Automated Loop Fusion for Image Processing

Madushan Abeysinghe¹, Jesse Villarreal¹, Todd T Hahn¹, and Jason D. Bakos¹

¹Affiliation not available

March 19, 2024
Automated Loop Fusion for Image Processing

Madushan Abeysinghe, PhD Candidate, University of South Carolina, Jesse Villarreal, SMTS, Texas Instruments, Todd T. Hahn, SMTS, Texas Instruments, and Jason D. Bakos, Professor, University of South Carolina

Abstract—In this paper, we develop a method for automatically selecting groups of loops to fuse in an image processing data flow graph, here referred to as a “fusing configuration”. The method is designed for use on Digital Signal Processors (DSP), many of which rely on statically scheduled Very Long Instruction Word architecture. Selection is guided by a heuristic scheduler that serves as a performance model for a candidate configuration. We show that for synthetically generated graphs of size 2 to 10 nodes, this approach is capable of selecting the optimal fusing configuration in 80% of graphs and selects a configuration that achieves within 10% of the performance of the optimal configuration for 90% of graphs.

Index Terms—DSL, DSP, Loop Fusion, Image Processing, Instruction Scheduling, Compilers, VLIW, Modulo Scheduling, Software Pipelining

I. INTRODUCTION

D IGITAL Signal Processors are currently the go-to processor type for low-power embedded vision systems, as exemplified by DSP processors integrated into systems-on-chips from NVIDIA, Samsung, Qualcomm, Apple, and Texas Instruments. DSPs differ from general purpose processors and Graphical Processor Units (GPUs) in three ways: (1) they often use a hybrid Very Long Instruction Word (VLIW) and Single Instruction Multiple Data (SIMD) architecture that relies on static instruction scheduling to maximize hardware utilization, (2) they have a complex instruction set, and (3) they use software-defined scratchpad memory and software-defined asynchronous pre-fetching and buffering of data blocks in addition to traditional caches. DSPs come in multiple cores, but they lack support for threads, out-of-order execution, or dynamic out-of-order execution. Examples of such processors include the Google Pixel Visual Core [1], the Qualcomm Hexagon DSP [2], the Nvidia Programmable Vision Accelerator [3], and the Texas Instruments C66x & C71x DSPs [4], [5].

Many image processing applications can be described using a graph-based model that allows the programmer to build processing pipelines by assembling a collection of cooperative image processing kernel primitives. In this way, each graph node represents a kernel and each edge represents the flow of one image or video frame between kernels or as top-level inputs or outputs. This representation potentially allows the run-time environment to identify opportunities for optimization [6].

The Texas Instruments VisionSDK includes an image processing library called VXLIB that targets their DSP architectures. The library consists of a set of primitive image processing kernels. Each kernel has a loop that operates on individual pixels or groups of pixels when multiple pixels are processed via SIMD instructions.

Fusing the loops of multiple VXLIB kernels instead of sequentially executing them allows the kernels to exchange intermediate results through registers instead of through the L1 cache and on-chip scratchpad memory [7]. This reduces the number of load and store instructions, reduces the number of cache misses, and allows the pool of functional units to be shared by multiple kernels within a basic block, increasing the utilization of functional units.

As a VLIW architecture, DSPs exploit instruction-level parallelism by scheduling multiple independent instructions in each clock cycle. Since data dependencies in the loop body would normally make this impractical, DSP compilers perform modulo scheduling to transform the innermost loop in a way to reduce or eliminate intra-loop dependencies and maximize the distance between loop-carried dependencies.

In this paper we propose a technique for fusing VXLIB kernel loops prior to compilation, with the fusing decision being made in a closed-loop process in which candidate sets of loops to fuse are systematically fused and their resultant performance is estimated using a performance model that is capable of estimating the resultant functional unit pressure, register pressure, and achieved loop throughput.

There are two reasons why identifying the sets of kernel loops to fuse is a challenging problem. The first is that some combinations of loops, when fused, cause the compiler to fail when performing modulo scheduling. This is because longer loop bodies may be significantly constrained by register pressure and cause register allocation to fail during scheduling. Second, although the throughput of a fused loop is generally greater than that of the effective throughput of individual loops, our goal is to identify the sets of loops that, when fused, result in the greatest overall throughput improvement for the whole graph as compared to the baseline case of not fusing any loops.

For a given directed acyclic graph (DAG) of VXLIB kernels, our proposed tool searches for an optimal graph decomposition of weakly connected subgraphs that, when the loops of each subgraph are fused, achieves maximum effective throughput.

In order to generate compilable code for merged loops, we developed HeRCide, a library built using C++ metaprogramming that simplifies the process of combining and connecting loop bodies. Additionally, we developed a fast, heuristic scheduler that can estimate the feasibility and performance of a given set of fused loops, based on a database of...
II. BACKGROUND

A. DSP Architecture

Fig. 1 illustrates the simplified microarchitecture of the Texas Instruments C66x DSP. Operating on a Very Long Instruction Word (VLIW) design, it emphasizes instruction-level parallelism by enabling the execution of up to eight instructions in parallel. To address the need for an excessive number of ports on the register file that would otherwise be required for this approach, the design is split into two datapaths, each featuring a register file and four associated functional units. Not depicted in the diagram is a multiplexer situated on the second input of each functional unit, enabling the use of one register from the opposite register file as a second operand when necessary.

In total, there are 64 32-bit registers available. Each functional unit can accept up to two 32-bit registers per operand for SIMD instructions, and the M unit can support up to four 32-bit registers as each operand. The D unit executes load/store instructions and both D units together can allow up to 128 bits to be loaded or stored per cycle to the L1D cache.

Each of the four types of functional units can execute a specific subset of instructions, while some instructions can be executed on more than one unit. For example LD (load) and ST (store) instructions can only be executed on the D units, while the ADD instruction can be executed on the L and S units and the MOVE instruction can be executed on the L, S, and D units. Note that the compiler determines which unit is used for each instruction.

