Multithreaded Pipeline Synthesis for Data-Parallel Kernels

Mingxing Tan[1], Bin Liu[2], Steve Dai[1], Zhiru Zhang[1]

[1]School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
[2]Facebook, Inc., Menlo Park, CA
mingxing.tan@cornell.edu, binliu@fb.com, stevedai@csl.cornell.edu, zhiruz@cornell.edu

Abstract

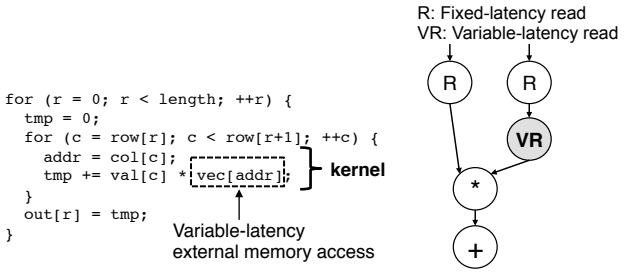
Pipelining is an important technique in high-level synthesis, which overlaps the execution of successive loop iterations or threads to achieve high throughput for loop/function kernels. Since existing pipelining techniques typically enforce in-order thread execution, a variable-latency operation in one thread would block all subsequent threads, resulting in considerable performance degradation. In this paper, we propose a multithreaded pipelining approach that enables context switching to allow out-of-order thread execution for data-parallel kernels. To ensure that the synthesized pipeline is complexity effective, we further propose efficient scheduling algorithms for minimizing the hardware overhead associated with context management. Experimental results show that our proposed techniques can significantly improve the effective pipeline throughput over conventional approaches while conserving hardware resources.

1. Introduction

As modern computation becomes increasingly limited by power, many systems rely on hardware accelerators for computation-intensive applications, such as matrix computation and multimedia processing. Typically, most of the time and energy in such applications are spent on inner loops, or *kernels*. A kernel can often spawn a large number of threads to operate on different data segments. Exploiting such data-level parallelism is the key to high performance and energy efficiency. For example, general-purpose GPUs, like those offered by NVIDIA, contain multiprocessors designed to execute a large number of identical threads [21]. While GPUs tend to perform well in dense floating-point matrix computation, they are less ideal for some integer or fixed-point applications where overhead incurred to support general-purpose processing is relatively large. Hardware accelerators specialized for a particular kernel has the potential for superior performance and energy efficiency by exploiting parallelism and reducing overhead at every aspect. In this paper, we are interested in creating efficient architectures from kernel code automatically using high-level synthesis (HLS) [8, 10].

While it is relatively straightforward to map data-parallel threads onto multiple copies of an accelerator, and to reuse an accelerator for another thread when the previous thread has finished, a kernel-specific accelerator can be designed to support pipelined thread execution, i.e., it allows a thread to start before the previous one finishes. This pipelined architecture often leads to high performance and low overhead for a number of reasons. First, by using a specialized datapath topology, application-level data processing, storage and movement are directly supported in hardware, overhead in instruction fetch/decode is eliminated, and instruction-level parallelism can be fully exploited by adding functional units as needed. Second, multiple threads can execute in parallel on a single piece of hardware. Different from CPU/GPU in which threads share functional units in a time-multiplexed manner, the customized architecture can have sufficient amount of functional units to allow a large number of threads to execute in a truly concurrent fashion. Third, reusing the entire accelerator across threads is much easier than reusing functional units among instructions in the same thread, because instruction-level resource sharing often incurs additional wires, multiplexers and control logic, which in turn results in inferior frequency, area and power.

Unfortunately, conventional pipeline synthesis techniques effectively enforce a total order of the threads. In other words, threads cannot be reordered once they start execution in the pipeline, even when there are no dependencies between them. In case a thread is waiting for a variable-latency operation (such as reading from an external memory) to complete, the thread needs to be stalled, and thus all subsequent threads are blocked until the operation completes. In some cases, it is possible to avoid the problem of external memory access by prefetching remote data into a local scratchpad memory before starting the thread. However, this is not always possible, because sometimes memory addresses can only be determined at runtime after some operations in the thread are executed. A classic example is accessing a sparse matrix in compressed sparse row format as in the sparse matrix vector multiplication (SpMV) example in Figure 1, where the addresses for accessing the sparse matrix are decided by the content of an index array. Therefore, it is important for the pipelined architecture to tolerate unpredictable operation latencies effectively.



(a) C code (b) Data flow graph

Figure 1. Sparse matrix vector multiplication (SpMV) kernel.

In this work, we investigate the problem of synthesizing efficient application-specific pipelines for data-parallel kernels, with a primary goal of obtaining high throughput in the presence of unpredictable operation latencies. Our major contributions are threefold:

1. We systematically study the problem of pipeline synthesis for data-parallel kernels that contain variable-latency operations, and motivate the needs for out-of-order thread execution.
2. We propose a novel context-switching-enabled pipelining approach to allow efficient out-of-order execution of data-parallel threads. We show that the proposed method achieves significant improvement in throughput compared to conventional pipelining techniques.
3. We formulate a scheduling problem for minimizing the context switching cost of the multithreaded pipeline; we then devise an exact formulation as well as an efficient heuristic algorithm

to solve the optimization problem. Experimental results show that our techniques can dramatically reduce the total context bitwidth, leading to substantial savings of on-chip memories.

The rest of the paper is structured as follows: Section 2 provides the preliminaries for pipelining with variable-latency operations; Section 3 describes our approach to multithreaded pipelining that allows out-of-order thread execution through context switching; Section 4 describes our optimization algorithms for reducing the context bitwidth; Section 5 reports experimental results; Section 6 reviews the previous work, followed by conclusions in Section 7.

