Architectural Synthesis of Computational Pipelines with Decoupled Memory Access

Shaoyi Cheng and John Wawrzynek
Department of EECS, UC Berkeley, California, USA 94720
Email: sh_cheng@berkeley.edu, johnw@eecs.berkeley.edu

Abstract—As high level synthesis (HLS) moves towards mainstream adoption among FPGA designers, it has proven to be an effective method for rapid hardware generation. However, in the context of offloading compute intensive software kernels to FPGA accelerators, current HLS tools do not always take full advantage of the hardware platforms. In this paper, we present an automatic flow to refactor and restructure processor-centric software implementations, making them better suited for FPGA platforms. The methodology generates pipelines that decouple memory operations and data access from computation. The resulting pipelines have much better throughput due to their efficient use of the memory bandwidth and improved tolerance to data access latency. The methodology complements existing work in high-level synthesis, easing the creation of heterogeneous systems with high performance accelerators and general purpose processors. With this approach, for a set of non-regular algorithm kernels written in C, a performance improvement of 3.3 to 9.1x is observed over direct C-to-Hardware mapping using a state-of-the-art HLS tool.

Keywords—FPGA, Hardware Acceleration, High-level Synthesis, Memory-level Parallelism, Pipeline Parallelism, Memory Subsystem Optimization

I. INTRODUCTION

As the complexity of FPGA designs increases, there has been a trend towards design synthesis from higher levels of specifications. Being more compact and expressive, high level languages, when used as design input, can greatly increase the productivity of engineers. To tackle the challenge of generating hardware functional blocks from high level behavioral descriptions, many commercial [1], [2] and open source [3], [4] tools have been developed over the years. Programming languages such as C/C++, designed for processor-centric execution, are used by these high-level synthesis programs as the medium for input specification. Meanwhile, recent developments in FPGA SoCs, where the reconfigurable arrays are integrated with hard processors and memory interface IPs, have created highly versatile computing platforms [5]. This combination of new tools and devices has created new opportunities for applications written in high level languages. Applications can be mapped to these heterogeneous substrates, with the compute intensive loop nests running in accelerators and the remainder of the code executing on processors. However, the performance boost of the mapped implementations is often less than optimal when HLS tools are employed to directly map the software code to the reconfigurable fabric. Fundamentally, the barrier between software and the FPGA fabric is more than just the programming language used—the real difference lies in the paradigms of computation. To produce good FPGA designs with HLS, the users still need to visualize and create hardware descriptions, albeit with the C/C++ syntax. To effectively harness the power of reconfigurable platforms for software acceleration, in addition to inserting pragmas and directives, designers often need to restructure the original code to separate out memory accesses before invoking HLS. Also, to boost FPGA accelerator efficiency it is often desirable to convert from conventional memory accesses to a streaming model and to insert DMA engines [6]. Further enhancements can be achieved by including accelerator specific caching and burst accesses.

In this paper, we try to narrow the gap between software and hardware execution mechanisms by automatically transforming application kernels into pipelines of processing stages, complemented by load/store primitives capable of pipelined data accesses. Our flow slices the original control dataflow graph (CDFG) of the performance critical loop nests into subgraphs, connected with acyclic communication (section III). Special transformations are then performed on memory operations to allow pipelining of outstanding requests in the memory subsystem (section IV-A). Furthermore, the hardware structures connecting the accelerator and the memory are synthesized based on the observed data access patterns of the program (section IV-B). Finally, each of the subgraphs is fed to a conventional high-level synthesis flow, generating independent datapaths and controllers. FIFO channels are instantiated to connect the datapaths, forming the final system (section V).

When compared to hardware synthesized directly from the original program using HLS, the accelerators produced by our flow have superior performance (section VI). Their tolerance to data access latency is also demonstrated with a variety of memory subsystem configurations.

