now show how to compute a modified Sethi-Ullman number, su' for each node n, when provided with an "evaluation context" in terms of live-in and live-out labels. Consider a node n with some in and out state. Just before the evaluation of n begins, $|in(n)|$ registers are live. Similarly, just after the evaluation of n finishes, $|out(n)|$ registers will be live. During the evaluation of n, additional registers may become live, while some of the other live registers may be released. Now $su'(n, in, out)$ represents the maximum number of registers that were simultaneously live at any point during the evaluation of n. We also define $su'(\pi, in, out)$, where π is a sequence of the children nodes of n. This value will represent the maximum number of registers that were simultaneously live at any point during the evaluation of n with its children ordered in the sequence described by π.

For simplicity we will use $su'(n)$ instead of $su'(n, in, out)$, but the definitions will use the live-in/out sets $in(n)$ and $out(n)$.

For a leaf node $n \in N_{mem}$ with $|in(n)| = \alpha$,

$$su'(n) = \begin{cases} \alpha + 1 & label(n) \notin in(n) \\ \alpha & label(n) \in in(n) \end{cases}$$

The first case implies that a new register must be reserved for the label of n if it is not already live before the evaluation of n. The second case is self-explanatory.

To compute su' for a k-ary (binary or accumulation) node n with children $n_1 \dots n_k$, we need to explore all $k!$ evaluation orders of the children. Let π be any permutation of the children of n representing their evaluation order. Then $su'(n) = \min_{\pi} su'(\pi)$.

For the purpose of explanation, suppose the permutation $\pi = \langle n_1, n_2 \rangle$ is one particular evaluation order for a binary node n with children n_1, n_2. To compute $su'(\pi)$, first we determine the live-in and live-out sets for nodes in π as follows: $in(n_1) = in(n)$, $in(n_2) = out(n_1)$, and $out(n_2) = out(n)$; here $out(n_1)$ is as defined in equation 2. This provides the
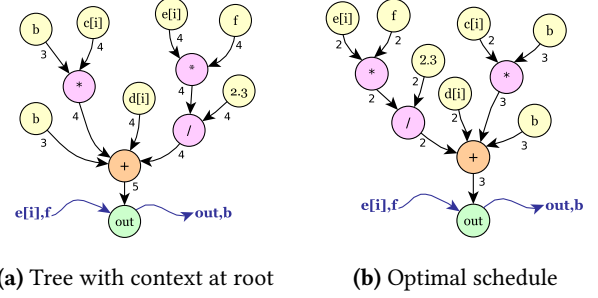
required context to compute $su'(n_1)$ and $su'(n_2)$. Let $mx = max_i\{su'(n_i)\}$, so that mx equals the maximum number of simultaneously live registers at any time during the evaluation of π. Then,

$$su'(\pi) = \begin{cases} 1 + mx & n_i \in N_{mem} \ \& \ label(n_i) \in out(n) \\ mx & \text{otherwise} \end{cases}$$

In case 2, if $n_1 \in N_{op}$, or if $n_1 \in N_{mem}$ but $label(n_1) \notin out(n)$, then the result of the computation, identified by the label of the node n, can reuse the register of n_1 (similarly for n_2). However, in case 1, where both n_1 and n_2 are leaf nodes in N_{mem} and both must be live after evaluating n, we need an additional register to hold the result.

For an accumulation node with k children, consider permutation $\pi = \langle n_1, n_2 \dots, n_k \rangle$. Suppose we have computed all $su'(n_i)$ and let $mx = max_i\{su'(n_i)\}$. Then,

$$su'(\pi) = \begin{cases} mx & su'(n_1) = mx \ \& \ n_1 \in N_{mem} \ \& \ label(n_1) \\ & \notin out(n_1) \ \& \ su'(n_j) \neq mx, 2 \leq j \leq k \\ mx & su'(n_1) = mx \ \& \ n_1 \in N_{op} \ \& \\ & su'(n_j) \neq mx, 2 \leq j \leq k \\ 1 + mx & \text{otherwise} \end{cases}$$

Just like the generalization of $su(n)$ for accumulation nodes in Section 3.1, $su' = mx$ when the following two conditions hold: (1) n_1 requires the maximum number of simultaneously live registers, and the rest of the nodes in π can be completely evaluated using the registers released by n_1, and (2) the register holding n_1 can be reused by n, i.e., either $n_1 \in N_{op}$ (case 2), or n_1 is a leaf node that is not live beyond this point (case 1). In all other scenarios, we need $mx + 1$ registers (case 3).

The computation of $su'(n)$ for a tree without an evaluation context is shown in Figure 3b, and with an evaluation context is shown in Figure 4a. For the same tree, Figure 4b shows the permutation with minimum su'. In all three figures, the children of a node are ordered left-to-right, which defines the corresponding permutation.

In some cases, exhaustively exploring all permutations of the children may be unnecessary. In the tree of Figure 4a, there are two subtree operands of the accumulation node that share no data (the first and the third subtree operands

in the left-to-right order). Therefore, even though the scheduling within those two subtrees may be influenced by the evaluation context, they do not influence each other's evaluation. Let *passthrough* denote the labels that are live both before and after the evaluation of node n: $passthrough(n) = in(n) \cap out(n)$. Then, for a k-ary node n, any two of its children n_i and n_j do not influence each other's evaluation if

$$(uses(n_i) \cap uses(n_j)) \setminus passthrough(n) = \emptyset \qquad (3)$$

In such a scenario, for the node n, we can create maximal clusters of its children that share data, but the only data shared among clusters is the passthrough labels of n. For example, if children t_1 and t_2 share label l_1, and children t_2 and t_3 share label l_2, where l_1, l_2 are non-passthrough labels of the parent node, then $\{t_1, t_2, t_3, t_4\}$ must belong to the same cluster. We extend the intuition behind Sethi-Ullman scheduling algorithm to establish that different clusters cannot influence each other's evaluation. Then, for each cluster c_i, we can independently compute $su'(c_i)$ with $in(c_i) = in(n)$. We only need to explore all permutations within the non-singleton clusters. We propose the following theorems to establish an evaluation order for the clusters.

Theorem 3.1. *For k clusters c_i $1 \leq i \leq k$ such that $|in(c_i)| \leq |out(c_i)|$, the one with larger $su'(c_i) - |out(c_i)|$ will be prioritized for evaluation over others in the optimal schedule. In the special case where all the clusters have the same $su'(c_i) - |out(c_i)|$, they can be evaluated in any order without affecting MAXLIVE.*

This result is a direct consequence of the Sethi-Ullman algorithm. The cluster with larger $su'(c_i) - |out(c_i)|$ will release more registers, which can be reused by the next cluster. The special case too is a direct consequence of the Sethi-Ullman algorithm, where two sibling nodes with the same su can be evaluated in any order (case 4 of equation 1).