2. Preliminaries

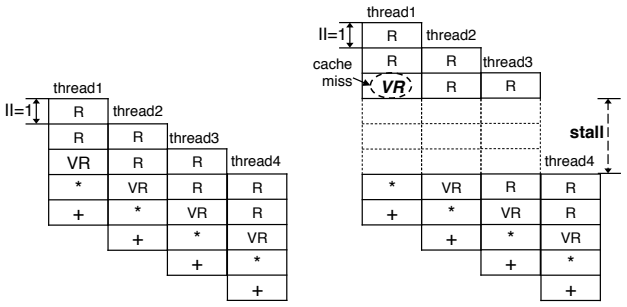
We define a *thread* as an instance of a kernel, which is a self-contained sequence of operations often taking the form of a function or loop body. Modern CPUs and GPUs maintain more execution contexts on chip than they can simultaneously execute to achieve hardware multithreading for tolerating memory access latency [20]. Our proposed approach for context-switching-enabled multithreading is partly inspired by these architectures; but we seek greater specialization by synthesizing custom pipelines that allow multiple threads to run in parallel on different pipeline stages. In contemporary HLS tools, modulo scheduling [22] is a widely employed technique for generating the custom pipelined architecture [5, 12, 26]. A modulo scheduling algorithm constructs a static schedule for a single iteration/thread so that the same schedule can be repeated at a constant interval, termed *initiation interval (II)*, without violating any dependence and resource constraints. The initiation interval imposes an upper bound on the throughput of the pipeline. The latency of a single iteration/thread is called the depth of the pipeline.

When threads execute on an accelerator, the accelerator often needs to access data from an external shared memory. Such an operation typically incurs a difficult-to-predict latency due to (1) access to the memory system hierarchy, (2) non-deterministic characteristics of certain memory types (e.g., DRAMs have data-dependent refreshing), and (3) contentions on the system-level interconnects and other shared on-chip resources. We further note that modern system-on-chips (SoCs) typically employ a packet-switched on-chip network, which improves the system scalability but also introduces additional uncertainties to the data access latencies. For conventional pipelining approaches that assume the threads are executed in-order and the schedule is carried out in lockstep, it is inevitable to stall the entire pipeline to account for the unpredictable completion time of a variable-latency operation, such as an external memory access. Depending on the technique used to tolerate the variable latency, the pipeline may experience different degree of stalling.

2.1 Pipeline Stalling

A basic approach to deal with variable-latency operations is to optimistically schedule each operation based on its minimum latency to obtain a compact schedule. If an operation does not complete within the minimum latency, the current thread and all its subsequent threads would be stalled by this blocking operation. This approach is commonly used in modern HLS tools; but unfortunately it often incurs excessive pipeline stalls and degrades the overall performance.

Figure 2 demonstrates the baseline approach with the SpMV kernel. Based on the data flow graph in Figure 1(b), we can pipeline the threads with $II = 1$ as shown in Figure 2(a). Under an ideal condition where all data can be obtained from a local cache or scratchpad memory with a fixed latency, the threads are executed in exact accordance to the schedule without any stalls. However, as illustrated in Figure 2(b), when a cache miss occurs due to a variable-latency read access (VR) to the external memory, the current thread has to be stalled to wait for the memory data. Moreover, all subsequent threads must be stalled as well to ensure in-order thread execution, resulting in in extra latency and decreased throughput.



(a) Normal pipeline execution (b) Pipeline stall

Figure 2. Pipeline stall due to the external memory access.

2.2 Deep Pipelining

We can enhance the baseline pipeline stalling approach by reserving a few additional pipeline stages for each variable-latency operation instead of simply using the minimum latency. With such *deep pipelining* approach, the extra pipeline stages allow the pipeline to proceed for a few extra cycles even while an operation is experiencing an unexpected delay.

Figure 3 illustrates the deep pipelining approach with the SpMV kernel. Considering that the operation `VR` may access the external memory with a variable latency, the scheduler reserves two additional pipeline stages `VR-1` and `VR-2` to tolerate part of the unexpected latency for `VR`. Therefore, when `VR` does not complete within the minimum latency (assumed to be one cycle in this example), the pipeline can still proceed for two more clock cycles, and thus allow two additional threads to enter the pipeline. However, if the actual access latency is longer than the allotted pipeline stages, the pipeline must be stalled until the requested data arrive. In this example, if `VR` has an actual latency of four cycles, the pipeline would only proceed two cycles and then stall for one cycle. Obviously, allocating even more pipeline stages for variable-latency operations can potentially better tolerate the memory latency and reduce pipeline stalls, although a deeper pipeline would come with additional hardware overhead in area, power, and timing. In addition, the deep pipelining approach still enforces in-order thread execution, which would result in a sub-optimal pipeline throughput in many cases, as discussed in Section 3.3.

3. Multithreaded Pipelining

In this section, we present a novel multithreaded pipelining technique that allows out-of-order thread execution to improve performance for data-parallel kernels. We note that a conventional in-order pipelining scheme also supports multithreading since it executes multiple threads concurrently on different stages of the pipeline. However, when a thread is blocked due to unexpected latency, the entire pipeline has to stall to enforce the thread ordering even if no data dependences exist between threads. In contrast, our proposed out-of-order multithreaded pipelining scheme allows the subsequent threads to proceed, thus leading to a higher throughput. More specifically, the suspended thread will release its occupied resources by saving all live values at the time of the suspension, so that the subsequent threads can march ahead without unnecessary stalling. Later the suspended thread will be swapped back to the pipeline once the variable-latency operation is completed.

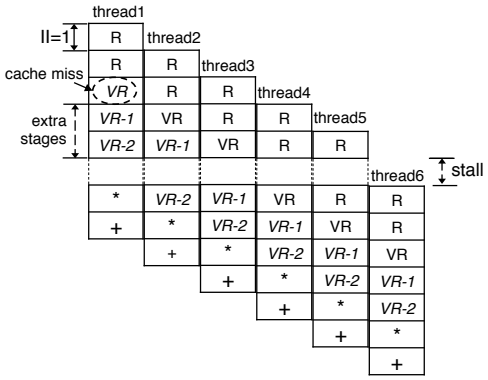


Figure 3. Deep pipelining approach.