The main contributions of this paper are:

- a novel tool flow for converting software loop nests to pipelines of decoupled processing stages, where:
  - the effects of long latency operations are localized,
  - memory load/store operations are converted to data access modules which use memory bandwidth efficiently, and
  - customization of memory access mechanisms based on the data access patterns of the accelerated loop nests.
- an experimental evaluation of our approach against direct mappings using a state-of-the-art HLS tool, on FPGA SoCs with hard processors and memory interface IPs.
II. BACKGROUND AND RELATED WORK

A. Hardware Generation In Current High-level Synthesis Tools

High level synthesis attempts to capture parallelism in the control dataflow described by high level languages. Compute operations and memory accesses are scheduled and allocated according to the dependency constraints between them and the resource constraints of the target platform. Activation of a particular operation in the circuit is associated with a certain clock cycle and the execution of the entire control dataflow graph is orchestrated by a central controller synthesized alongside the datapath. As the scheduling is done statically, the runtime behavior of the accelerator is rather simple. Different parts of the generated circuit run in lockstep with each other, no dynamic dependency checking mechanisms such as scoreboard or load-store queueing, are needed. In terms of accessing data in memory, multiported cache [7] or network of caches [8] are sometimes generated complementing the datapath. The hardware accelerators in these systems are presented with a memory interface rather similar to that used by a processor. As the entire circuit is running on a rigid schedule, stalls introduced by cache misses propagate to the entire processing engine. Higher level parallelization can help the HLS tools produce separate processing lanes, isolating the effect of memory introduced stalls. This is especially useful when the input is expressed as multithreaded program [9].

In some other scenarios, inputs to HLS bear more resemblance to the final hardware than conventional high level algorithm descriptions. In [1] for instance, special primitives and pragmas are provided to enable the users to create streaming engines. They can also add in DMA and on-chip buffers for explicit management of data transfer from/to the main memory, simplifying the interaction between compute engine and data storage. It is apparent that high level synthesis can be used in a number of different ways, sometimes with more user effort and higher quality of results.

With respect to transforming sequential programs before FPGA mapping, there has also been plenty of work over the years [10], [11], though most of it focused on highly regular applications. In some of this work [12], [13], imperative descriptions in the original code is converted to a different model of computation (MoC) such as Kahn Process Networks, before being mapped to functional units in FPGAs. There are, however, unaddressed issues such as how to close the gap between the memory model of the new MoC and that of a general purpose processor, which is essential in the context of generating closely coupled CPU+accelerator systems.

B. Loop Parallelization

An essential aspect of modern high level synthesis flow lies in the exploitation of parallelism between loop iterations. Due to the presence of loop carried dependencies in many applications, the HLS tools use software pipelining [14] to initiate new iterations before previous ones are completed. The latency of the longest circular dependence in the control dataflow graph dictates the minimum interval with which a new iteration can be initiated, which ultimately bounds the overall throughput achievable by an accelerator. However, due to reasons mentioned in Section II-A, cache misses can cause the real performance of accelerators to fall far short of these theoretical bounds, especially for memory intensive applications.

Higher level transformations such as loop unrolling and loop splitting are sometimes used to complement the loop pipelining performed during HLS. Incurring higher area overhead, they improve the performance upper bound by either decreasing the total number of iterations or simplifying the logic in the critical cycle of dependencies. When and where these techniques should be applied are well researched and the flow introduced in this paper can be applied to loop nests already having undergone these transformations.

Another type of parallelism exploited by HLS are the parallelism between separate loop nests, sometimes called inter-block pipelining [15]. This technique mostly applies to regular loops where data dependence through memory can be accurately determined. One way to look at our methodology is that it extracts multiple blocks from one loop nest and then perform inter-block pipelining. The targets in our case, however, are non-regular loops.