Theorem 3.2. *For two clusters c_1 and c_2 such that $|in(c_1)| > |out(c_1)|$ and $|in(c_2)| \leq |out(c_2)|$, c_1 must be evaluated before c_2 in the optimal schedule.*

We prove the result by contradiction. Suppose that c_2 is evaluated before c_1 in the optimal schedule. Since the schedule is optimal, $su'(c_2) \geq su'(c_1)$. Now we change this optimal schedule by moving the evaluation of c_1 before c_2. Evaluating c_1 earlier will release $|in(c_1)| - |out(c_1)|$ (i.e., $\geq$ 1) registers, which can then be used in the evaluation of c_2. Based on the previous equations, the $su'(c_2)$ will either decrease or remain the same, depending on whether the number of registers released by c_1 is greater than, or equal to 1. This modified schedule therefore either has the same, or has lower su' than the optimal schedule, making it an optimal schedule.

Theorem 3.3. *For two clusters c_1 and c_2 such that $|in(c_1)| > |out(c_1)|$ and $|in(c_2)| > |out(c_2)|$, the one with smaller su' must be prioritized for evaluation in the optimal schedule.*

Algorithm 1: Schedule-Tree (n, in, out)

Input : A tree rooted at n with live-in/out contexts in and out
Output: An optimal schedule S for the tree

1 $sched\_cost \leftarrow \emptyset, S \leftarrow \emptyset$;
2 $C \leftarrow create\_maximal\_clusters(n)$; (Sec. 3.2)
3 **for** *each cluster c in C* **do**
4   **if** $|c|$ *is 1* **then**
5        $in(c) \leftarrow in(n)$;
6        $out(c) \leftarrow$ computed using equation 2;
7        $sched\_cost[c] \leftarrow su'(c)$;
8   **else**
9   compute in and out for each tree in c; (Sec. 3.2)
10       $\pi \leftarrow$ all permutations of the trees in c;
11       $sched\_cost[c] \leftarrow su'(\pi)$;
12 $P \leftarrow$ sequence clusters using $sched\_cost$ and Thms. 3.1, 3.2, 3.3;
13 **for** *each subtree t_s in the sequence described by P* **do**
14   append the schedule for t_s in S;
15 **return** S;

Again, we prove the result by contradiction. Suppose that $su'(c_1) < su'(c_2)$, and c_2 is scheduled before c_1 in the optimal schedule. We change this schedule by moving the evaluation of c_1 before that of c_2. From Theorem 3.2, $su'(c_2)$ after this change will either remain the same, or decrease. Thus, su' for the new schedule will either be the same, or reduce if $su'(c_2)$ was the maximum, making it an optimal schedule.

Based on these theorems, Algorithm 1 summarizes the evaluation an optimal schedule for a tree with data sharing.

3.3 Heuristics for Tractability

For a non-singleton cluster c, the algorithm presented in Section 3.2 can become prohibitively expensive if $|c|$ is large. For example, when $|c|$ changes from 7 to 8, the permutations explored increase from 5040 to 40320. We now present some heuristics that trade off optimality for tractability, and a caching technique to further speed up the algorithm.

Pruning Heuristics We begin by establishing, for any node n, the bounds on $su'(n)$. When n is evaluated with non-empty contexts in and out, the bounds are:

$$su'(n, \emptyset, \emptyset) \leq su'(n, in, out) \leq su'(n, \emptyset, \emptyset) + |in \cup out|$$

We prove the lower bound by contradiction; the proof for upper bound is omitted due to space constraints.

$su'(n, \emptyset, \emptyset) \leq su'(n, in, out)$: Assume to the contrary that $su'(n, in, out) < su'(n, \emptyset, \emptyset)$. We will modify the schedule S corresponding to $su'(n, in, out)$ as follows: prepend a stage to S that loads the labels $\in in(n)$ into $|in|$ registers, and make $in(n) = \emptyset$. Append a state to S that stores all the labels $\in out(n)$ from the respective registers into memory, and make $out(n) = \emptyset$. This modified schedule corresponds to $su'(n, \emptyset, \emptyset)$, and hence, $su'(n, \emptyset, \emptyset) = su'(n, in, out)$.

With the bounds established, instead of exploring all permutations, we can sacrifice optimality and stop further exploration when we are *close to* the optimal schedule. We

use a tunable parameter d, and stop trying the permutations any further when $su'(n, in, out) - su'(n, \emptyset, \emptyset) \leq d$. For the experimental evaluation in Section 5, we set d to 1.

For a cluster c with $|c| > 8$, we also apply a partitioning heuristic, which recursively partitions the subtrees in c into sub-partitions where each sub-partition can be of a maximum size p, with $p < 8$. The partitioning is based on either of the two criteria:

- on "label affinity": the subtrees that share the maximum labels are greedily assigned to the same sub-partition as long as the size of the sub-partition is less than p. Such partitioning is based on the notion that evaluating subtrees with maximum uses together will potentially reduce passthrough labels, and MAXLIVE.
- on "release potential": the subtrees that have the last uses of some labels are placed in a sub-partition, and that sub-partition is eagerly evaluated. This partitioning is based on the notion that the released registers can be reused by the next partition.

Once the sub-partitions are created, we only exhaustively explore all permutations of subtrees within a sub-partition. If the number of sub-partitions created is less than 8, then we also try all the permutations of the sub-partitions themselves. For example, if $|c| = 8$, and the partitioning heuristic creates two sub-partitions p_1 and p_2 of size 4 each, then our exploration space will be $\{p_1, p_2\}$ and $\{p_2, p_1\}$, while exploring all 4! permutations of subtrees within p_1 and p_2 each – a total of $2 \times 4! \times 4!$ permutations instead of 8! permutations.

We also let the user externally specify a threshold that upper-bounds the total number of permutations for a tree.

Memoization For a node n, a lot of permutations of its children will differ in only a few positions. In such cases, we end up recomputing su' for a child multiple times, even when the live-in/out context for the child remains unchanged.

These recomputations can be avoided by a simple memoization, where for a node n, we map $su'(n)$ as a function of a *minimal* context. The minimal context strips away labels that are not in $uses(n)$, but are in $passthrough(n)$. The $su'(n)$ with minimal context can be suitably adjusted to get $su'(n)$ with a different context that has some passthrough labels added to the minimal live-in/out. For example, suppose that $su'(n)$ is 3 when the minimal $in(n) = \{a,b\}$ and the minimal $out(n) = \emptyset$. Then $su'(n)$ when evaluating it with $in(n) = \{a, b, c\}$, $out(n) = \{c\}$ and $c \notin uses(n)$ will be 2+1=3, and the optimal schedule will remain unchanged. Memoization greatly reduces the total evaluation time, thereby enabling the exploration of a large number of permutations.

3.4 Scheduling a DAG of expression trees

For each stencil statement that is mapped to an expression tree, Section 3.2 described a way to schedule it. This section ties everything together for a multi-statement stencil by describing how to schedule a DAG of expression trees.