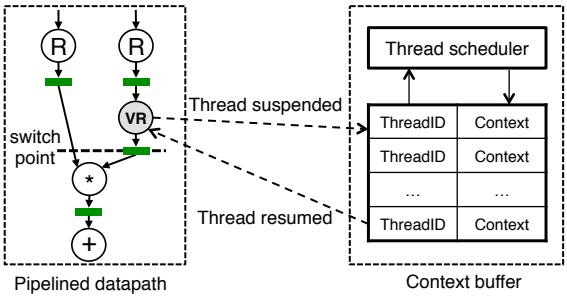
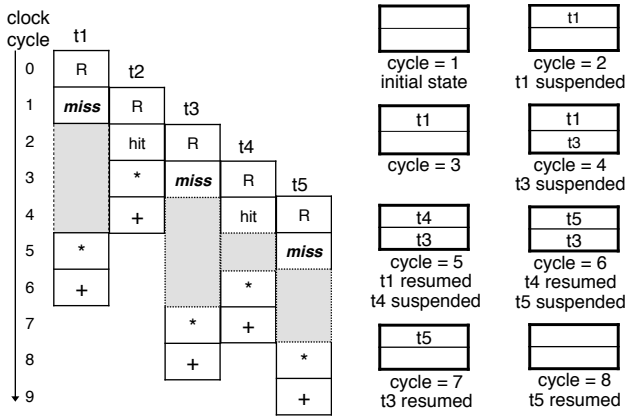


Figure 4. Basic mechanisms of a multithreaded pipeline: The pipelined datapath is extended with context switching support; Each context buffer saves the thread context when a thread is suspended; The thread scheduler decides which thread will be resumed if there are multiple ready threads in the context buffer.

3.1 Context Switching

In this paper, we use the term *context switching* to denote the pipeline behaviors involved for suspending and/or resuming a thread. Here the *context* denotes the collection of all live values at a specific cycle when the context switching occurs, which we term as the *switch-point*. To achieve context switching for out-of-order thread execution, a *context buffer* is designated to store all the contexts of the suspended threads.

Figure 4 illustrates the basic mechanisms of a multithreaded pipeline with the context switching support. Compared to the conventional pipelining schemes, we mainly extend the pipelined datapath with a context buffer to support out-of-order thread execution. The context buffer is a dedicated hardware storage equipped with a lightweight thread scheduler. Each entry of the buffer contains the context of a suspended thread and its thread ID. When a thread is switched out due to a blocking operation with unexpected latency, the context of the thread is saved into the context buffer. Afterwards, when the blocking operation is completed, the context of the thread is restored into the pipeline. Note that at the switch-point where a thread needs to be suspended, the pipeline examines the context buffer to determine if there exist any threads that are previously switched out at the same switch-point and are ready to continue execution. If so, one of the ready threads will be selected and its context will be restored into the pipeline before it resumes execution. In case there is no ready thread and no free entry in the context buffer, the pipeline will stall until at least one thread becomes ready.



(a) Pipelined execution (b) Context buffer snapshots

Figure 5. Multithreaded pipelined execution of the SpMV kernel: (a) Pipelined execution of five threads (`t1-t5`) within ten cycles; (b) Snapshots of the context buffer for cycles 1-8.

Figure 5 demonstrates the execution of the SpMV kernel in the multithreaded pipeline, where Figure 5(a) shows the execution of five threads (`t1-t5`) within ten clock cycles and Figure 5(b) sketches how the context buffer is updated each cycle. Initially, there is no thread executing in the datapath before cycle 1, and the context buffer is empty. The first thread `t1` starts executing in the first cycle. In cycle 2, `t1` incurs a cache miss and results in a context switching; thus the context of `t1` is saved into the context buffer in the next cycle. In contrast to the conventional pipelining approach, our multithreaded pipelining architecture allows the pipeline to proceed even though `t1` is blocked. In cycle 4, `t3` is suspended due to another cache miss, and the context of `t3` is saved into the context buffer. The context buffer now contains `t1` and `t3`. In cycle 5, `t1` has completed its memory access and becomes ready. Hence it is resumed and its context is restored into the pipeline. At the same time, `t4` gets suspended and joins `t3` in the context buffer; but since `t4` is swapped out to make room for `t1`, it is flagged as ready immediately. In cycle 6, `t5` incurs a cache miss and triggers the context switching; thus `t5` is suspended and `t4` is swapped back into the pipeline. In cycle 7, `t3` becomes ready and its context is restored, leaving only `t5` in the context buffer. Eventually, `t5` will be resumed after it becomes ready.

As evident in Figure 5, our multithreaded pipeline allows threads to execute out of order. For instance, while thread `t2` enters the pipeline later than `t1`, it actually completes sooner since `t1` is suspended for three cycles. Thread `t4` also finishes ahead of `t3` in the same example. In Section 3.3, we will provide quantitative analysis to justify that this added flexibility in thread scheduling leads to a much higher pipeline throughput for data-parallel kernels.

3.2 Context Buffer Microarchitecture

Context buffer plays a critical role in supporting out-of-order execution for multithreaded pipelining. As opposed to the fixed-size context register files commonly used in multithreaded CPU/GPU architectures, our context buffer is application specific. Nevertheless, our synthesis engine instantiates context buffers with different capacities based on a common microarchitecture template, which is illustrated in Figure 6. The microachitecture template comprises of four major components: `ContextMem` is the memory block that stores the context data of the suspended threads, `ReadyReg` and `FreeReg` indicate the status of each entry in `ContextMem`, and `MemReqBuf` keeps track of the pending memory requests. The bitwidth and depth (i.e., number of entries) of these mem-

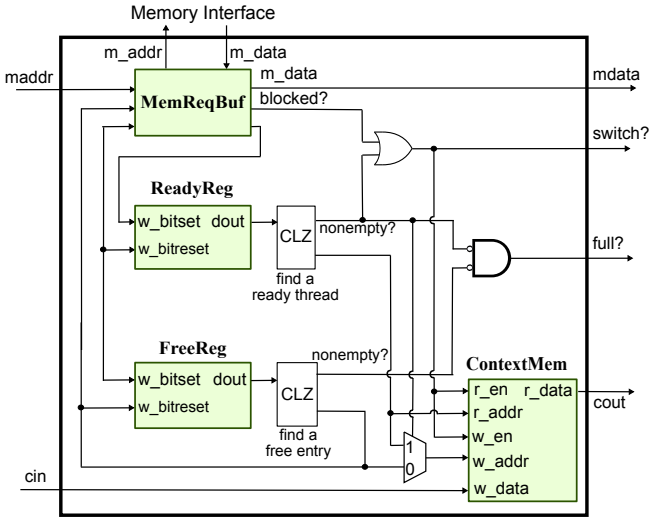


Figure 6. Microarchitecture template for the context buffer.

ory/register units are determined at compile time based on the application characteristics, the throughput requirement, as well as the pipeline synthesis algorithms that will be discussed in Section 4.