B. Modulo Scheduling

To maximize the utilization of functional units on a VLIW DSP architecture, the compiler must transform the innermost loop in a way that exploits any iteration-to-iteration independence that may exist. To achieve this, compilers use modulo scheduling, a form of software pipelining, which overlaps successive iterations of a loop [9]. The resulting software-pipelined loop contains a prolog, software pipelined kernel ("piped kernel"), and epilog. The prolog, run once, initiates a number of iterations of the loop, "piping-up" the software pipelined loop. Execution then falls into the repeating piped kernel, which starts additional iterations, continues the work of some iterations, and completes some iterations. When no further iterations need to be started, executions falls into the epilog (run once), which completes any iterations that have started but not yet completed.

The objective of this transformation is to maximize the throughput of the loop by minimizing the number of cycles between initiations of each iteration of the loop, or "initiation interval (II)". A loop’s total latency is decomposed into stages, each with length II [8], and the number of iterations that must be executed in parallel is determined by the ratio of loop latency to II, as shown in Eq. 1.

\[
\text{loop throughput} = \frac{1}{\text{initiation interval}(II)} = \frac{\text{iterations in parallel}(IP)}{\text{iteration latency}(IL)}
\]  

The lower bound of the II is determined in part by the resource constraint, or the minimum number of cycles required to execute the instructions in the loop body based on the number of function units available, or ResMII [9]. ResMII depends on the number of instructions comprising the loop body that can only be executed on one of each of individual units \((n_L, n_S, n_M, \text{ and } n_D)\), which we refer to as single unit
instructions, the number of instructions that can be executed on either the L or S units ($n_{LS}$), which we refer to as two-unit instructions, and the number of instructions that can be executed on either the L, S, or D units ($n_{LSD}$), which we refer to as three-unit instructions (note that no instructions can execute on all four units). Since there are two of each unit type, the number of single-unit instructions must be divided by two, the number of two-unit instructions must be divided by four, and the number of three-unit instructions must be divided by six. Thus, the lowest II at which the software pipeliner can reasonably start attempting to schedule is determined in part by Eq. 2:

$$Res_{II} = \max\left(\frac{n_{L}}{2}, \frac{n_{S}}{2}, \frac{n_{D}}{4}, \frac{n_{LS}}{6}, \frac{n_{LSD}}{6}\right) \tag{2}$$

The minimum II at which the software pipeliner can reasonably start attempting to schedule is also determined in part by data dependencies from iteration $i$ to iteration $i + k$. For example, if a value is produced late in iteration $i$, but consumed early in iteration $i + 1$, the start of iteration $i + 1$ may need to be delayed so that the value produced from iteration $i$ is ready by the time iteration $i + 1$ needs it. This iteration-to-iteration dependence is called a loop-carried dependence. Scheduling at a lower II than what the largest loop-carry dictates will result in an illegal software pipeline schedule, and thus the scheduling attempt at that lower II will fail as a result. Because of this, the compiler will compute the recurrence bound of a loop, or $Rec_{II}$. The $Rec_{II}$ is the minimum II in which a loop can be scheduled without violating the iteration-to-iteration data dependence. The actual minimum II is thus the larger of $Res_{II}$ and $Rec_{II}$, i.e. $Min_{II} = \max(Res_{II}, Rec_{II})$.

The minimum number of parallel iterations is determined by the minimum loop iteration latency, or $IL$, which is determined by the critical path latency of the directed acyclic graph (DAG) that describes the dependencies in the assembly code representation of the loop body. However, the critical path delay does not include the effect of additional delays caused by resource constraints. Determining the minimum loop latency when including functional unit constraints requires a combinatorial search for all potential mappings of instructions to functional units and cycles.

Adding to this complexity are two factors. First, various scheduling constraints can cause register lifetimes to be extended, increasing register pressure. When register allocation fails during modulo scheduling, the TI compiler will increase the II and retry modulo scheduling (and register allocation). Second, registers may become live-too-long as a result of the modulo scheduling process, which means a register is longer for II cycles. When registers that are written and read inside the loop are live-too-long, this condition results in a schedule that will not produce the correct results. The TI compiler typically attempts to split any live-too-long registers after a modulo scheduling attempt with live-too-long-split-moves [10], but this does not always work. If live-too-long splitting does not work, the II will need to be increased and modulo scheduling tried again.

III. VXLIB KERNEL LOOP FUSION

In image processing applications, input images are often subjected to a progressive sequence of transformations. For example, an incoming image might be converted to another color space, then have color channels extracted, then be added to another image, then filtered with several convolutions such as blurring and edge extraction, and then compute feature vectors for each pixel. These transformations are often available as separate functions in an image processing library. In our case, we target the Texas Instruments VXLIB library [11].

In this scenario, each successive transformation may be performed with a separate loop but with the same number of loop iterations. On a DSP, each loop would read the input data from an memory hierarchy buffer and store the output data in another memory hierarchy buffer that would be read by a downstream loop.

Fusing any pair of loops where one produces data consumed by another allows two loops to be replaced by a single loop and avoids all the memory transactions that would otherwise be needed to convey the intermediate image between the loops. In other words, the intermediate data would be transferred through registers rather than through memory hierarchy buffers.

Fusing loops in this way improves performance through several mechanisms. First, the fused loop, having more workload, can often exploit more on-chip resources, i.e. achieve higher functional unit utilization. Second, the store and load instruction pairs that would otherwise be needed to store intermediate results to memory hierarchy would be eliminated, reducing the total dynamic instruction count. Finally, reducing memory hierarchy access will correspondingly reduce dynamic stalls resulting from cache misses. Fusing loops also improves energy efficiency by reducing memory system utilization.

Choosing which loops to fuse is a challenge due to several practical considerations. First, maximizing functional unit utilization requires that the loops to be fused have a complementary set of instructions, allowing instructions from one loop to fill empty slots that would otherwise be present in the other loop’s schedule.