Algorithm 2: Schedule-DAG (D, R)

Input : D: DAG of expression trees, R: Per-thread register limit
Output : An optimal schedule S for the D

1   $D' \leftarrow D$; $fusion\_feasible \leftarrow true$; $tree\_order \leftarrow \emptyset$; $S \leftarrow \emptyset$;
2   **while** $fusion\_feasible$ **do**
3   **for** *each pair of transitive dep-free nodes* t_i, t_j *in* D' **do**
4       $M \cup = compute\_metric(D', t_i, t_j)$; (sec. 3.4)
5     $sort\_descending\ (M)$;
6     $(t_p, t_q, fusion\_feasible) \leftarrow find\_fusion\_candidate(M)$;
7   fuse t_p and t_q;
8     $update\_dependence\_edges\ (D', t_p, t_q)$;
9   **for** *each node* d *in* D' **do**
10   append the tree sequence of d in $tree\_order$;
11   $split\_versions \leftarrow create\_split\_versions(tree\_order)$;
12   **for** *each split in* $split\_versions$ **do**
13     $S' \leftarrow \emptyset$;
14   **for** *each kernel* k *in split* **do**
15   **for** *each tree* t *in* k **do**
16   compute in and out for t;
17   append output of Schedule-Tree(t, in, out) to S';
18   execute S' after compiling it with register limit R;
19     $S \leftarrow S'$ if S' is a faster schedule than S, or if S is $\emptyset$;
20   return S;

For optimal scheduling, one needs to explore all topological orders for the trees in the DAG, and then evaluate all the trees independently for each topological order. This may be practical if the size of the DAG is small. Otherwise, we must sacrifice optimality for tractability, and fix the evaluation order of the trees in the DAG before the trees are individually evaluated.

We use the greedy heuristic described by Rawat et al. [44] to fix the evaluation order of trees in the DAG. At each step, the heuristic tries to fix the evaluation order of two nodes in the DAG. We begin by computing, for each pair of transitive dependence-free trees p_i in the DAG, a metric M_i that encodes: (a) the number of labels shared between them, and (b) the number of common input arrays read by them. Among the computed M_i, we choose the one that has the highest non-zero value, and fix the evaluation order of its tree pair to be contiguous to enhance reuse proximity in the final schedule. The DAG is updated by fusing the nodes corresponding to the two trees into a "macro node". Post fusion, we update the dependence edges to and from the macro node, and recompute the metrics for the next step. The algorithm terminates when no more nodes can be fused.

Once the algorithm terminates, we perform a topological sort of the final DAG, and expand the DAG macro nodes to their tree sequences. For these ordered trees, we can generate code versions with different degree of splits. One extreme would be a version where all the trees are in a single kernel (*max-fuse*), and another extreme would be a version where each tree is a distinct kernel (*max-split*) [8, 55]. For compute-intensive stencils with many-to-many reuse, a single kernel can have extremely high register pressure, sometimes

causing spills despite allowing for the maximum permitted registers per thread. For such cases, performing kernel fission instead of generating a single kernel for the entire computation might improve performance. The split kernels will incur additional data transfers from global memory, but the register pressure per kernel will be much lower, giving the user an opportunity to further enhance register-level reuse via unrolling. Note that none of the production GPU compilers are capable of performing kernel fusion/fission optimizations. For each split version created, the tree sequence in it is evaluated using Algorithm 1. The returned schedule is the one that gives maximum performance. Algorithm 2 outlines the entire process.

4 Interleaving Expressions

At this point, we have a schedule for the entire DAG of trees, but with atomic evaluation enforced. However, interleaving within/across trees can be instrumental in reducing MAXLIVE. For example, in the unrolled stencil of Listing 1, there is no reuse within a stencil statement, but plenty of reuse across stencil statements. We will see later in Section 5 that relaxing the constraint of atomic evaluation, and performing interleaving is imperative for performance in such stencils. A compiler optimization that performs some interleaving is common subexpression elimination (CSE). However, we require a more general interleaving that works at the granularity of common labels instead of common subexpressions. For example, Figure 5a shows an expression tree where $su'(S)$ is the largest, and the operands of the accumulation node are evaluated in order from left to right in the final schedule. Also, $\{c[i], b\} \notin uses(S)$. The fact that $\{c[i],b\} \in passthrough(S)$ adds to $su'(S)$. By slicing the expression $(e[i] * b)/c[i]$ and placing it after the expression $b * c[i]$ as shown in Figure 5b, $c[i]$ and b will no longer be in $in(S)$. Instead of those two labels, a temporary label holding the value of the sliced expression will be added to $in(S)$, and hence $su'(S)$ will reduce by 1. Note that this is not CSE, but a more general optimization aimed to reduce MAXLIVE. This slice-and-interleave optimization slices a *target* expression, and interleaves it with a *source* expression, so that su' at a program point reduces. It subsumes CSE if the source and target expressions are the same.

We perform the slice-and-interleave optimization at two levels: (a) within an expression tree, where the source and target expressions belong to the same tree; and (b) across the expression trees in the DAG, where source and target expressions belong to different trees. For a chosen source expression e_s rooted at node n, we compute a set of labels, L_{ilv}, which is a union of all the labels that were observed in the schedule till now, with the labels that have a single occurrence in the DAG.

We now try to find a set of target expressions operating on just the labels from L_{ilv}. To find the target expressions, we
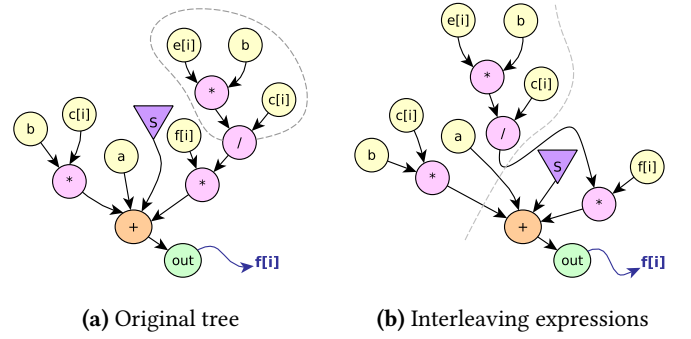

(a) Original tree   **(b)** Interleaving expressions

Figure 5. Example: interleaving to reduce MAXLIVE

Algorithm 3: slice-and-interleave (T, in, out)

Input : T: an input tree with schedule S and contexts in and out
Output: S: The schedule after applying slice-and-interleave

1 $L_{ilv} \leftarrow \emptyset$;
2 $min\_exprs \leftarrow$ sequence of minimal expressions extracted from S
   whose operands are leaf nodes;
3 **for** *each expression s in min_exprs* **do**
4    $\quad L_{ilv} \leftarrow$ all the labels seen in the schedule till s $\cup$ labels with
      single occurrence in T;
5    $\quad$ **for** *each expression t appearing after s in min_exprs* **do**
6      $\quad\quad$ **if** *t only operates on the labels in L_{ilv}* **then**
7        $\quad\quad\quad t' \leftarrow$ maximal expression obtained by growing t
          until it operates on the labels in L_{ilv};
8        $\quad\quad\quad$ **if** *live ranges reduced by placing t' after s* **then**
9          $\quad\quad\quad\quad$ slice and place t' after s in S;
10         $\quad\quad\quad\quad$ move t after s in min_exprs;
11 **return** S;

---

start with minimal expressions, i.e., the simplest expressions whose operands are leaf nodes $\in N_{mem}$. Once we find a minimal expression e_m that operates only on the labels $\in L_{ilv}$, we find the root node r of e_m, and grow e_m to the expression rooted at $parent(r)$. We continue to grow the expression till we have a maximal expression that only operates on the labels $\in L_{ilv}$. For each target expressions thus discovered, we check if slicing and placing it between the source expression e_s and subtree t_s immediately following e_s in the schedule decreases $|in(t_s)|$. If it does, then slice-and-interleave is performed.