In each cycle, the context buffer accepts memory requests from the pipelined datapath and check if the requested memory data are immediately available from the local cache. If not, the context buffer forwards the memory request to the shared memory controllers and initiates the context switching process. At the first step of the context switching, `FreeReg` and `ReadyReg` in the context buffer are scanned to locate a free entry where the new context can be stored. If there is not a free entry and a ready thread, the pipeline is stalled. If there are multiple ready threads, the thread scheduler determines which thread will be resumed. To implement a very lightweight thread scheduler, we use a count leading zero (CLZ) component to realize a basic round-robin scheduling policy. The microarchitecture can be extended to support other thread scheduling policies.

Since the threads are executed out of order, it is important to avoid starvation. To this end, we currently enforce the pipeline to switch context at the earliest switch-point to restore the ready thread with the highest priority. If the thread being swapped out is not blocked by any variable-latency operations, the context buffer will immediately mark it as ready. An alternative, more energy-efficient method is to intentionally inject a bubble to the pipeline for delaying the processing of the incoming thread so that the ready thread in the context buffer can be swapped back when the bubble arrives at the anticipated switch-point.

3.3 Throughput Analysis

In this section, we compare the throughput between context switching and deep pipelining. Considering a pipeline with a single variable-latency memory access, the intuition is that with the same amount of additional storage capable of holding N blocking threads (i.e., N extra pipeline stages or N context buffer entries), the context switching approach always achieves a throughput (in number of threads processed per cycle) at least as high as that by deep pipelining. This can be shown by the fact that the *serial* buffer in the deep pipelining architecture can be emulated by the *parallel* context buffer with a modification to the thread scheduler. Instead of checking all threads in the context buffer to find a ready thread to resume, the scheduler could only check if the thread with smallest ID (oldest thread) and wait if it is not ready. In this way, the in-order thread execution is guaranteed, and the context switching architecture behaves exactly as the deep pipeline. With a more aggressive scheduling policy, a ready thread can execute even if it is not the oldest, and this clearly leads to better performance.

It is possible to model the system using queuing theory and analyze the throughput under certain distributions [19]. For example, if the cache miss penalty is constant, and different accesses have independent miss rate, the context buffer is equivalent to the Geo/D/N queue, i.e., a queue with Bernoulli arrivals, deterministic service time, and N servers. However, simple analytical solutions are usually available only when the distribution is memoryless.

In this work, we experimentally measure the throughput with different parameters using Monte Carlo simulation. Figure 7 shows the throughput comparison with different parameters R and N, where R is the number of variable-latency operations, and N is the total extra storage space (not used in the baseline approach) that is evenly allocated to all variable-latency operations. For simplicity, we assume each variable-latency operation corresponds to a unique switch-point. The variable-latency operation can be a cache hit or miss with a cache miss rate of 5%. We assume the cache hit latency is one cycle, while cache miss latency is a random variable following binomial distribution with a maximum value of 100, a mean value of 90 and a variance of 9. As shown in Figure 7, a larger N can achieve a higher throughput in both deep pipelining and context switching approaches by tolerating more cache misses, and the context switching approach can quickly approach the optimal throughput with a smaller N. In addition, as the number of variable-latency operations increases, the deep pipelining approach would have a lower throughput due to more frequent pipeline stalls, while our context switching approach still maintains a high throughput by better tolerating the difficult-to-predict memory latencies with out-of-order thread execution.

4. Optimization of Context Switching Cost

While context switching effectively leads to a higher throughput in presence of unpredictable operation latency, it also potentially incurs nontrivial hardware overhead, because the contexts of the pending threads must be saved. Thus, minimizing the size of the context becomes crucial to the overall hardware resource usage.

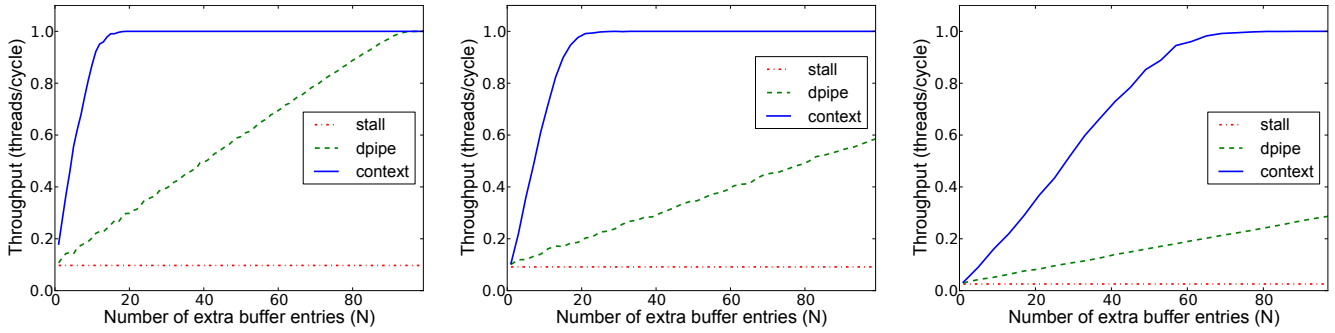
We use *context bitwidth* to denote the total bitwidth of all live values when a thread is suspended. Figure 8 shows how different schedules affect the context bitwidth for the SpMV kernel. Since `VR` is a variable-latency operation, a switch-point is needed between `VR` and its successors. With a suboptimal schedule (e.g. ASAP or ALAP schedule) shown in Figure 8(a), the context contains a 32-bit value, resulting in a context bitwidth of 32. In contrast, an optimal schedule can achieve a context bitwidth of 8 as shown in Figure 8(b). Clearly, a good schedule can significantly reduce the context bitwidth and thus the overall resource usage. In this section, we present both the exact mathematic formulation and an efficient heuristic algorithm to solve the context minimization problem.