C. FPGA Designs with Decoupled Memory Access

The idea of decoupled memory access in FPGA designs has been previously explored as well. In [16], streaming engines are interfaced with external memory through a decoupled architecture, where the address generation unit and the memory access scheduler can be configured to provide data for the datapath IOs. In [17], the complexity of managing memory hierarchy is hidden from the user with a standard abstraction of data access interfaces. The FPGA designers can use the provided primitives to explicitly manage data movement between the on-board memory and the SRAM on-chip, allowing locally-addressed memory access by the computation pipeline. A set of regular kernels are synthesized to this architecture in [18], demonstrating its applicability in the context of high-level synthesis.

III. FROM THE CDFG TO DECOUPLED SUBGRAPHS

Because standard HLS tools use a simplistic approach of statically scheduling operations at compile time, variable latency operations or operations that occur out of synchrony can result in stalls to the entire accelerator. Separating the control dataflow graph into decoupled subgraphs can contain the effect of these stalls, boosting the overall performance.

A. A Motivating Example

Shown in figure 1 is a simple example where separating a software kernel into multiple decoupled stages can improve the overall performance. There are two memory reads and a floating point multiply in the inner loop. When this for loop is pipelined with HLS, its initiation interval is dictated by the latency of the multiplier, assuming the best case memory access latency. However, since the computation kernel is turned into a monolithic accelerator, the centralized controller would have to stall the floating point multiplier when there is a long cache miss in data fetch. Consequently, the performance degrades due to the combination of the long latency of FP multiply and cache misses, resulting in a significant reduction of throughput.
Fig. 1. Conversion to Decoupled Stages

On the other hand, if the memory accesses and FP multiplication are decoupled, they can be running out of sync. The hardware queues used for communication between them can buffer data already fetched but not yet used. When the memory access parts are stalling for cache misses, the backend of data already produced so far can continue supplying the multiplier unit. Over a long period of time, the stalls introduced can be shadowed by the long latency of the FP multiplication, and consequently, the overall performance can be improved significantly.

B. Subgraph Formation Through Clustering

Given a performance critical loop nest, to generate the pipeline of decoupled processing modules, the instructions in the original CDFG are first clustered to form subgraphs. To maximize the performance of the resulted implementation, a few requirements must be addressed by our clustering algorithm. First, as mentioned in Section II-B, the circular dependencies in the innermost loop bound the overall throughput of the generated accelerators. It is therefore crucial that these cycles do not traverse multiple subgraphs as the FIFOs used for communication between modules always add latency. Secondly, as we have described in Section III-A, it is beneficial to have memory operations separated from dependency cycles involving long latency compute, so that cache misses can be shadowed by the slow rate of data consumption. Thirdly, to localize the effects of stalls introduced by cache misses, the number of memory operations in each subgraph should be minimized, especially when they address different parts of the memory space.

The first requirement was one of the factors addressed in [19], where sequential programs were converted into multithread codes running on multicore processors. Their algorithm finds strongly connected components (SCCs) in the original dataflow graph, collapses them into nodes and then heuristically partitions the resulted directed acyclic graph (DAG) into threads with balanced load. In our flow, the search for SCCs is also necessary and its outcome is used for the ensuing memory access centric clustering. Conceptually, a topological sort is performed on the DAG formed after the SCCs are collapsed into single nodes. In the linear array thus obtained, we find memory operations and SCCs with long latency compute operations, which are tagged as “terminals” for our clustering process. The algorithm then traverses the array, starting a new cluster every time a “terminal” is added. The detailed steps involved are shown in Algorithm 1.

As a DAG can have multiple valid topological orderings, many different implementations can potentially be generated from one input dataflow graph. The exploration of this design space is currently in progress, and the results reported later in this paper only represent our first attempt in evaluating potential benefits of the described approach. The quality of results may improve further upon application of more optimization techniques in our flow.

One important part of our algorithm involves the identification of long latency computation in SCCs. These operations are the ones which cannot be completed within a clock cycle, and their categorization ultimately depends on the target frequency of the final implementation on the FPGA. Currently, we leverage Xilinx’s Vivado HLS to generate latency estimates for various compute operations. With a target clock frequency of 150MHz, for instance, floating point multiply takes four clock cycles while a 32 bit integer addition can be completed within a cycle. As Vivado HLS is eventually used as the backend for our HDL generation, it provides accurate annotations for our flow.