Illustrative Example Let $b * c[i]$ be the source expression in the tree of figure 5a. One of the explored target expressions will be $e[i] * b$, since it only uses nodes $\in L_{ilv}$. Now we try to grow the target expression by changing the root from $*$ to $/$, making $(e[i] * b)/c[i]$ the new target expression. All the labels used in the grown target expression also belong to L_{ilv}. A further attempt to grow will change the target expression to $((e[i] * b)/c[i]) * f[i]$. However, $f[i] \notin L_{ilv}$. Therefore, we backtrack and finalize $(e[i] * b)/c[i]$ as a target expression, since it is the maximal expression with all the labels in L_{ilv}. Placing the target expression after the source expression decreases $in(S)$ by 1. Therefore, we perform the

slice-and-interleave optimization. Algorithm 3 outlines the slice-and-interleave algorithm that tries out different source expressions, and continuously finds the target expressions within the tree to interleave in order to reduce the live ranges. The slice-and-interleave across the trees in a DAG is similar.

5   Experimental Evaluation

Our framework parses stencil statements written in a subset of C: the array access indices in the stencil statements must be an affine function of the surrounding loop iterators, program parameters, and constants; loop iterators and parameters must be immutable in the stencil statements. The framework supports auto-unrolling along different dimensions to expose spatial reuse across stencil statements.

To ensure a tight coupling, several prior efforts on guiding register allocation or instruction scheduling were implemented as a compiler pass in research/prototype compilers [7, 16, 20, 41, 45], or open-source production compilers [29, 46]. However, like some other recent efforts [6, 28, 50], we implement our reordering optimization at source level for the following reasons: (1) it allows external optimizations for closed-source compilers like NVCC; (2) it allows us to perform transformations like exposing FMAs using operator distributivity, and performing kernel fusion/fission, which can be performed more effectively and efficiently at source level; and (3) it is input-dependent, not machine- or compiler-dependent – with an implementation coupled to compiler passes, it would have to be re-implemented across compilers with different intermediate representation. Our framework massages the input to a form that is more amenable to further optimizations by any GPU compiler, and we use appropriate compilation flags whenever possible to ensure that our reordering optimization is not undone by the compiler passes.

We evaluate our framework for the benchmarks listed in Table 1 on a Tesla K40c GPU (peak double-precision performance 1.43 TFLOPS, peak bandwidth 288 GB/s) with NVCC-8.0 [38] and LLVM-5.0.0 compiler (previously gpucc [57]). The first five benchmarks are stencils typically used in iterative processes such as solving partial differential equations [26]. The remaining three are representative of complex stencil operations extracted from applications. *hypterm* is a routine from the ExpCNS Compressible Navier-Stokes mini-application from DoE [17]; the last two stencils are from the Geodynamics Seismic Wave SW4 application code [51]. For each benchmark, the original version is as written by application developers without any loop unrolling; the unrolled version has the loops unrolled explicitly; and the reordered version is the output from our code generator. On an Intel i7-4770K processor, the code generator generated each reordered version under 4 seconds. When the net unrolling factor is limited to 4, the size of each reordered version is under 600 lines of code. The read-only arrays are annotated with the *restrict* keyword in all the versions to

Benchmark	N	UF	k	F	R	A	U
2d25pt	8192^2	4	2	33	2	104	44
2d64pt	8192^2	4	4	73	2	260	92
2d81pt	8192^2	4	4	95	2	328	112
3d27pt	512^3	4	1	30	2	112	58
3d125pt	512^3	4	2	130	2	504	204
hypterm	300^3	1	4	358	13	310	152
rhs4th3fort	300^3	1	2	687	7	696	179
derivative	300^3	1	2	486	10	493	165

N: Domain Size, UF: Unrolling Factor, k: Stencil Order, F: FLOPs per Point R: #Arrays, A: Total Elements Accessed per Thread, U: Unique Elements Accessed per Thread

Table 1. Benchmark characteristics

allow efficient loads via the texture pipeline. All the stencils are double-precision, compiled with NVCC flags '*−use_fast_-math Xptxas "-dlcm=ca"*', and LLVM flags '*-O3 -ffast-math -ffp-contract=fast*'. Since none of the versions use shared memory, using '*dlcm=ca*' for NVCC enhances performance by caching the global memory accesses at L1. However, we notice no discernible performance difference when compiling without the '*−use_fast_math*' flag in NVCC. We explore different instruction schedulers implemented in LLVM (*default, list-hybrid, and list-burr*) for the original and unrolled code, and report the best performance. To minimize instruction reordering for our reordered code, we use LLVM's default instruction scheduler, and do not use the *-ffast-math* option during compilation. We test all the versions against a standard C implementation of the benchmarks for correctness: the difference in each computed output value with that of the C implementation must be less than a tolerance of 1E-5.

Loop Unrolling   For the experiments, the iterative kernels were unrolled along a single dimension to expose spatial reuse. Loop unrolling offers the compiler an opportunity to exploit ILP, but scheduling independent instructions contiguously may increase register pressure. Consider an unrolled version of *2d25pt*, compiled with 32 registers. From table 1, it is clear that the unrolled code has a high degree of reuse. Listing 2 shows the SASS (Shader ASSembler) snippet generated using NVCC for the unrolled version of *2d25pt* after register allocation. The instructions not relevant to the discussion are omitted in Listing 2 (and 3), leading to non-contiguous line numbers. The lines highlighted in red show the instructions involving the same memory location – line 1 loads a value from global memory into register R4, and spills it in line 2 without using R4 in any of the intermediate instructions. Such wasteful spills are a characteristic of register-constrained codes. The same value is reloaded from local memory into R4 in line 4, and R4 is subsequently used in lines 5 and 8. The uses of R4 are placed far apart in SASS, adding to the register pressure. Interspersed with these instructions are the load (line 3) and subsequent uses of register R12. The interleaving increases ILP, but the uses of R12 are also placed very far apart. A better schedule can perhaps achieve the same ILP with less register pressure and fewer spills.

Listing 2. SASS snippet for the unrolled code

```
1  106  /*0328*/      @P0 LDG.E.64 R4, [R24];
2  144  /*0458*/      @P0 STL [R1+0x10], R4;
3  332  /*0a38*/      @P0 LDG.E.64 R12, [R8];
4  350  /*0ac8*/      @P0 LDL.LU R4, [R1+0x10];
5  354  /*0ae8*/      @P0 DADD R16, R16, R4;
6  358  /*0b08*/      @P0 DADD R16, R12, R10;
7  376  /*0b98*/      @P0 DFMA R6, R12, c[0x2][0x40], R14;
8  374  /*0b88*/      @P0 DADD R16, R6, R4;
9  436  /*0d78*/      @P0 DADD R12, R12, R24;
```