4.1 Exact Formulation

We first formulate the scheduling problem for minimizing the context switching cost as an integer linear program (ILP). The problem we seek to solve is formally stated as follows:

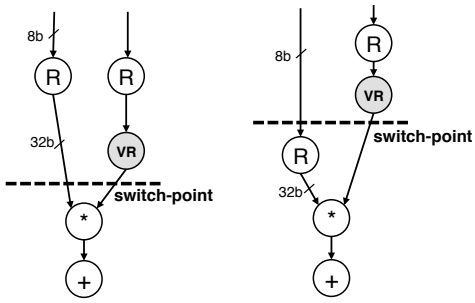
Given: (1) A data-parallel kernel represented by a control data flow graph; (2) A set of variable-latency operations O; (3) A set of scheduling constraints C, such as dependence constraints, timing constraints, resource constraints, etc.

Goal: Find a pipelined schedule and a set of switch-points for operations in O without violating any given constraint in C. Let U_t denote the context bitwidth for the switch-point at time step t, the



(a) $R = 1$ (b) $R = 2$ (c) $R = 4$

Figure 7. Throughput study for different parameters: N = total number of extra pipeline stages or context buffer entries for all variable-latency operations; R = number of variable-latency operations. `stall` = baseline pipelining approach; `dpipe` = deep pipelining approach; `context` = context switching approach.



(a) ASAP schedule (b) Optimal schedule

Figure 8. Context bitwidth for different schedules: (a) An as-soon-as-possible (ASAP) schedule resulting in a 32-bit context; (b) An optimal schedule achieving an 8-bit context.

objective is to minimize the total context bitwidth.

$$\textit{Objective:} \quad \text{minimize} \sum_{t} U_t \tag{1}$$

Let the binary variable x_{ij} denote whether operation i is scheduled at time step j. We use Equation (2) to restrict that each operation be scheduled at exactly one time step.

$$\sum_{j=1}^{L} x_{ij} = 1, \quad \forall i \tag{2}$$

We define binary variable w_{kj} to indicate if a switch-point for operation k occurs at time step j. With Equation (3), we specify that each variable-latency operation has exactly one switch-point. We also note that multiple variable-latency operations can share the same switch-point.

$$\sum_{j=1}^{L} w_{kj} = 1, \quad \forall k \in O \tag{3}$$

In the following, we will make use of x_{ij}'s and w_{kj}'s to capture various other scheduling constraints, including the dependence constraints, resource constraints, and context switching constraints.

Dependence constraints A dependence constraint between operations u and v, where u needs to finish before v starts, is captured by the following equation:

$$\sum_{j=1}^{L} j \cdot x_{uj} + l_u - \sum_{j=1}^{L} j \cdot x_{vj} \leq 0, \ \forall (u,v) \in E \tag{4}$$

Here E denotes the set of dependencies in the kernel and l_u denotes the latency of operation u.

Resource constraints Let $type(i)$ be the resource type of operation i, Res_r be the upper bound of the number of resources of type r, and II be the initiation interval. The resource constraints can be formulated as follows:

$$\sum_{s=0}^{\lfloor (L-p)/II \rfloor} \sum_{type(i)=r} x_{ij_{(p,s)}} \leq Res_r, \tag{5}$$
$$\text{where } j_{(p,s)} = p + II \times s, \quad \forall r, \ \forall p : 1 \leq p \leq II$$

Essentially, the constraint sums up the resource usage of all overlapping threads and ensure that the resource constraints are satisfied based on the current modulo schedule.

Context switching constraints To ensure a thread can be safely suspended when an operation u incurs an unpredictable latency, the context switch needs to happen before executing any operations that depend on the result of u. In fact, when u is blocked, we do not necessarily need to stall the thread right away; instead, we can allow it to continue until one of u's successors is encountered. In other words, the switch-point should be scheduled after operation u and before any successors of u. These constraints are captured by Equations (6) and (7) as follows.

$$\sum_{j=1}^{L} j \cdot x_{uj} + l_u - \sum_{j=1}^{L} j \cdot w_{uj} \leq 0, \ \ \forall u \in O \tag{6}$$

$$\sum_{j=1}^{L} j \cdot w_{uj} - \sum_{j=1}^{L} j \cdot x_{vj} \leq -1, \ \ \forall u \in O \text{ and } (u,v) \in E \tag{7}$$

In order to compute the context bitwidth for each switch-point, we use binary variable P_j to indicate whether time step j is a switch-point. $P_j = 1$ if there is at least one variable-latency operation k that has $w_{kj} = 1$; otherwise $P_j = 0$. We use Equations (8) and (9) to ensure this constraint:

$$w_{kj} - P_j \leq 0, \quad \forall k \in O, \forall j \tag{8}$$

$$P_j - \sum_{k} w_{kj} \leq 0, \quad \forall j \tag{9}$$

Let W_u denote the bitwidth of operation u, we have constraint (10) for the context bitwidth of each time step t ($0 \leq t \leq L$).

$$B_{it} = \sum_{j \leq t} x_{ij}, A_{it} = \sum_{j > t} x_{ij}$$
$$\sum_{(u,v) \in E} W_u \left(B_{ut} - B_{vt} + A_{vt} - A_{ut}\right) - H \cdot (1 - P_t) \leq 2 \cdot U_t \tag{10}$$

Here B_{it} denotes whether operation i is scheduled at or before time step t, A_{it} denotes whether operation i is scheduled after time step t, and H is a large constant that is used to relax this constraint when time step t is not a switch-point (i.e., $P_t = 0$). Equation (10) bounds the total bitwidths on all edges that span across the time step (i.e., live values) if it is a switch-point.