It is also worth mentioning that some of the dependencies in the dataflow are implicitly carried by memory accesses. If two memory operations access the same location and one of them is a store, their order in the original program execution must be preserved—a dependency edge needs to be added between them. Since the generation of the subgraphs is performed around strongly connected components in the original dataflow, it is important to avoid adding unnecessary memory dependency edges. To achieve this, our flow currently relies on alias analysis to perform partitioning of the memory space. Accesses to disjoint partitions can be safely reordered. When the source code contains pointer arithmetic, compile time alias analysis may produce overly conservative results, in which case user annotations can be used to provide hints to the tool, similar to [20]. This partitioning of memory space naturally leads to creation of multiple data access interfaces, whose interactions with the memory subsystem are customized, as will be elaborated in section IV-B.
C. Communication Between Decoupled Subgraphs

The dependencies between the generated subgraphs necessitate the insertion of communication primitives. In our case, these correspond to the push and pop operations to/from a hardware queue between processing stages. Other than the explicit data flows, control dependencies are also communicated using branch target tags, such that the locally duplicated branch operations can synchronize across different subgraphs. In the case where ordering of memory accesses needs to be enforced, special tokens are sent between modules.

Eventually, in the hardware module synthesized from each subgraph, both push and pop from a FIFO are blocking. Flow control between different processing stages is thus naturally introduced. As shown in figure 2, subgraphs whose first instruction reads value off a FIFO should be placed in an infinite loop to ensure it is repeated the right number of times, which is especially important if the subgraph is strictly within a loop. The execution of the entire processing pipeline is completed when all modules are idling waiting for inputs and all hardware queues are empty. The runtime behavior of this pipeline thus resembles a streaming processing engine, with uncertainties introduced by memory access nodes smoothed out by FIFOs.

The implementation of our clustering algorithm and the insertion of the communication primitives leverage the LLVM infrastructure [21]. The LLVM front end converts the instructions in the original program to the single static assignment (SSA) form, which makes it easy to track dependencies and thus facilitates all the steps in our algorithm. Figure 2 illustrates the result of clustering on our earlier example. Each of the generated subgraphs (SGs) corresponds to a decoupled stage in figure 1. For better readability, we have converted the LLVM intermediate representation to a less verbose C-like version. Operators not available in C are explained in the figure.

IV. MEMORY OPTIMIZATION

A. Pipelining of Memory Transactions

When creating the pipeline of decoupled processing modules, each memory operation is assigned to a subgraph and the generation of memory requests are synchronized with the execution of the associated module. For instance, in the subgraph shown in figure 3, the HLS tool eventually responsible for RTL generation will need to create a unified schedule where the loop counter addition (line 13), load (line 9) and push (line 10) operations are each assigned to a fixed time slot. As the entire module would be stalled when the load misses, no further memory transactions are initiated even though the address needed for the next load can be computed, and the downstream FIFO has enough empty space. Meanwhile, modern memory subsystems usually have the capability to handle many outstanding memory transactions, in fact, their bandwidth has been improving much faster than their latency [22]. It is therefore undesirable for these hardware components to be artificially sensitized towards memory access latencies, resulting in a underutilization of the bandwidth.
To resolve this issue, our flow again splits the involved memory access operation into two disjoint portions: push_addr and send_req. As shown in figure 3, push_addr takes the place of the load instruction in the original subgraph, and pushes the addresses onto a newly added FIFO. As our algorithm always terminates a cluster after adding a memory access, the response data are immediately pushed to a FIFO linked to downstream subgraphs. The data count of this FIFO is monitored by the send_reg module. When enough space is present, an address is popped off the address FIFO and a new memory transaction is initiated. Data returned from the memory subsystem are routed directly to the downstream FIFO. Store instructions are dealt with in a similar fashion, with two new FIFOs accommodating incoming addresses and data respectively. With this special transformation, each memory access node is capable of pipelining many outstanding requests so long as the memory interface is ready.