Listing 3. SASS snippet for the reordered code

```
1   163  /*04f0*/      @P0 DFMA R14, R22, c[0x2][0x8], R14;
2   164  /*04f8*/      @P0 DFMA R8, R22, c[0x2][0x18], R8;
3   166  /*0508*/      @P0 DFMA R12, R22, c[0x2][0x30], R12;
4   175  /*0550*/      @P0 DFMA R8, R20, c[0x2][0x30], R8;
5   176  /*0558*/      @P0 DFMA R12, R20, c[0x2][0x18], R12;
6   178  /*0568*/      @P0 DFMA R8, R30, c[0x2][0x38], R8;
7   183  /*0590*/      @P0 DFMA R22, R20, c[0x2][0x8], R10;
8   184  /*0598*/      @P0 DFMA R10, R20, c[0x2][0x18], R14;
9   187  /*05b0*/      @P0 DFMA R16, R30, c[0x2][0x20], R12;
10  191  /*05d0*/      @P0 DFMA R10, R30, c[0x2][0x20], R10;
```

Version	reg	IPC	inst. exec.	ld/st exec.	FLOPs	L2 reads	tex txn	tex GB/s
Original	128	1.76	2.74E+9	5.28E+8	1.73E+10	5.27E+8	4.19E+9	899.53
Unrolled	255	1.12	1.36E+9	2.14E+8	1.72E+10	2.94E+8	1.67E+9	457.23
Reordered	64	2.00	1.41E+9	2.14E+8	3.34E+10	1.55E+8	1.67E+9	791.16

Table 2. Metrics for *3d125pt* for tuned configurations

Metrics	rhs4th3fort		hyperm		derivative	
	maxfuse	split-3	maxfuse	split-3	maxfuse	split-2
Inst. Exec.	8.52E+9	8.25E+8	7.48E+8	7.71E+8	8.79E+8	8.96E+8
IPC	1.07	1.11	0.97	1.06	1.02	1.14
DRAM reads	9.07E+7	1.65E+8	1.57E+8	1.77E+8	1.34E+8	2.47E+8
ldst exec.	1.55E+8	1.08E+8	1.27E+8	1.46E+8	1.45E+8	1.30E+8
FLOPs	1.73E+10	1.81E+10	9.66E+9	9.36E+9	1.28E+10	1.34E+10
tex txn.	1.11E+9	8.24E+8	9.72E+8	1.06E+9	1.14E+9	1.01E+9
l2 read txn.	4.64E+8	3.79E+8	6.52E+8	5.90E+8	4.97E+8	4.51E+8
GFLOPS	237.16	274.52	140.71	155.02	168.27	182.83

Table 3. Metrics for reordered max-fuse vs. split versions

Listing 3 shows the SASS snippet for the reordered code generated by our code generator. Using operator distributivity, the multiplication of the coefficient to the additive contributions is converted by our preprocessing pass into fused multiply-adds. Notice that all the uses of register R20 (highlighted in red) are tightly coupled. The same holds for registers R22,R30, and the remaining instructions. Independent FMAs are scheduled together without increasing the MAXLIVE. This reduces register pressure without compromising ILP. Therefore, even though the unrolled version performs fewer FLOPs than the reordered version, we incur less spill LDL/STL instructions per thread (101 for unrolled vs. 7 for reordered).

For the *3d125pt* stencil, table 2 shows some profiling metrics gathered by Nvprof with NVCC. The texture throughput for the original code indicates that the stencil performance is limited by the texture cache bandwidth. Loop unrolling halves the accesses to texture cache and the executed load instructions, but results in a significant drop in IPC due to lowered occupancy. To better expose reordering opportunities after unrolling, the preprocessing pass of the reordering

framework exploits operator distributivity and converts all the contributions in an individual statement to FMA operations. Therefore, instead of 130 FLOPs per stencil point, the reordered version performs 250 FLOPs. As measured by Nvprof, we incur a 2× increase in floating point operations, but achieve significant reuse in registers at a higher occupancy, which consequently improves the IPC and execution time.

Register Pressure Sensitivity In GPUs, the number of registers per thread can be varied at compile time by trading off the occupancy. Many auto-tuning efforts have recently been proposed to that end [24, 32]. Table 4 shows the performance, and the local memory transactions reported by Nvprof, with varying register pressure. Due to space constraints, we only present the data for NVCC compiler. We make the following observations: (a) our optimization strategy reduces the register pressure for all the thread configurations; (b) increasing registers per thread for codes exhibiting very high spills results in better performance, e.g., 8× for *rhs4th3fort*; and (c) for low spills, better performance can be achieved by either increasing occupancy (e.g., reordered code for *3d125pt* and *hypterm*), or maximizing registers per thread (e.g., all the codes for *rhs4th3fort*).

Finding a right balance between register pressure and occupancy is non-trivial, and an active research field [24, 32, 53, 58]. We perform a simple auto-tuning by varying the tile sizes by powers of 2, and varying registers per thread [32]. The best performance in GFLOPS for the auto-tuned code with NVCC and LLVM compilers is shown in figure 6. Unlike the case with 32 and 64 registers per thread, the unrolled code outperforms the original code for all benchmarks, highlighting the importance of loop unrolling and register-level reuse. Our reordering optimization improves the performance by (a) producing a code version that uses fewer registers, and hence can achieve higher occupancy; and (b) helping expose and schedule independent FMAs together for simple accumulation stencils, thereby hiding latency.

Kernel fission From table 1, we select the last three multi-statement, compute-intensive stencils, for which we anticipate high volume of spills in the max-fuse form, and expect kernel fission to be beneficial. For these three stencils, we generate versions with varying degree of splits (Section 3.4). Some splits require promoting the storage from scalars to global arrays, while others require recomputations due to dependence edges in the DAG. Table 3 shows the Nvprof metrics with NVCC for two reordered codes: a version with maximum fusion (max-fuse), and a version with split kernels. Note that in each case, even though the DRAM reads increase going from max-fuse to split version, the IPC also increases. This is because the register pressure per kernel is much lower in the split version, and hence we can unroll the computation to further exploit register-level reuse. This increase in register-level reuse is reflected in the reduced
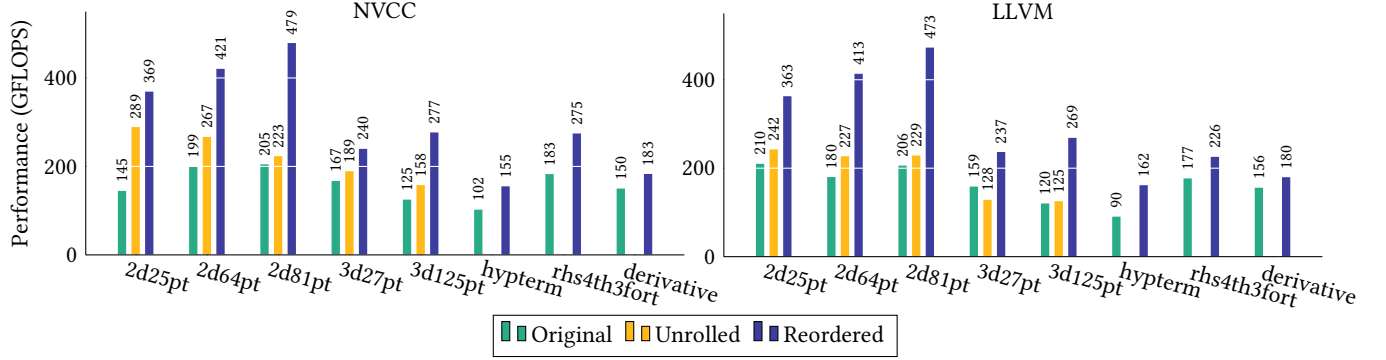
Figure 6. Performance on Tesla K40c with benchmarks tuned for tile size and register limit

L2 read transactions. We observe nearly 10% performance improvement for the split version over max-fuse version for all three stencils. Prior works have noted the importance of kernel fusion for bandwidth-bound stencils [22, 44, 54], and trivial kernel fission to aid fusion by reducing shared memory usage [55]. Such a kernel fission has very limited applicability in stencils with significant many-to-many reuse across statements. However, our motivation for kernel fission is orthogonal to prior efforts – we use kernel fission as a means to reduce the register usage of the max-fuse kernel, and then improve the register reuse for split kernels by ample unrolling and instruction reordering.