Second, optimally statically scheduling loops is an NP-complete problem and is thus sensitive to the size of the loop body. However, typical heuristics that are used to software pipeline loops on VLIW architectures have compile times that increase by at least the square or cube of the number of nodes in the scheduling graph. Therefore, to avoid excessive compile time, the TI compiler limits modulo scheduling to loops that have fewer number of instruction nodes than an experimentally determined threshold. Above this threshold, the compiler will simply schedule the loop without modulo scheduling, leading to poor utilization of the available functional units, and thus poor execution performance.

Third, the size of the register file is a constraint for pipeline scheduling, which can lead to reduced performance for certain combinations of loop bodies.

In order to illustrate how loop fusion can lead to speedup after modulo scheduling, consider the following example. Assume a hypothetical processor with one load/store unit and...
Fig. 2: Modulo schedule for a loop that loads two values, adds them, then stores the result.

Fig. 3: Modulo schedule for a loop that loads a value, increments it, and stores the result.

Fig. 4: Modulo Schedule for the fused loop from Figs. 2 and 3

Consider another loop that only loads one value, increments it, and stores the result back to memory. This loop has \( ResMII = \max\left(\left\lfloor \frac{1}{\mathit{IL}} \right\rfloor, \left\lfloor \frac{1}{\mathit{II}} \right\rfloor\right) = 2 \) and minimum iteration latency (IL) of 5, also requiring \( \left\lfloor \frac{1}{\mathit{IL}} \right\rfloor = \left\lfloor \frac{1}{5} \right\rfloor = 1 \) iterations in parallel to achieve its minimum II of 2, as shown in Fig. 3 and achieving a functional unit utilization of \( \frac{8}{12} = 75\% \).

Assuming both loops would normally be executed in sequence, fusing the loops adds an additional 2 cycles to the minimum iteration latency (IL), caused by the addition of the increment operation that depends on the previous sum operation. However, fusing these loops allows the store instruction from the first loop and the load instruction from the second loop to be eliminated, since the intermediate result (variable c) can be allocated in a register. The resultant loop body has three load/store instructions and two arithmetic instructions, with \( ResMII = \max\left(\left\lfloor \frac{1}{\mathit{IL}} \right\rfloor, \left\lfloor \frac{1}{\mathit{II}} \right\rfloor\right) = 3, IL = 2+2+2+1 = 7, \) and minimum iterations in parallel = \( \frac{7}{3} = 3 \).

A schedule for the fused loop is shown in Fig. 4. It has a resource utilization of 15/18 (18 slots available for 9 cycles with 2 functional units each and 15 occupied slots). The throughput of the fused loop is \( \frac{1}{7} = \frac{1}{5} \) and must be compared the combined throughputs of the two constituent loops, computed as \( \frac{1}{7} + \frac{1}{7} = \frac{2}{7} \). The resultant speedup in terms of throughput achieved from the fusing is this \( \frac{1/5}{1/7} = 1.66 \).

For the fused loop, as shown in Fig. 4 the II is 3, iteration latency (IL) = 9 and iterations in parallel (IP) = 3. It is evident that by fusing the 2 loops, the II did not increase but the functional unit utilization was increased to 15/18 which means the fused loop is doing more work within the same clock cycles as the first loop from Fig. 2. So by fusing these two loops we have achieved a speedup of \( (3+2)/3 = 1.66 \) in addition to eliminating the memory hierarchy accesses by removing the intermediate load/store instructions.

IV. PROPOSED APPROACH

We have developed a framework that accepts an image processing dataflow graph and generates a corresponding C++ code with explicit loop fusing that, when compiled with the TI DSP compiler, produces near-optimal code.
This framework is comprised of three main components: (1) a graph generator that enumerates all possible decompositions of an input VXLIB graph to identify which kernels in the dataflow graph to fuse to maximize performance, (2) HeRCide: a C++ code generator and corresponding target library that takes an input graph comprised of VXLIB kernels and generates a single loop encompassing the functionality of multiple fused VXLIB kernels as executable code, and (3) VXOPT: a heuristic modulo scheduler and a library of assembly-level single scheduled iterations corresponding to each VXLIB kernel.

A. Graph Generator

For a given image processing application described as a graph of VXLIB kernels, our objective is to find the graph decomposition—that is, the assignment of each kernel to a kernel group—that when each group of kernels is synthesized as a single loop and scheduled, achieves the best overall speedup over the baseline case of executing each kernel as a separate loop.

The upper bound for the maximum number of decompositions is defined by the Bell number, defined in Eq. 3 which is an exponential function. In practice, since each kernel group must be weakly connected in order for the grouping to allow for the elimination of load and store instructions corresponding to the values conveyed between grouped kernels, the total number of decompositions is potentially small enough to be enumerated and evaluated using a fast performance model, but too large to evaluate on the DSP.

\[
B_n = \sum_{k=0}^{n-1} \binom{n-1}{k} \times B_k
\]

\[
B_0 = 1
\]  

(3)

Our graph generator enumerates all weakly connected subgraphs for a given application and estimates the resultant speedup against a performance model that uses a heuristic to compute a modulo schedule for the fused loop. The heuristic scheduler contains an instruction-level dataflow graph for each VXLIB kernel. When the optimal decomposition is found, the resulting fused loops are composed in C++ using a template library called HeRCide.

B. HeRCide

HeRCide is designed as a DSP-centric analog to Halide \[12\], is a hierarchical C++ library containing a PixelLib, a template function corresponding to the single iteration workload associated with each VXLIB kernel. As such, a fused loop can be generated by instancing calls to these methods in a single loop body. Since the C66x DSP contains Single Instruction, Multiple Data (SIMD) support, the PixelLib is built on top of a SIMD datatype library called the “core functions” library, which contains assembly-level implementations of primitive operations for SIMD types.

This stack is shown in Fig. 5 in which a synthesized fused loop is comprised of one call to the PixelLib library for each kernel being fused, and the PixelLib functions having template parameters corresponding to whatever primitive SIMD type is being processed by each kernel.