Not all memory accesses undergo the above transformation. There are cases where the result of a load, or the completion of a store is necessary for the execution to proceed, as determined by the dependency constraints. A classic example is the pointer chasing in linked list traversal, where the address for the subsequent memory request would not be available until the current load gets its response. In general, if a memory access is in a dependency cycle carried by the innermost loop, our flow categorizes it as non-optimizable. Here we assume that the input to our tool has already undergone potentially helpful high level optimizations. It is well known that for regular applications with statically analyzable memory access patterns, techniques like loop interchange can move the dependency cycle to the outer loops [23]. The applicability of these transformations, however, is not within the scope of this paper.

B. Customization of Data Access Mechanism

Non-regular application kernels often contain a variety of memory access patterns, i.e. streaming, strided or random. General purpose processors use caches as a best effort solution to serve all the different interminglings of these patterns in various applications. The flexibility of the FPGAs, on the other hand, allows for customization of the data access mechanism.

In our flow, partitioning of the memory space has provided an opportunity to create better hardware for memory access on the reconfigurable fabric. Each independent data access interface, corresponding to one memory partition, can be supported differently according to the nature of the address stream it generates. In particular, for streaming type accesses, there is no reuse of data, and thus our flow does not allocate an on-FPGA buffer. Rather, the send_reg module described in section IV-A is modified to send burst requests, concatenating multiple load/store in the original program execution. On the other hand, if there is a cycle of dependency through memory, an on-FPGA buffer would be beneficial. Our flow currently adds a general purpose cache in this case, but if the particular address stream is analyzable and the reuse distance can be determined statically, structures like smart buffers [24] can be incorporated. Even in the case when the memory accesses are random and a general purpose cache is the only plausible solution, its size and associativity can be adjusted according to a runtime profile.

V. HARDWARE GENERATION

To create the RTL implementation from each subgraph, we convert the LLVM intermediate representation back to C syntax and then feed it to an existing HLS tool. By building a source to source transformation flow, we can ensure portability across different back-end RTL generation tools.

At the instruction level, the LLVM to C translation is rather straightforward. Most LLVM IR instructions can be mapped directly to C statements, as seen in figure 2. The only non-trivial transformation we perform is in dealing with the \( \phi \) operations, which generate values of variables based on the the incoming control edges. In the case where the sources of the data are all within the same subgraph, the assignments to its sources are replaced with assignment to its output, and the \( \phi \) instruction itself is removed. On the other hand, if a data source is in another subgraph, a “pop” is inserted to the basic block where the data is produced, but assigning the result directly to the output variable of \( \phi \). An example of this conversion is shown in figure 4.

![Fig. 4. Conversion of SSA to C](image)

As each subgraph contains instructions from only a subset of the basic blocks in the original program, it does not always have a complete control flow. To ensure each processing module generated is self-contained and has well-defined behavior, extra basic blocks are sometimes added. Our tool finds the nearest common dominator of all the basic blocks in a subgraph and add all the control flow statements between this dominator and the other basic blocks. Consequently, there is a unique entry block for every subgraph, and different module will traverse the exact same execution path when the processing pipeline is active.

For the next step, where all the components are connected together by FIFOs, we rely on vendor specific IP generators. The sizing of these FIFOs can affect the final performance of the processing pipeline, and thus may require application specific tuning. The exploration of that design space has been left to future work. Currently, all communication queues between decoupled modules have 64 entries. In addition, vendor supplied IPs are also used for on-FPGA cache and the interconnect, which is used to bridge the processing pipeline and the memory subsystem. All the steps involved in our pipeline generation flow are summarized in figure 5.