4.2 Heuristic Algorithm

As ILP is in general not scalable, we further design a heuristic scheduling algorithm to minimize the context bitwidth. The key idea is to coordinate scheduling and context switching based on the graph min-cut theory [9]. Intuitively, given the control data flow graph, each context contains a set of nodes that divide the graph into two disjoint subgraphs: one containing operations scheduled before the switch-point, and the other containing all succeeding operations. Thus, the operations in a context actually form a node cut in the data flow graph. Finding a min-cost context (i.e., with minimum total bitwidth) for a variable-latency operation can be reduced to the problem of finding a min-cut for the control data flow graph with some additional constraints for context switching.

However, it is intrinsically hard to obtain an optimal schedule when we need to account for a variety of scheduling constraints. It is even more difficult to further minimize the context bitwidths for multiple variable-latency operations. In this work, we propose a coordinated scheduling and context switching heuristic, which iteratively applies the min-cut algorithm to guide the scheduling process for optimizing the context switching cost associated with each variable-latency operation.

Algorithm 1: Heuristic scheduling algorithm for context switching.

```
LastSwitchPoint ← 0
while more variable-latency operations to schedule do
    pick a variable-latency operation o with max ASAP
    G' ← construct_auxiliary_graph(CDFG, o)
    C ← find_min_cut(G')
    NewSwitchPoint ← max ASAP for all nodes in C
    foreach n in C do
        schedule n as late as possible before
        NewSwitchPoint
    end
    insert a switch-point at NewSwitchPoint
    foreach predecessor node p of C do
        if (in_degree_cost(p) > out_degree_cost(p)) then
            schedule p as early as possible after
            LastSwitchPoint
        else
            schedule p as late as possible before
            NewSwitchPoint
        end
    end
    schedule o as late as possible
    LastSwitchPoint ← NewSwitchPoint
end
schedule remaining operations as close as possible to
previously scheduled nodes
```

Algorithm 1 shows the pseudo code for our heuristic. It iteratively schedules variable-latency operations in a topological order. For each variable-latency operation o, we first construct a auxiliary graph as follows. Given a directed control data flow graph $CDFG = (V, E)$ with a node set V and an edge set E, and a given node $o \in V$ with predecessors $Preds(o)$ and successors $Succs(o)$, the corresponding auxiliary graph $G' = (V', E')$ is defined as follows:

- $V' = V - Preds(o) - Succs(o) + \{s, t\}$
- $e = (u, v) \in E',\ \forall e : e \in E$ if $u \in V'$ and $v \in V'$
- $(s, x) \in E',\ \forall x : \exists (u, x) \in E$ and $u \in Preds(o)$
- $(y, t) \in E',\ \forall y : \exists (y, v) \in E$ and $v \in Succs(o)$

Based on the auxiliary graph, we apply a max-flow min-cut algorithm to find the min-cost node cut C. These nodes compose a context and we tend to schedule them together and as late as possible before the switch-point. A switch-point is inserted right after the node in C with the largest $ASAP$ schedule. After all nodes in the min-cut set are scheduled, their predecessors are scheduled according to their in_degree_cost (sum of all incoming edge costs) and out_degree_cost (sum of all outgoing edge costs). If in_degree_cost is greater than out_degree_cost, then the node would be scheduled as early as possible to minimize pipeline registers for its incoming edges; otherwise, it is scheduled as late as possible to minimize pipeline registers for its outgoing edges. Scheduling constraints are checked and updated each time a node is scheduled. It is important to note that our algorithm always attempts to schedule operations between $LastSwitchPoint$ and $NewSwitchPoint$ to avoid that a value is unnecessarily live across multiple switch-points.

After all variable-latency operations are scheduled and all required switch-points are inserted, the remaining operations are scheduled as close as possible to previously scheduled ones in order to minimize register usage, in a way similar to the swing modulo scheduling [16, 17].

5. Experimental Results

The proposed context management scheme and pipeline scheduling algorithms are implemented within the framework of a commercial HLS tool, which is based on LLVM compiler infrastructure [14]. We implement our scheduling algorithms as a separate LLVM pass, and push the scheduling results through the default RTL code generator backend. We use CPLEX [3] to solve the ILP problem for minimizing the context cost. The generated RTL code is synthesized by Xilinx ISE 14.7 targeting the Virtex-7 FPGA device.

Experiments are performed on a number of data-parallel kernels extracted from a variety of irregular applications. `spmv` is a classic sparse matrix vector multiplication kernel with matrix stored in the compressed sparse row format. `bfs` is a parallel breath-first search kernel extracted from the Graph 500 benchmark suite [2]. Other designs are extracted from a widely used sparse matrix package SuiteSparse [1]: `decompress` translates a sparse matrix into a regular matrix; `tranpose` transposes a sparse matrix while still keeping the same format; `utsolve` solves a sparse upper-triangular matrix system; `mmadd` performs a specialized matrix addition between two sparse matrices; `maxtrans` finds an augmenting path starting at a given column and extends the match if found. All sparse matrices are stored in the compressed sparse row format.

5.1 Throughput Comparison

Table 1 compares the throughput results between different approaches: `stall` is the baseline pipelining approach, `dpipe` is the deep pipelining approach, and `context` is the proposed context-switching-enabled pipelining approach. Each evaluated design contains a number of indirect array reads and writes, which may access either a local memory (cache hit) in two cycles or an external memory (cache miss with a 5% miss rate) with a variable latency. For each external memory access, we insert N extra pipeline stages for

Table 2. QoR comparison for different approaches: `CP` = achieved clock period in *ns* with an 8ns target; `LAT` = pipeline latency/depth in clock cycles; `SLICE` = # of slices; `LUT` = # of lookup tables; `FF` = # of flip-flops; `BRAM` = # of block RAMs.