With our reordering optimizations applied to the benchmarks, we achieve speedups in the range of 1.22×−2.34× for NVCC, and 1.15×−2.08× for LLVM. We finally discuss the effect of optimizations discussed in Section 3.3. The benchmark *derivative* has a large number of independent trees; each tree is an accumulation. The framework takes 3.42 seconds to generate code for it, and the memoization function is invoked 1.42E+05 times. Without memoization, the code generation time increases to 5.71 seconds. Our framework is well suited to enhance the performance of "optimize once, execute multiple times" stencils found in production codes where the compilation/optimization time is amortized over the stencil execution.

6 Related Work

Register allocation has been extensively studied: from the seminal work of Chaitin [11] on using graph coloring, to the more recent SSA based schemes that exploit the possible decoupling of the allocation phase to the assignment phase [13, 34]. Many extensions/improvements have been applied [33, 42, 48], but almost all the existing works are restricted to the register allocation for fixed schedule. However, it is folk knowledge that improvements to register allocation alone does not bring much performance gain. The interplay of register allocation and scheduling becomes quite important for architectures with ILP, since there is a tradeoff between increasing ILP and exposing locality. Hence, prior work on hyper/super-block list or modulo pre-pass scheduling [12,

Bench.	Reg	Original		Unrolled		Reordered	
		LMT	GFLOPS	LMT	GFLOPS	LMT	GFLOPS
2d25pt	32	1.83E+7	**144.56**	1.18E+8	57.34	4.21E+6	302.12
	48	0	127.35	0	**289.03**	0	357.76
	64	0	115.03	0	261.43	0	**369.09**
2d64pt	32	3.39E+7	111.75	7.17E+8	18.07	6.74E+6	315.49
	48	0	191.17	4.04E+8	26.03	0	393.90
	64	0	**198.95**	2.76E+8	38.89	0	**420.61**
	128	0	146.64	1.31E+7	**231.72**	0	303.02
2d81pt	32	3.94E+7	101.72	8.2E+8	19.49	4.62E+6	426.87
	48	0	202.73	5.13E+8	25.12	0	466.03
	64	0	**204.73**	3.96E+8	30.67	0	**478.95**
	128	0	151.38	7.11E+07	161.00	0	415.30
	255	0	83.58	0	**223,17**	0	340.90
3d27pt	32	4.28E+7	126.07	2.24E+8	37.93	1.65E+7	182.53
	48	0	**167.23**	2.01E+7	149.51	0	229.01
	64	0	160.55	0	172.37	0	**269.69**
	128	0	160.93	0	**180.49**	0	211.78
3d125pt	32	5.04E+7	99.77	2.13E+9	19.06	1.85E+7	327.64
	48	0	96.26	1.42E+9	21.65	0	**339.16**
	64	0	107.61	1.35E+9	25.62	0	336.77
	128	0	**143.12**	5.25E+8	64.00	0	282.68
	255	0	93.64	0	**188.36**	0	173.738
hypterm	32	9.14E+8	21.82	-	-	4.51E+7	83.83
	48	1.04E+8	74.33	-	-	0	100.63
	64	0	100.66	-	-	0	138.12
	128	0	**102.27**	-	-	0	**155.02**
	255	0	74.92	-	-	0	141.52
rhs4th3fort	32	2.10E+9	19.93	-	-	1.37E+9	31.47
	48	1.21E+9	28.76	-	-	4.30E+8	87.82
	64	8.73E+8	38.26	-	-	9.99E+7	171.01
	128	1.60E+8	166.59	-	-	0	241.16
	255	0	**182.67**	-	-	0	**274.52**
derivative	32	1.63E+9	13.54	-	-	6.56E+8	34.25
	48	1.16E+9	17.04	-	-	1.29E+8	84.69
	64	8.90E+8	20.91	-	-	0	116.02
	128	3.60E+8	53.15	-	-	0	153.61
	255	0	**149.95**	-	-	0	**182.83**

LMT: Local Memory Transactions

Table 4. Spill metrics and performance in GFLOPS with NVCC on K40c device for different register configurations

35, 56] were extended to account for register pressure. Other works proposed a reverse scheme where register allocation was made sensitive to not change the minimum initiation interval for the scheduler to expose sufficient ILP [52]. Most of the current mainstream open-source compilers [31, 49] have adopted the first approach: when the register pressure is too high, the pre-pass list scheduling heuristics prioritize scheduling instructions that reduce the register pressure. However, such algorithms lack a global view, focusing only on the local register pressure at the current scheduled point. The associated optimization problem is NP-hard, and it is

known that heuristics implemented in production compilers perform quite poorly on long straight line code [30], such as loop-body of highly unrolled loops.

This observation motivated developers of auto-tuned libraries [18, 19] to consider specific properties of the computational DAG to generate codelets that expose good locality for register reuse at source level: for certain domain-specific applications like FFT, a scheduling that minimizes the spill volume is well understood [27]. The problem addressed in this paper belongs to a similar category, where one has to optimize register reuse for long straight-line code arising from domain-specific kernels. The main difference is that for the computational DAGs considered by our framework, optimal or nearly-optimal scheduling is unknown. Our proposed heuristic is a solution to address the optimization of such DAGs. Prior work on code generation for expression trees [2, 3, 47] were discussed in details in Section 3. We now discuss some related work on combined register allocation and instruction scheduling, and register optimizations for high-order stencils.

Integrated Register Allocation and Instruction Scheduling Motwani et al. [35] show that integrated register allocation and instruction scheduling is NP-hard. They propose a combined heuristic that provides relative weights for controlling register pressure and instruction parallelism. For instructions where the register of an operand can be used for the result, Govindrajan et al. [20] try to generate an instruction sequence from data dependence graph that is optimal in register usage. Berson et al. [7] 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 [40] 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. [37] propose an algorithm that constructs an interference graph with all feasible schedules, and then removes interferences for schedules that are to be least likely followed.

While these efforts consider integrated register management and instruction scheduling, the goals are very different and the contexts quite dissimilar. Prior work in this category has focused on maximizing parallelism without significantly increasing the MAXLIVE. In our context, the main reason for reordering instructions is to effectively exploit the significant potential for many reuses of values held in registers, while reducing the MAXLIVE.