This approach requires that the kernels being fused have the same number of loop iterations, which consequently requires that all the included kernels use the same SIMD width.

An example HeRCide core function class for the 8-element 8-bit unsigned SIMD type is shown in Code. It implements the primitive “bitwise-or” and “add” functions. Note that the operators are overridden and include processor-specific vectorized intrinsics that map to specific SIMD instructions.

The set of VXLIB kernels currently supported by HeRCide is presented in Table. 1

An example set of PixelLib routines is shown in Code. 2 Code. 3 shows an example loop generated from fusing the “VXLIB_Add” kernel and “VXLIB_Or” kernel. This code, which comprises the top-level loop and several compiler pragmas, is machine-generated based on the kernels and interconnections given by a graph-based input description.

The VXLIB_ADD kernel has two external inputs named src1 and src2 and the output is connected to one of the inputs of the VXLIB_Or kernel, with the other connected to an external input named src3. The output of the VXLIB_Or kernel is an external output.

The "nassert" statement indicates to the compiler that the arrays are aligned and can be accessed with load and store instructions that require aligned addresses. The “UNROLL” pragma tells the compiler exactly how many times to unroll the loop and the “MUST_ITERATE” pragma indicates minimum and maximum trip counts. Within the kernel “for loop” customized SIMD load instructions are generated from the overloaded assignment (=) operators. Calls to the PixelLib library functions and their corresponding inputs and outputs are connected according to the graph structure shown in Fig.

Fig. 5: HeRCide Architecture

Fig. 6: Graph structure for the fused add-or kernels
The output is stored using the store function call which includes a SIMD store intrinsic.

**Code 1** Example type library of “Core Funcions” for uint8x8_t type

```cpp
class uint8x8_t {
public:
    uint64_t data;
    uint8x8_t(uint64_t val=0) : data(val) {}  
    uint8x8_t() {}  
    inline uint8x8_t operator= (const uint8_t * restrict input) {
        data = _amem8_const(input); 
        return uint8x8_t(data); 
    }
    inline uint8x8_t operator| (const uint8x8_t op1) {
        uint64_t t0 = data | op1.data; 
        return result(t0); 
    }
    inline uint8x8_t operator+ (const uint8x8_t op1) {
        uint64_t t0;
        t0=_itoll(_add4(_hill(data),
                       _hill(op1.data)),
                       _loll(data),
                       _loll(op1.data)); 
        return uint8x8_t(t0); 
    }
    inline void store (const uint8_t * restrict output) {
        _amem8((void *)output) = data; 
    }
};
```

**Code 2** PixelLib library code for ‘add’ and ‘or’ kernels

```cpp
template <typename simd_width>
static inline
void pixellib_add(simd_width src0,
                   simd_width src1,
                   simd_width &dst) {
    dst = src0 + src1;
}

template <typename simd_width>
static inline
void pixellib_or(simd_width src0,
                 simd_width src1,
                 simd_width &dst) {
    dst = src0 | src1;
}
```

**Code 3** Generated code for fused ‘add’ and ‘or’ kernels

```cpp
void vxAddOrKernel(
    const uint8_t * restrict src1,
    const uint8_t * restrict src2,
    const uint8_t * restrict src3,
    uint8_t * restrict dst,
    uint32_t width) {
    uint8x8_t src1a, src2a, src3a, dsta, t0;
    _nassert(((uint32_t)src1 % 8U) == 0);
    _nassert(((uint32_t)src2 % 8U) == 0);
    _nassert(((uint32_t)src3 % 8U) == 0);
    _nassert(((uint32_t)dst % 8U) == 0);
    #pragma UNROLL(1)
    #pragma MUST_ITERATE(0,,2)
    for (int x=0; x<width; x++) {
        #pragma FORCEINLINE_RECURSIVE
        src1a = &src1[x*8];
        #pragma FORCEINLINE_RECURSIVE
        src2a = &src2[x*8];
        #pragma FORCEINLINE_RECURSIVE
        src3a = &src3[x*8];

        #pragma FORCEINLINE_RECURSIVE
        pixellib_add<uint8x8_t>(src1a, src2a, t0);
        #pragma FORCEINLINE_RECURSIVE
        pixellib_or<uint8x8_t>(t0, src3a, dsta);
        #pragma FORCEINLINE_RECURSIVE
        dsta.store(&dst[x*8]);
    }
}
```

### C. VXOPT

Fusing kernels has the potential to achieve substantial speedup for VXLIB graphs, but identifying sub-graphs to fuse requires a model to predict both the feasibility and performance impact of a proposed fusing strategy.

Certain combinations of kernels cannot be fused because the vendor’s DSP compiler will fail to modulo schedule loops in which the hardware constraints make it difficult or impossible to find a valid schedule having more than one iteration in parallel. The most common cause of this is when there is an insufficient number of registers to hold all the needed live values or when the iteration latency must be increased to accommodate the schedule to the point where the II becomes equal to the latency of the single scheduled iteration. In other words, it is not feasible to fuse a large set of kernels.

Even for kernel sets that can be fused, kernel combinations that have complementary functional unit requirements or ones that are tightly interconnected will achieve a greater speedup than combinations that lack these qualities.

Thus, in order to maximize speedup through kernel fusing, it is necessary to have knowledge of which kernel combinations can be fused and which kernels combinations when fused give the best speedup.

For this reason, we propose a performance model that will take, as input, a set of kernels and predict the initiation interval...
that the compiler would achieve for a loop that contains the combined workload of the fused kernels, minus the load and store instructions that would have been required to convey intermediate results if the kernels were not fused. Ideally, this model would be substantially faster than compiling the fused loop. For this reason, we have developed a heuristic scheduler that approximates the modulo scheduling algorithm.