Design	Approach	CP	LAT	SLICE	LUT	FF	BRAM
spmv	stall	4.43	4	120	191	280	3
	dpipe	4.67	60	634	803	1320	3
	context	4.67	10	189	233	546	6
bfs	stall	5.76	5	141	264	269	3
	dpipe	5.68	82	1366	1339	2593	4
	context	4.67	9	226	281	589	8
decompress	stall	4.43	3	265	455	764	1
	dpipe	6.57	88	1798	2212	2998	2
	context	4.67	13	337	457	1169	6
transpose	stall	4.43	4	188	269	444	3
	dpipe	6.33	83	1491	1752	3059	3
	context	6.04	8	255	298	756	8
utsolve	stall	4.67	7	197	308	343	7
	dpipe	5.05	87	1428	1504	3043	7
	context	4.67	10	245	328	514	10
mmadd	stall	4.43	5	419	738	1089	4
	dpipe	6.67	90	3140	3022	5951	3
	context	5.47	11	475	645	1404	13
maxtrans	stall	4.67	6	226	406	594	5
	dpipe	6.07	104	2445	2216	4148	7
	context	4.67	13	332	464	887	12

the deep pipelining approach and employ an N-entry context buffer for the context switching approach. Based on a target clock period of 8ns, we measure the number of threads that can be processed per second for each approach. With the number of extra buffer entries varying from 4, 8, 16, and 32, the deep pipelining approach can achieve an average speedup of 1.5x, 2.5x, 3.9x, and 4.7x respectively over the baseline approach, while the context switching approach can achieve an average speedup of 7.2x, 14.1x, 17.5x, and 19.3x respectively. Unlike the deep pipelining approach, which is still limited by the thread ordering, our context switching approach can significantly improve the overall performance and achieve a near-optimal throughput by allowing out-of-order thread execution.

5.2 Timing and Resource Usage Comparison

Table 2 lists the timing and resource usage of the synthesized designs under three different pipelining approaches. In this experiment, all designs are meeting the 8ns clock period (CP) constraints. Not surprisingly, the baseline approach has the shortest latency and demands the least resource usage, but at the expense of low throughput; the deep pipelining approach incurs a much longer latency and results in more lookup tables (LUTs, especially in the form of shift register lookup tables) and flip-flops (FFs) due to the additional pipeline stages; the context switching approach requires more block RAMs (BRAM) to store the thread contexts.

We note that the scheduling optimization algorithms described in Section 4 are able to significantly reduce the size of the context buffers. Figure 9 compares the size of the contexts (in total number of bits) under different scheduling strategies. On average, the heuristic and exact approach can reduce the context size by 84% and 88%, respectively. Although the ILP formulation can achieve optimal resource reduction, it is not as scalable as the heuristic algorithm for larger designs such as `mmadd` and `maxtrans`.

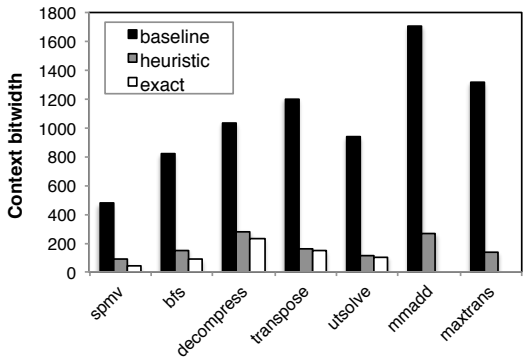


Figure 9. Context bitwidth comparison with different scheduling algorithms: baseline, heuristic, and exact. With the exact approach, the ILP solver times out in `mmadd` and `maxtrans`.

6. Related Work

Pipelining is an important optimization in HLS for improving design throughput because it allows multiple loop iterations or function invocations to overlap and execute in parallel on different stages of a pipelined datapath. Many HLS systems, such as LegUp [6], PICO-NPA [24], and Vivado HLS [8], have enabled pipelining based on modulo scheduling [23]. Additional optimizations, such as memory port reduction [4], register pressure minimization [26], pipeline flushing [12], and polyhedral analysis [18, 27], continue to extend the state of the art of pipelining techniques. A pipelined architecture naturally enables efficient hardware multithreading for data-parallel applications. For instance, Altera's OpenCL compiler [11] constructs kernel-specific pipelines to implement high-performance hardware on FPGAs for applications described in OpenCL [25]. However, it appears that the synthesized pipeline enforces an in-order execution between threads, which is potentially less effective in tolerating the memory latency with an irregular address stream.

There has been a growing interest in optimizing HLS techniques for irregular applications with non-deterministic workload and unpredictable memory access latency. For example, Halstead and Najjar [13] proposed the CHAT compiler, which can accelerate sparse matrix multiplication (SPMV) through a multithreaded datapath on FPGAs. While CHAT uses deep FIFOs to realize multithreading for SPMV, our approach explores context switching as a more general approach for enabling out-of-order thread execution to achieve even higher speed-up for a wider range of applications. Liu et al. [15] recently proposed coarse-grained pipeline accelerators (CGPAs) to exploit coarse-grained parallelism by decomposing irregular loops into serial/parallel stages, where the serial stages perform the difficult-to-parallelize data structure manipulations and the replicated datapaths are used to speed up the main computation in the loop. In [7], Choi et al. proposed to generate multithreaded parallel hardware architectures using Pthreads and OpenMP, and map software threads to multiple copies of the same accelerator instead of using the pipelined architectures. We currently focus on exploiting the data-level parallelism that allows multiple threads to execute on the same pipelined datapath. Nevertheless, our approach can also be extended to incorporate coarse-grained parallelization techniques to provide additional throughput improvement, which will be investigated as part of our future work.

7. Conclusions

In this paper, we study the problem of multithreaded pipeline synthesis for data-parallel kernels and propose a context-switching-

Table 1. Throughput comparison for different approaches: N is the number of extra pipeline stages or context buffer entries (not used in `stall`) for each variable-latency operation; factors in parenthesis are the speedup over the baseline approach (`stall`).