Register Optimizations for High-Order Stencils Stock et al. [50] identify a performance issue with register reuse for iterative stencils by noting that even though the computation becomes less memory-bound with increase in the stencil order, their register pressure worsens. They use a generalized version of [14] to interleave the additive contributions in an unrolled computation. Their approach uses

retiming to homogenize stencil accesses and reduce register pressure, whereas our approach is based on a very different computational abstraction, and is more broadly applicable. Basu et al. [6] propose a partial sum optimization implemented within the CHiLL compiler [23]. The partial sums are computed over planes for 3D stencils, and redundant computation is eliminated by performing array common subexpression elimination (CSE) [15]. However, this optimization is only applicable to stencils with constant and symmetrical coefficients. While their work does not claim to reduce register pressure, it may do so as a consequence of array CSE. Based on the concepts of rematerialization [9], Jin et al. [28] propose a code generation framework that trades off recomputations for reduction in register pressure. It uses dynamic programming to iteratively determine the minimum amount of recomputations required to reduce the register consumption by one. This approach is limited to the stencils described in [28], where the recomputation of an expression does not increase the live ranges of the values involved in it.

In summary, existing approaches targeting register optimization for high-order stencils do not generalize well. Unlike these approaches, our framework optimizes both iterative and more general multi-statement stencils.

7 Conclusion

Despite a rich literature on register allocation and instruction scheduling, current production compilers are not sufficiently effective in reducing register pressure for compute-intensive high-order stencil codes. For such codes, register spills are a major performance limiter. Unfortunately, the compiler fails to perform an instruction reordering that can relieve register pressure, and the reordering it does perform to increase ILP often increases register pressure.

This paper presents a register optimization framework for multi-statement, high-order stencils, which views such computations as a collection of trees with significant data reuse across nodes, and systematically attempts to reduce register pressure by decreasing the simultaneously live ranges. Just as pattern-specific optimization techniques have demonstrably been more beneficial over traditional compiler optimizations for stencil computations, we show through this work that a specialized register management framework can be highly beneficial for stencil computations.

Acknowledgments

References

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. 2007. *Compilers: Principles, Techniques, and Tools (2nd ed)*. Pearson.

[2] A. V. Aho and S. C. Johnson. 1975. Optimal Code Generation for Expression Trees. In *Proceedings of Seventh Annual ACM Symposium on Theory of Computing (STOC '75)*. ACM, New York, NY, USA, 207–217.

[3] A. V. Aho, S. C. Johnson, and J. D. Ullman. 1977. Code Generation for Expressions with Common Subexpressions. *J. ACM* 24, 1 (Jan. 1977), 146–160.

[4] A. W. Appel and K. J. Supowit. 1987. Generalization of the Sethi-Ullman Algorithm for Register Allocation. *Softw. Pract. Exper.* 17, 6 (June 1987), 417–421.

[5] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling Stencil Computations to Maximize Parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 40, 11 pages.

[6] P. Basu, M. Hall, S. Williams, B. V. Straalen, L. Oliker, and P. Colella. 2015. Compiler-Directed Transformation for Higher-Order Stencils. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. 313–323.

[7] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. 1999. Integrated Instruction Scheduling and Register Allocation Techniques. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing (LCPC '98)*. Springer-Verlag, London, UK, UK, 247–262.

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 101–113.

[9] Preston Briggs, Keith D. Cooper, and Linda Torczon. 1992. Rematerialization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. ACM, New York, NY, USA, 311–321.

[10] Preston Briggs, Keith D. Cooper, and Linda Torczon. 1994. Improvements to Graph Coloring Register Allocation. *ACM Trans. Program. Lang. Syst.* 16, 3 (May 1994), 428–455.

[11] G. J. Chaitin. 1982. Register Allocation & Spilling via Graph Coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN '82)*. ACM, New York, NY, USA, 98–105.

[12] J. M. Codina, J. Sanchez, and A. Gonzalez. 2001. A unified modulo scheduling and register allocation technique for clustered processors. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. 175–184.

[13] Q. Colombet, B. Boissinot, P. Brisk, S. Hack, and F. Rastello. 2011. Graph-coloring and treescan register allocation using repairing. In *2011 Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. 45–54.

[14] Raúl De La Cruz, Mauricio Araya-Polo, and José María Cela. 2010. Introducing the Semi-stencil Algorithm. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I (PPAM'09)*. Springer-Verlag, Berlin, Heidelberg, 496–506.

[15] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. 2001. Eliminating Redundancies in Sum-of-product Array Computations. In *Proceedings of the 15th International Conference on Supercomputing (ICS '01)*. ACM, New York, NY, USA, 65–77.

[16] Lukasz Domagala, Duco van Amstel, Fabrice Rastello, and P. Sadayappan. 2016. Register Allocation and Promotion Through Combined Instruction Scheduling and Loop Unrolling. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, New York, NY, USA, 143–151.

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

[18] M. Frigo and S. G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (Feb 2005), 216–231.

[19] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (May 2008), 25 pages.

[20] Ramaswamy Govindarajan, H. Yang, Chihong Zhang, José N. Amaral, and Guang R. Gao. 2001. Minimum Register Instruction Sequence Problem: Revisiting Optimal Code Generation for DAGs. In *Proceedings of the 15th International Parallel &Amp; Distributed Processing Symposium (IPDPS '01)*. IEEE Computer Society, Washington, DC, USA, 26–33.

[21] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. 2014. Hybrid Hexagonal/Classical Tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, Article 66, 10 pages.

[22] Tobias Gysi, Tobias Grosser, and Torsten Hoefler. 2015. MODESTO: Data-centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, 177–186.

[23] Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Murtaza Khan. 2010. Loop Transformation Recipes for Code Generation and Auto-tuning. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing (LCPC'09)*. Springer-Verlag, Berlin, Heidelberg, 50–64.

[24] Ari B. Hayes, Lingda Li, Daniel Chavarría-Miranda, Shuaiwen Leon Song, and Eddy Z. Zhang. 2016. Orion: A Framework for GPU Occupancy Tuning. In *Proceedings of the 17th International Middleware Conference (Middleware '16)*. ACM, New York, NY, USA, 18:1–18:13.

[25] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2013. A Stencil Compiler for Short-vector SIMD Architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*. ACM, 13–24.

[26] HPGMG 2016. High-Performance Geometric Multigrid. https://hpgmg.org/. (2016).

[27] Hong Jia-Wei and H. T. Kung. 1981. I/O Complexity: The Red-blue Pebble Game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing (STOC '81)*. ACM, New York, NY, USA, 326–333.

[28] Mengyao Jin, Haohuan Fu, Zihong Lv, and Guangwen Yang. 2016. Libra: An Automated Code Generation and Tuning Framework for Register-limited Stencils on GPUs. In *Proceedings of the ACM International Conference on Computing Frontiers (CF '16)*. ACM, New York, NY, USA, 92–99.

[29] David Ryan Koes and Seth Copen Goldstein. 2006. A Global Progressive Register Allocator. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 204–215.

[30] Stefan Kral, Franz Franchetti, Juergen Lorenz, Christoph W. Ueberhuber, and Peter Wurzinger. 2004. FFT Compiler Techniques. In *Compiler Construction: 13th International Conference, CC 2004*. Springer Berlin Heidelberg, 217–231.

[31] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–.

[32] A. Li, S. L. Song, A. Kumar, E. Z. Zhang, D. ChavarrÃŋa-Miranda, and H. Corporaal. 2016. Critical points based register-concurrency auto-tuning for GPUs. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1273–1278.