V. VXLIB GRAPH OPTIMIZATION

For a given set of VXLIB kernels and associated graph, our performance model will fuse the corresponding instruction-level dataflow graphs of each kernel, compute the As Soon as Possible (ASAP) and As Late as Possible (ALAP) cycles for each instruction, construct a modulo schedule for a loop containing the kernels, and report the resulting initiation interval, schedule, and register usage table. The reported II is used to estimate the achieved throughput of the fused VXLIB kernels and is compared against other candidate fusing strategies.

Our scheduler is substantially faster (30 to 60x) than the TI compiler because it does not require C/C++ parsing, it does not perform register allocation or code generation, and it uses a heuristic scheduler. For these reasons, it is not guaranteed to produce results that are consistent with the TI compiler, but is intended to deliver sufficient accuracy to significantly outperform a random fusing strategy.

A. Merging DAGs

The single scheduled iteration of each VXLIB kernel is stored in a database and, when used, converted to a DAG in which each vertex corresponds to each assembly instruction and each edge represents a dependency between two instructions. To fuse two or more VXLIB kernels that are weakly connected in a VXLIB graph, their corresponding DAGs are combined by matching the edges in the VXLIB graph to the corresponding load and store instructions in the assembly-level DAGs for each kernel.

In other words, each external input and output from each VXLIB kernel is associated with a load or store instruction, respectively. When fusing two kernels in which kernel A's output is connected to the kernel B's input, the corresponding store instruction in kernel A and the corresponding load instruction in kernel B are removed and replaced with a single DAG edge.

Fig. 7 shows an example VXLIB graph in which the VXLIB absDiff kernel fans out to two VXLIB OR kernels. Figs. 9a and 9b shows the instruction-level DAGs for the absDiff and OR kernels, respectively.

When fusing these 3 kernels, the store instructions corresponding to the output of the absDiff are removed, and the load instructions corresponding to the inputs of the OR kernels are removed. Fig. 10 shows the merged DAG for the resultant "absDiff-OR-OR kernel".

B. Performance Model Design

To identify the most efficient set of subgraphs in which to decompose a VXLIB graph, our proposed method performs a search over the decomposition space, evaluating each decomposition with our performance model. Our performance model uses a heuristic to schedule the fused DAG to determine the feasibility of scheduling and the resultant iteration interval (II).

A modulo scheduler must make several decisions when constructing a schedule, each of which represents a branching point in the search space. For example, many instructions can be scheduled on more than one functional unit type, for a given function unit, instructions can be scheduled on either the A-register path or the B-register path, and there may be a difference between the earliest and latest cycle in which an instruction may be scheduled. For this reason, finding an optimal schedule requires a combinatorial search. After the schedule is constructed, the compiler must allocate registers, the decisions of which comprise its own combinatorial space.

In our proposed performance model, we use a scheduling algorithm without a register allocation phase, which in practice produces a sufficiently high quality of results to prune away virtually all unpromising VXLIB graph decompositions.

1) Schedule Initialization: The modulo schedule is made up of functional unit slots, the number of which is $II \times n_{units}$, where $II$ is the initiation interval or the inverse of the loop throughput. The number of units available in the C66x DSP is eight, namely L1, L2, D1, D2, S1, S2, M1, M2. Each unit can be issued one instruction per cycle. Each issued instruction performs workload on behalf of one of $IP$ loop iterations, in which $IP$ is the number of iterations processed in parallel. Recall that $IP$ is a function of $II$ and $IL$, where $IL$ is the loop iteration latency as shown in Eq. [1].

2) Register Allocation: The number of live registers in each of the $II$ modulo cycles is an important constraint when scheduling, as the number of live registers cannot exceed 64. For each edge in the DAG connecting instructions $a$ and $b$, when $a$ is scheduled in modulo cycle $a_{cycle}$ and relative iteration $a_i$ and instruction $b_i$ is scheduled in modulo cycle $b_{cycle}$ and relative iteration $j$, and given the latency of instruction $a$ is $a_{latency}$, the number of live registers in cycles 

\[(a_{cycle} - a_{latency}) + a_i \times II \mod II + (b_{cycle} + b_i \times II) \mod II \]

is incremented by the number of 32-bit values conveyed on that edge.

In other words, a value is live from the time it is written to the time it is last consumed. Thus, the number of live registers...
in cycle \( c \) is incremented for each value written \( n \) cycles prior to cycle \( c \), where \( n \) is the latency of the producing instruction, and for which at least one of the instruction’s successors is scheduled on or after cycle \( c \). This way, the register count is incremented in the cycles between producing and consuming instructions, not including the cycles covered by the latency of the producing instruction.

3) Register Live-to-Longs: Additionally, a live register cannot persist for more than II cycles without being transferred via a move instruction to another register, since each live register can only be associated with one loop iteration at any moment. In other words, if an instruction consuming a value is scheduled more than II cycles away from an instruction which produces that value (also taking into account the latency of the producing instruction to account for when the register is written), a live-too-long situation results. Without intervention, the register written by the dependency will be invalidated before it can be used. Inserting an intervening move instruction will remedy this problem but the move instruction will occupy a scheduling slot.

Figs. 9 shows how the schedule from Fig. 4 could be alternatively generated in a way that includes a register live-too-long. In this case, the number of iterations in parallel has increased from 3 to 4. A register must be allocated to hold the value of \( d \) when it becomes available two cycles after the “\( d = c + 1 \)” instruction that generates it. This register is read by the “store \( d \)” instruction four cycles later. This requires the register holding the value of \( d \) to be alive for 4 cycles, which is greater than the II of 3. This is a “register live-too-long” problem, because that register would be invalidated by the “\( d = c + 1 \)” instruction corresponding to the next loop iteration 3 cycles later. Our proposed performance model detects registers that are alive too long, and attempts to repair by inserting move instructions to split the lifetime of the register that is live too long.

This is shown in Fig. 10, in which the “\( d1 = d \)” instruction is inserted into an empty slot two cycles after the “\( d = c + 1 \)” instruction and four cycles prior to the “store \( d1 \)” instruction that reads it. Assuming that latency of the move instruction is one cycle or more, the lifetime of the register allocated to the \( d1 \) variable is guaranteed to be less than or equal to the II of 3.