		N=4		N=8		N=16		N=32	
Design	`stall`	`dpipe`	`context`	`dpipe`	`context`	`dpipe`	`context`	`dpipe`	`context`
spmv	0.9	1.3 (1x)	5.3 (6x)	1.9 (2x)	9.8 (11x)	2.9 (3x)	11.6 (13x)	4.9 (6x)	12.5 (14x)
bfs	0.8	1.1 (1x)	4.9 (6x)	1.6 (2x)	9.3 (12x)	2.7 (3x)	11.1 (14x)	4.8 (6x)	12.5 (16x)
decompress	0.6	1.1 (1x)	4.8 (8x)	1.6 (3x)	9.3 (15x)	2.6 (4x)	11.3 (18x)	4.7 (8x)	12.5 (20x)
transpose	0.6	1.1 (2x)	4.8 (7x)	1.7 (3x)	9.3 (14x)	2.6 (4x)	11.4 (18x)	4.7 (7x)	12.5 (20x)
utsolve	0.7	1.1 (2x)	4.8 (7x)	1.6 (2x)	9.2 (14x)	2.6 (4x)	11.3 (17x)	4.7 (7x)	12.5 (18x)
mmadd	0.5	1.0 (2x)	4.5 (8x)	1.5 (3x)	8.7 (16x)	2.6 (5x)	11.2 (21x)	4.6 (9x)	12.5 (23x)
maxtrans	0.5	1.0 (2x)	4.3 (9x)	1.4 (3x)	8.5 (18x)	2.5 (5x)	11.1 (24x)	4.6 (10x)	12.5 (27x)

enabled pipelining approach which allows out-of-order thread execution. We further devise an exact formulation and a heuristic scheduling algorithm to minimize the hardware resources for supporting context switching. Experimental results demonstrate that our approach can achieve much higher throughput compared to conventional pipeline synthesis approaches. In addition, our exact and heuristic scheduling algorithms can significantly reduce the amount of storage needed for context management.

Acknowledgments

This work was supported in part by NSF Award CCF-1337240 and a research gift from Xilinx, Inc.

References

[1] SuiteSparse: A Suite of Sparse Matrix Packages. `https://www.cise.ufl.edu/research/sparse/SuiteSparse/`.

[2] The Green Graph 500. `http://www.graph500.org`.

[3] CPLEX: High-Performance Software for Mathematical Programming and Optimization, 2005.

[4] Y. Ben-Asher, D. Meisler, and N. Rotem. Reducing Memory Constraints in Modulo Scheduling Synthesis for FPGAs. *ACM Trans. on Reconfigurable Technology and Systems*, 3(3):1–19, 2010.

[5] A. Canis, J. H. Anderson, and S. D. Brown. Modulo SDC Scheduling with Recurrence Minimization in High-Level Synthesis. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2014.

[6] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 33–36, Mar 2011.

[7] J. Choi, S. Brown, and J. Anderson. From Software Threads to Parallel Hardware in High-Level Synthesis for FPGAs. *Int'l Conf. on Field Programmable Technology (FPT)*, pages 270–277, Dec 2013.

[8] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: from Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 30(4):473–491, 2011.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to Algorithms*, volume 2. MIT press Cambridge, 2001.

[10] P. Coussy and A. Morawiec. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer, 2008.

[11] T. S. Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, J. Freeman, D. P. Singh, and S. D. Brown. OpenCL for FPGAs: Prototyping a Compiler. *Int'l Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 3–12, Jul 2012.

[12] S. Dai, M. Tan, K. Hao, and Z. Zhang. Flushing-Enabled Loop Pipelining for High-Level Synthesis. *Design Automation Conf. (DAC)*, Jun 2014.

[13] R. J. Halstead and W. Najjar. Compiled Multithreaded Data Paths on FPGAs for Dynamic Workloads. *Intl'l Conf. on Compilers, Architectures and Synthesis of Embedded Systems (CASES)*, Oct 2013.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. *Int'l Symp. on Code Generation and Optimization (CGO)*, pages 75–86, Mar 2004.

[15] F. Liu, S. Ghosh, N. P. Johnson, and D. I. August. CGPA: Coarse-Grained Pipelined Accelerators. *Design Automation Conf. (DAC)*, Jun 2014.

[16] J. Llosa, E. Ayguadé, A. Gonzalez, M. Valero, and J. Eckhardt. Lifetime-Sensitive Modulo Scheduling in a Production Environment. *IEEE Trans. on Computers (TC)*, 50(3):234–249, Mar 2001.

[17] J. Llosa, A. González, E. Ayguadé, and M. Valero. Swing Module Scheduling: A Lifetime-Sensitive Approach. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 80–86, Oct 1996.

[18] A. Morvan, S. Derrien, and P. Quinton. Polyhedral Bubble Insertion: A Method to Improve Nested Loop Pipelining for High-Level Synthesis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 32(3):339–352, 2013.

[19] R. Nelson. *Probability, Stochastic Processes, and Queueing Theory: The Mathematics of Computer Performance Modeling*. Springer, 1995.

[20] M. Nemirovsky and D. M. Tullsen. Multithreading Architecture. *Synthesis Lectures on Computer Architecture*, 8(1):1–109, 2013.

[21] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[22] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. *Int'l Symp. on Microarchitecture (MICRO)*, pages 63–74, Nov 1994.

[23] B. R. Rau and C. D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. *ACM SIGMICRO Newsletter*, 12(4):183–198, 1981.

[24] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-level Synthesis of Nonprogrammable Hardware Accelerators. *Journal of VLSI Signal Processing*, 31(2):127–142, 2002.

[25] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Design & Test*, 12(3):66–73, 2010.

[26] Z. Zhang and B. Liu. SDC-Based Modulo Scheduling for Pipeline Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 211–218, Nov 2013.

[27] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong. Improving High Level Synthesis Optimization Opportunity through Polyhedral Transformations. In *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 9–18. ACM, 2013.