VI. EXPERIMENTAL EVALUATION

To demonstrate the benefits of our approach, processing pipelines are synthesized from several benchmark kernels. These benchmarks are non-regular, as their control flow and
data access patterns depend on the runtime data. The sparse matrix vector (SpMV) multiply uses compressed sparse row (CSR) format to store the matrix. Loads from an index array are performed before numbers can be fetched for the actual floating point multiply. Knapsack is a problem in combinatorial optimization. Given a collection of items each with its own weight and value, knapsack tries to select a subset of them such that the total profit is maximized while the weight limit is not violated. Floyd-Warshall takes a graph as input and computes the shortest distances between any pairs of vertices. For both knapsack and Floyd-Warshall, the memory addresses accessed come from the result of computations. Iterative depth first search (DFS) is again a widely used graph algorithm. The version used for our experiment makes use of a stack and operates on pointer based data structures. All the kernels are sequential code, to which no high level optimizations, e.g. loop unrolling, have been applied. The irregularity in memory access and execution path in these benchmarks makes it hard for existing HLS tools to generate efficient hardware. The conventional accelerators, when implemented on the FPGA, are also very sensitive to the latency of data access, due to the high ratio of memory operations to computation.

Table I describes the characteristics of the input data set for each benchmark. As our approach is primarily used for cases where off-chip communication plays a significant role in determining the final performance, the input data size are chosen to be much larger than typical on-FPGA cache. For smaller problems where the entire input data set can be buffered on chip, the conventional DMA+accelerator approach, as described in [6], would not suffer from variable data access latency and our decoupled processing pipelines would offer little advantage.

The physical device used for the experiments is the Zynq-7000 XC7Z020 FPGA SoC from Xilinx, installed on the ZedBoard evaluation platform. The SoC is divided into two parts: an ARM-processor based processing system (PS), and the programmable logic (PL). The baseline for our evaluation is the performance of each software kernel running on the ARM core in the PS. It is an out of order, dual issue hard processor running at 667MHz. The Zynq platform also provides two options for the accelerators in PL to access the main memory subsystem: through the accelerator coherence port (ACP), or the high performance (HP) port. The former connects to the snoop control unit in the processing system and thus uses/modifies the processing system’s on chip cache. The HP port connects directly to the memory controller, which necessitates the flushing of cache lines by the processor if a cached data structure is accessed by the accelerator. In either case, if memory states are also buffered in the PL with caches, they need to be explicitly pushed to the processing system side after the accelerator finishes running. As both ACP and HP are slave ports, they provide no mechanisms to extract data from the programmable logic when the ARM processor is running. The interaction between the generated accelerators and the main pieces of the FPGA SoC is shown in figure 6.

![Fig. 6. Implementation of Processing Pipeline in FPGA SoC](image)

In our study, Vivado HLS, a state-of-the-art high level synthesis tool provided by Xilinx, is used for generating the conventional accelerator (Con.ACC) as well as the individual stages in our decoupled processing pipeline (DPP). With the target clock period set to 8ns during HLS, the tightest timing constraints post place & route implementations managed to meet range from 111 to 150MHz. All design points shown in this section use the highest achievable frequency as the actual operating clock frequency.

A. Performance Comparisons

In figure 7, performance of the different implementations are presented. Conventional accelerators and decoupled
processing pipelines with different memory subsystem configurations are compared. All the performance numbers are normalized to the baseline.

In all four benchmarks, accelerators generated directly from software kernels using conventional HLS flow actually result in a performance degradation compare to running the kernels on the hard processor. Even with on-PL caches, these accelerators only manage to achieve throughput less than 50% that of the baseline. The superscalar, out-of-order ARM core is capable of exploiting instruction level parallelism to a good extent and also has a high performance on-chip cache. The additional parallelism extracted by the HLS tool is evidently not enough to compensate for the clock frequency advantage the hard processor core has over the programmable logic and the longer data access latency from the reconfigurable array.