[33] Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai. 2000. Fusion-based Register Allocation. *ACM Trans. Program. Lang. Syst.* 22, 3 (May 2000), 431–470.

[34] Hanspeter Mössenböck and Michael Pfeiffer. 2002. *Linear Scan Register Allocation in the Context of SSA Form and Register Constraints*. Springer Berlin Heidelberg, 229–246.

[35] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. 1995. *Combining Register Allocation and Instruction Scheduling*. Technical Report. Stanford, CA, USA.

[36] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, 429–443.

[37] C. Norris and L. L. Pollock. 1993. A scheduler-sensitive global register allocator. In *Supercomputing '93. Proceedings*. 804–813.

[38] NVCC 2017. NVIDIA CUDA Compiler Driver NVCC. docs.nvidia.com/cuda/cuda-compiler-driver-nvcc. (2017).

[39] NVprof 2017. NVIDIA Profiler. http://docs.nvidia.com/cuda/profiler-users-guide. (2017).

[40] Shlomit S. Pinter. 1993. Register Allocation with Instruction Scheduling. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. ACM, New York, NY, USA, 248–257.

[41] Massimiliano Poletto and Vivek Sarkar. 1999. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.* 21, 5 (Sept. 1999), 895–913.

[42] Fernando Magno Quintão Pereira and Jens Palsberg. 2008. Register Allocation by Puzzle Solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 216–226.

[43] Mahesh Ravishankar, Justin Holewinski, and Vinod Grover. 2015. Forma: A DSL for Image Processing Applications to Target GPUs and Multi-core CPUs. In *Proc. 8th Workshop on General Purpose Processing Using GPUs*. 109–120.

[44] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noel Pouchet, Atanas Rountev, and P. Sadayappan. 2016. Resource Conscious Reuse-Driven Tiling for GPUs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, 99–111.

[45] Hongbo Rong. 2009. Tree Register Allocation. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 67–77.

[46] Vivek Sarkar and Rajkishore Barik. 2007. Extended Linear Scan: An Alternate Foundation for Global Register Allocation. In *Proceedings of the 16th International Conference on Compiler Construction (CC'07)*. Springer-Verlag, Berlin, Heidelberg, 141–155.

[47] Ravi Sethi and J. D. Ullman. 1970. The Generation of Optimal Code for Arithmetic Expressions. *J. ACM* 17, 4 (Oct. 1970), 715–728.

[48] Michael D. Smith, Norman Ramsey, and Glenn Holloway. 2004. A Generalized Algorithm for Graph-coloring Register Allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, New York, NY, USA, 277–288.

[49] Richard M. Stallman and GCC Developer Community. 2009. *Using The GNU Compiler Collection: A GNU Manual For GCC Version 4.3.3*. CreateSpace, Paramount, CA.

[50] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. 2014. A Framework for Enhancing Data Reuse via Associative Reordering. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 65–76.

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

[52] Sid Touati and Christine Eisenbeis. 2004. Early Periodic Register Allocation on ILP Processors. *Parallel Processing Letters* 14, 2 (June 2004), 287–313.

[53] Swapneela Unkule, Christopher Shaltz, and Apan Qasem. 2012. Automatic Restructuring of GPU Kernels for Exploiting Inter-thread Data Locality. In *Proceedings of the 21st International Conference on Compiler Construction (CC'12)*. Springer-Verlag, Berlin, Heidelberg, 21–40.

[54] Mohamed Wahib and Naoya Maruyama. 2014. Scalable Kernel Fusion for Memory-bound GPU Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, 191–202.

[55] Mohamed Wahib and Naoya Maruyama. 2015. Automated GPU Kernel Transformations in Large-Scale Production Stencil Applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*. ACM, New York, NY, USA, 259–270.

[56] Jian Wang, Andreas Krall, M. Anton Ertl, and Christine Eisenbeis. 1994. Software Pipelining with Register Allocation and Spilling. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO 27)*. ACM, New York, NY, USA, 95–99.

[57] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. 2016. gpucc: An Open-source GPGPU Compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. 105–116.

[58] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. 2015. Enabling Coordinated Register Allocation and Thread-level Parallelism Optimization for GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 395–406.

[59] Jingling Xue. 1997. On Tiling as a Loop Transformation. *Parallel Processing Letters* 07, 04 (1997), 409–424.

A   Artifact appendix

Submission and reviewing guidelines and methodology:
http://cTuning.org/ae/submission.html

A.1   Abstract

The artifact comprises an automated framework to perform reordering optimization on stencil computations; the algorithmic details of the framework are described in the PPoPP'18 paper *Register Optimizations for Stencils on GPUs*. The artifact is publicly available for download from the github link *https://github.com/pssrawat/ppopp-artifact*. 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 the reordering framework, and shell scripts to run the benchmarks and verify the results reported in the paper

A.2   Artifact check-list (meta-information)

- **Algorithm:** Register reordering framework for stencil computations on GPU.
- **Program:** C/C++ and CUDA input.
- **Compilation:** CPU code: g++ with c++11 support (GCC 4.9.2 and 5.3.0 tested); GPU code: NVCC (8.0 tested) and LLVM (5.0.0 tested).
- **Transformations:** The framework extracts the stencil statements from the pragma-demarcated input .cu file, performs lowering transformations on the stencil 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, and a GPU device with compute capability >= 3.5.
- **Output:** GFLOPS for all the input benchmarks.
- **Publicly available?:** Yes.

A.3   Description

A.3.1   How delivered

The framework is open-source, and is available for download from the git repository *https://github.com/pssrawat/ppopp-artifact*. 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.

A.3.2   Hardware dependencies

The framework has been tested on Ubuntu 16.04 and Red Hat Enterprise Linux Server release 6.7. The generated code can be executed

on any NVidia device with compute capability >=3.5. For reproducing the results reported in the paper, we suggest using Kepler K40c device.

A.3.3   Software dependencies

  – flex version >= 2.6.0 (2.6.0 tested)
  – bison version >= 3.0.4 (3.0.4 tested)
  – cmake version >= 3.8 for gpucc (3.8 tested)
  – Boost version >= 1.58 (1.58 tested)
  – GCC version >=4.8.1 with c++11 support to compile the framework (4.9.2 and 5.3.0 tested)
  – NVCC version >= 7.5 for benchmarking
  – LLVM version >= 5.0.0 for benchmarking

A.3.4   Data sets

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.4   Installation

First clone the artifact source to a local machine:

    $. git clone *https://github.com/pssrawat/ppopp-artifact*

Then compile the source code to create the executable `test`:

    $. cd ppopp-artifact
    $. make all

Now set some environmental variables that are necessary for benchmarking. The first variable is to indicate the path to CUDA installation, and the second variable is to identify the compute capability of the GPU device:

    $. export CUDAHOME=*path-to-CUDA-installation*
    $. export CAPABILITY=*capability-of-GPU-card*

The compute capability of the K40c device is 35; for a Pascal device, it is 60.

To run the benchmarks:

    $. cd examples
    $. ./run-benchmarks.sh

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

A.5   Evaluation and expected result

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

A.6   Experiment customization

The unrolling factors can be changed in the input .cu file, along with the launch parameters, 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 with the framework.