C. Heuristic Scheduler

Our heuristic instruction scheduler is shown in Alg. 4. It takes as input the fused assembly instruction DAG. It begins by computing the resource minimum II (\( ResMII \)) as described in Section II-B and computes the “As Soon As Possible (ASAP)” and “As Late As Possible (ALAP)” cycle for each instruction in the DAG.

The ASAP cycle of each instruction is computed as the sum of instruction latencies corresponding to its longest chain of dependencies. The ALAP cycle for all store instructions is set to the maximum ASAP cycle among all the instructions that have no output (i.e. the terminating instructions). Finally, the DAG is traversed in reverse order, setting the remaining ALAP values to the minimum ALAP value of each instruction’s successors, minus its instruction latency. The critical path consists of all instructions that have the same ASAP and ALAP values.

An example of this is shown for the DAG in Fig. 8, under the simplifying assumption that all instructions have a latency of two cycles. In this case, all instructions are part of the critical path except for store2.

After computing \( ResMII \) and the ASAP and ALAP cycles for each instruction, our heuristic first sorts the critical path instructions using \(-ALAP\) as the sort key. Second, any pair of instructions A and B in which instruction A is a critical path instruction and instruction B is a non-critical path instruction but a successor of A, the algorithm will insert instruction B before A in the sorted list.

Next, the non-critical path instructions are added to the tail of the sorted list using the edge distance to its nearest critical path instruction as the primary sort key and the ALAP value as the secondary sort key. In the example shown in Fig. 8, the resulting sorted order is store1, add2, store2, add1, load1, and load2.

The algorithm then allocates one slot for each functional unit for each of the \( ResMII \) cycles. The schedule is con-
Fig. 8: Example of scheduling heuristic.

Fig. 9: Modulo Schedule for the fused loop with an inferred register alive too long.

Fig. 10: Modulo Schedule for the fused loop where the register alive too long has been resolved.
Algorithm 4 Heuristic Scheduler

1: Input: inst, dag
2: Output: sched
3: function SCHED_INST(inst, dag)
4: for each i ∈ [0, ii − 1], unit ∈ {L|S|D|M} do
5: place inst into sched(i, unit)
6: if avail_units > calc_units_needed(dag) then
7: if is_reg_press_high(sched) then
8: exit recursion
9: else
10: SCHED_INST(inst_next, dag)
11: end if
12: else
13: return NULL
14: end if
15: end for
16: if are_reg_live_too_long(sched) then
17: mv = insert_mv_instructions(dag)
18: is_success = sched_mv_inst(mv, sched)
19: if is_success then
20: return sched
21: else
22: exit recursion
23: end if
24: end if
25: return sched
26: end function

D. Making Scheduling Decisions and Pruning the Search Space

As described in Sec. V-B1, the C66x DSP has eight functional units, two of each type, and many instructions can be scheduled on multiple function unit types. Additionally, each instruction can potentially be scheduled in II possible cycles, since the iteration number attached to the instruction can be used to meet dependency constraints.

Our scheduling heuristic uses a depth-first branch-and-bound search when searching for a schedule. For each instruction to be scheduled, the heuristic recurses on every available cycle and every available compatible functional unit.

For each candidate instruction placement, the algorithm performs a feasibility test to determine if there is a sufficient number of remaining functional unit slots to accommodate the unscheduled instructions. If not, the partial schedule is rejected and the search space under it is pruned, at which point the heuristic backtracks to the most recent decision point and selects a different unit and/or cycle for the last-scheduled instruction.

Additionally, each time an instruction is placed on the schedule, the heuristic computes the number of live registers scheduled in reverse order by choosing and scheduling instructions in sorted order. The cycle numbers are labeled in descending order starting from 0.

1) Scheduling Heuristic Example: For the example assembly instruction DAG shown in Fig. 8a, each instruction is labeled with its corresponding ASAP and ALAP cycles and the instructions are sorted in scheduling order as described above. The ResMI for this DAG is 4 cycles, based on the four load/store instructions and the availability of only one load/store unit in the simplified architecture used for this example.

store1 is the first instruction to be scheduled. The scheduler can place this instruction in any cycle since it has no successor instructions. As shown in Fig. 8a, our heuristic allocates the instruction to cycle 0, the latest cycle in the schedule. This instruction is associated with loop iteration 0, which is meant to indicate a loop iteration number that is relative to the iteration number associated with the other instructions in the schedule.

Next, the heuristic schedules the add2 instruction—a dependency of the store1 instruction—which must be placed at least two cycles prior to the store1 instruction. Our heuristic places it in the latest possible position, cycle -2 as shown in Fig. 8a.

Next, as shown in Fig. 8b, the heuristic schedules the store2 instruction in the latest possible cycle in which the corresponding functional unit is available, -1, since it has no successors.

Next, as shown in Fig. 8c, the heuristic schedules the add1 instruction. The add1 instruction must be scheduled no less than two cycles earlier than its successors, add2 and store2, which are already scheduled in cycles -2 and -1, respectively.

The latest cycle would therefore be -4. However, since this would require more than II cycles, the heuristic must place the add1 instruction in a cycle of another iteration of the piped kernel, pushing it into cycle 0 of the previous iteration. In the figure, both iterations are shown in the schedule table in order to depict the flow of data across multiple iterations of the loop. Note that each iteration of the piped kernel must execute the same instructions, but intermediate results flow across loop boundaries. These are indicated by the dotted lines.

Finally, as shown in Fig. 8d, the heuristic schedules the load1 instruction, which must be scheduled no later than two cycles earlier than its immediate successor, add1, so the heuristic schedules it in the latest available cycle -2. Note that this instruction is placed in cycle 2 of each iteration, as shown in the schedule table.

The latest cycle would therefore be -4. However, since this would require more than II cycles, the heuristic must place the add1 instruction on cycle 0 of another iteration of the piped kernel, pushing it into cycle 0 of the previous iteration. In the figure, both iterations are shown in the schedule table in order to depict the flow of data across multiple iterations of the loop. Note that each iteration of the piped kernel must execute the same instructions, but intermediate results flow across loop boundaries. These are indicated by the dotted lines.

For each candidate instruction placement, the algorithm performs a feasibility test to determine if there is a sufficient number of remaining functional unit slots to accommodate the unscheduled instructions. If not, the partial schedule is rejected and the search space under it is pruned, at which point the heuristic backtracks to the most recent decision point and selects a different unit and/or cycle for the last-scheduled instruction.

Additionally, each time an instruction is placed on the schedule, the heuristic computes the number of live registers
at each of the II cycles and prunes the partial schedule if any cycle exceeds a given register number threshold. When expanding scheduling choices, the heuristic uses a cycle-priority, then unit-priority order, meaning that it first attempts to schedule an instruction in the latest possible cycle for each unit type in a given order and then repeats this procedure for progressively earlier cycles.

If the search space is exhausted without finding a valid schedule for the ResMII, then the scheduler abandons the schedule, increases the II by one, and repeats the scheduling process. If the II reaches the latency of the single scheduled iteration, the algorithm reports a failure and exists.

After a schedule is found, the heuristic identifies any register live-to-longs and attempts to split the register lifetime by inserting a move instruction into the schedule. Move instructions can be scheduled on the L, S, or D units. If none of these units are available in the range of cycles required to split the register lifetime, the scheduler abandons the schedule increases the II and repeats the schedule process.

VI. RESULTS

Table II shows the potential speedup gained from fusing VXLIB graphs as well as the accuracy of our proposed performance model. Each row of the table summarizes the results from up to 500 random VXLIB graphs of the size given in column 1. These results were generated by fusing all the valid sub-graph decompositions for each of the randomly generated graphs using our “HeRCide” automated loop fusing library.

The column labeled “Average Number of Fusing Configurations” indicates the average number of ways the randomly generated graphs can be decomposed. Note that the number of decompositions depends on the graph structure since all sub-graphs in a decomposition must be weakly connected. Thus, graphs that are more densely connected will generally have a greater number of decompositions.

The column labeled “Median Potential Speedup” indicates the median best speedup that can potentially be obtained from any fusing configuration. In other words, it is the speedup one may expect if it were known for each graph which decomposition achieves the best performance. Speedup is measured relative to the baseline performance given the graph without kernel fusing, as shown in Eq. 4.

Likewise, the column labeled “Average Potential Speedup” indicates, the average best speedup achieved for all graphs by selecting the set of sub-graphs achieving the best performance for each graph when evaluated for II for by the vendor compiler. In other words, this represents the speedup a user would expect if it were feasible to evaluate all sub-graph enumerations using the compiler.

\[
\text{speedup} = \frac{\text{Sum of Initiation Intervals (II) for the baseline graph with NO fused groupings}}{\text{Sum of Initiation Intervals (II) for the graph with fused groupings}}
\]  \hspace{1cm} (4)

The column labeled “Average Prognosticated Speedup” shows the speedup achieved on average if a user selects the fusing configuration suggested by the proposed heuristic scheduler, as opposed to the best overall fusing configuration. Compared to the previous column, this indicates how the accuracy of the heuristic scheduler affects the overall optimization problem of selecting the best fusing configuration. In other words, in cases where the heuristic scheduler does not exactly identify the best performing configuration (due to prediction error), it is still able to select a closely performing configuration.

Likewise, the “Average Heuristic Predicted Best Speedup” is the average speedup reported by the heuristic scheduler. This column, compared to the “Average Potential speedup” column, indicates the overall accuracy of the heuristic scheduler.

The column labeled ”Max Potential Speedup” shows the speedup of the highest-performing fused graph among all those tested for each graph size, showing the maximum potential for speedup using kernel loop fusing. The column labeled “% Opt. Config. Found” shows the percentage of graphs for which the performance model was able to select the optimal fusing configuration. Note that, while this percentage trends downward as the graph size is scaled up, it does not proportionally affect the average prognosticated speedup, which indicates that non-optimal fusing configurations can still achieve near-optimal performance. The “Actual vs Prognosticated Error %” shows the relative difference in performance between selecting the best-performing configuration based on performance given by the compiler or the heuristic scheduler.

Fig. 11 shows the distribution of speedups for all the 4,606 fusing configurations for a randomly-chosen 10-node VXLIB graph, computed from the output of the DSP compiler. The highest potential speedup achieved for this graph is 2.00, but only 4 out of 4,606 fusing configurations achieve a speedup of 2.00. This result illustrates the rarity of optimal graph fusing configurations.

VII. RELATED WORK

Qui et al proposed a kernel fusion technique based on graph partitioning problem using the minimum cut technique to recursively search fusible kernels [13] [14]. The fusible
kernels are selected based on the weights of the edges in the graph, assigned based on a benefit estimation model. They implemented their solution within HIPAcc, an image processing DSL and source-to-source compiler targeted for GPUs [15]. Kennedy and McKinley’s [16] loop-fusing approach reorders the loop or changes the order in which loop iterations are executed by preserving output dependencies and then fuses multiple distinct loops into a single loop. Their graph partitioning approach is based on maximum-flow/minimum cut. Meng et al developed a method to fuse parallel kernels—as opposed to loops—onto the same GPUs to improve data locality [17]. This reduces DMA accesses to the memory hierarchy and improves data locality in L1 cache. Intel proposed a loop-fusing technique for nested loops to maximize memory locality [18]. They convert the source code to a DAG (Directed Acyclic Graph) and merge different DAGs before converting the DAGs back to source code.

Fraboulet et al developed a loop fusing approach designed to improve data reuse [19]. Kernel fusion could be potentially advantageous in data-intensive applications like data warehousing. Wu et al evaluated the benefits of kernel fusion on relational algebra operations implemented using CUDA for an NVIDIA Fermi GPU [20]. Fousek et al decompose large functions into smaller kernels and then evaluate which kernel combinations perform best when fused [21]. Their predictor model focuses on computation against memory access time. Xue et al. propose a technique to fuse loops while removing fusion-preventing dependencies by applying loop tiling [22]. Mehta et al developed a method to perform loop transformations using the polyhedral framework [23].

<table>
<thead>
<tr>
<th>Node Count</th>
<th>Average Number of Fusing Configurations</th>
<th>Average Potential Speedup</th>
<th>Average Prognosticated Speedup</th>
<th>Average Heuristic Predicted Best Speedup</th>
<th>Max Potential Speedup</th>
<th>% Opt. Config. Found</th>
<th>Actual vs Prognosticated Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>1.33</td>
<td>1.48</td>
<td>1.48</td>
<td>1.55</td>
<td>4.00</td>
<td>98%</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1.60</td>
<td>1.64</td>
<td>1.62</td>
<td>1.72</td>
<td>3.00</td>
<td>95%</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>1.67</td>
<td>1.71</td>
<td>1.69</td>
<td>1.81</td>
<td>3.00</td>
<td>89%</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>1.71</td>
<td>1.80</td>
<td>1.76</td>
<td>1.91</td>
<td>3.33</td>
<td>82%</td>
</tr>
<tr>
<td>6</td>
<td>41</td>
<td>1.70</td>
<td>1.81</td>
<td>1.74</td>
<td>1.92</td>
<td>3.50</td>
<td>74%</td>
</tr>
<tr>
<td>7</td>
<td>85</td>
<td>1.73</td>
<td>1.84</td>
<td>1.79</td>
<td>1.98</td>
<td>4.00</td>
<td>66%</td>
</tr>
<tr>
<td>8</td>
<td>208</td>
<td>1.69</td>
<td>1.80</td>
<td>1.70</td>
<td>1.93</td>
<td>5.00</td>
<td>53%</td>
</tr>
<tr>
<td>9</td>
<td>483</td>
<td>1.74</td>
<td>1.83</td>
<td>1.72</td>
<td>1.98</td>
<td>3.14</td>
<td>48%</td>
</tr>
<tr>
<td>10</td>
<td>1216</td>
<td>1.86</td>
<td>1.90</td>
<td>1.78</td>
<td>2.08</td>
<td>3.00</td>
<td>49%</td>
</tr>
</tbody>
</table>

TABLE II: Comparison of average actual speedup values vs heuristic predicted speedup values for up to 500 graphs per each node count.

In this paper, we describe a method for quickly estimating the performance given by fusing two or more image processing loops taken from a graph composed of VXLIB image processing kernels (part of the Texas Instruments Vision SDK). The performance model allows for the estimation of end-to-end performance of the graph when decomposed into multiple weakly connected fused subgraphs. Using our code synthesis tool HeRCide we can generate fused loops as C++ code for graph deployment.

Our performance model is based on modulo scheduling the instructions comprising the body of the fused loop. The scheduler is a heuristic that trades off less than 10% accuracy, notably by searching only a portion of the schedule space and skipping the register allocation phase, allowing for the rapid exploration of a large graph decomposition space. Our results indicate that the accuracy of the performance model is sufficient to often select the optimal decomposition and is demonstrated to select a decomposition that is within 10% of the optimal decomposition on average for 90% of the graphs we tested.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under Grant No. 1910748 and through financial, hardware, and intellectual support from Texas Instruments Inc.

REFERENCES


IX. BIOGRAPHY SECTION

Madushan Abeysinghe is currently a PhD candidate in Computer Science and Engineering and a member of Heterogeneous and Reconfigurable Computing (HeRC) Lab under the advice of Professor Jason D. Bakos at University of South Carolina, Columbia, SC. He received his B.Sc. in Electronic and Telecommunication Engineering from University of Moratuwa, Sri Lanka and B.Sc. in Business IT from University of Greenwich, United Kingdom. His research interests focus on DSPs, FPGAs, DSLs, Image Processing, Compilers, Machine Learning in the Jacinto Processors product line at TI, and serves as a member of the Khronos OpenVX working group.

Jesse Villarreal received the B.S. degree in Computer and Electrical Engineering at Baylor University in 2002, and the M.S. degree in Computer Engineering at the University of Texas at Dallas in 2004. Since then, Jesse has worked at Texas Instruments (TI) in various software roles in the imaging and vision domains, supporting embedded processing in camera, phone, and automotive ADAS processors. He has authored 5 conference papers, and has been awarded four U.S. patents related to his work at TI. He is currently a software architect in the Jacinto Processors product line at TI, and serves as a member of the Khronos OpenVX working group.

Todd T. Hahn received a B.S. in Computer Science from Michigan Technological University in 1997 and an M.S. in Computer Sciences from the University of Wisconsin in 1999. He has worked in the Texas Instruments’ compiler team for 24 years and holds seven U.S. patents. Todd aids the development of highly-optimizing compilers for TI’s high-performance VLIW DSP architectures, specializing in the areas of instruction scheduling, software pipelining, compiler loop transformations, and instruction set design and evaluation.

Jason D. Bakos (M’96) received the B.S. degree in Computer Science from Youngstown State University, Youngstown, OH, in 1999, and the Ph.D. degree in Computer Science from the University of Pittsburgh, Pittsburgh, PA, in 2005. He is currently a Professor of Computer Science and Engineering at the University of South Carolina, Columbia. Prof. Bakos’s research focuses on embedded and domain-specific architectures, including those based on reconfigurable, graphical, many-core, digital signal, automotive, and neuromorphic processor technology. Dr. Bakos has published over 70 refereed publications, a textbook on ARM embedded system programming, and two U.S. patents. He received the US National Science Foundation CAREER award and his work is supported by NSF, DOE, ONR, and Texas Instruments Corporation. He currently serves as associate editor for ACM Transactions on Reconfigurable Technology and Systems (TRETS).