With our methodology, the processing pipelines generated are rather competitive against the hard processor, even without a reconfigurable cache. For SpMV multiply, knapsack and Floyd-Warshall, when access pattern customized DPPs are directly connected to the PS through the ACP, the average performance is 2.3 x that of the baseline—representing an 8.4 x gain over the conventional accelerators. Upon the addition of caches, the average runtime of DPPs was reduced by 18.7%, while that of the conventional accelerators was cut by 45.4%. The gap between their performance is thereby reduced from 8.4 to 5.6 times.

A few transformations were mentioned in section IV-B for optimizing data access mechanisms according to the observed pattern. In the three benchmarks other than DFS, the performance of DPPs with cached ACP port improved by 72% when those transformations are applied. This boost in performance can be attributed to multiple factors. The burst accesses make more efficient use of the memory bandwidth, as each request/response pair carries more useful data. Also, the clock frequency of the system generally improves when the cache is shared by a smaller number of memory interfaces. Meanwhile, the interference between different streams of addresses is also minimized when they are separated, which reduces conflict miss in the cache. In addition, as the cache space is shared by a smaller number of data structures, it is less strained by its limited capacity.

It is also apparent that our approach has its limitations, as demonstrated by its ineffectiveness in the benchmark depth first search. The kernel performs very little computing but lots of memory accesses. The use of a stack in DFS also creates a dependence cycle through the memory and consequently, the performance is fundamentally limited by the latency of memory access. Thus there were only small differences between the performance of the conventional accelerator and the decoupled processing pipeline. Besides, the memory access pattern does not provide many opportunities for optimizations. As a result, DPP and Con.ACC achieves performance far below that of the baseline, which has a much higher clock frequency and a faster cache.

Overall, for kernels suitable for FPGA acceleration, there is a significant performance advantage in using decoupled processing pipelines. If we compare the best results using DPP to conventional accelerators, we see improvement of 3.3 to 9.1 times, with an average of 5.6.

B. Area comparison

To quantify the impact of our proposed methodology on area, we have compared the FPGA resource usage of conventional accelerators and the decoupled processing pipelines. Table II shows the results, where each accelerator is complemented with two different memory subsystem configurations.

The difference in area between DPPs and Con.ACCs is effected by two factors. There are additional costs associated with the communication primitives and FIFOs for the DPP implementations. On the other hand, the original programs are...
partitioned into subgraphs and separately turned into hardware in DPPs, which sometimes can reduce the depth of the internal pipeline in the processing modules, resulting in area savings. The overall change therefore depends on which factor plays a larger role, and is ultimately application specific.

VII. CONCLUSION

This paper presents a method for generating processing pipelines from nonregular loop nests. This method decouples memory accesses and long dependence cycles in the control dataflow graph, such that cache misses do not stall the other parts of the accelerator. Optimizations are also incorporated for more efficient use of memory bandwidth. Comparing against a state-of-the-art high level synthesis tool, the new approach produces hardware engines with an average 5.6 times performance advantage.

ACKNOWLEDGEMENT

This research is supported by the Berkeley Wireless Research Center and the ASPIRE Lab. The ASPIRE Lab is funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and industrial sponsors and affiliates: Intel, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

REFERENCES


<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ACP</th>
<th>ACP + 64KB Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LUT</td>
<td>FFs</td>
</tr>
<tr>
<td>SpMV Multiply</td>
<td>Con.ACC</td>
<td>9875</td>
</tr>
<tr>
<td></td>
<td>DPP</td>
<td>8577</td>
</tr>
<tr>
<td>% change</td>
<td>-13.1</td>
<td>-3.1</td>
</tr>
<tr>
<td>Knapsack</td>
<td>Con.ACC</td>
<td>7672</td>
</tr>
<tr>
<td></td>
<td>DPP</td>
<td>8089</td>
</tr>
<tr>
<td>% change</td>
<td>+25.7</td>
<td>+17.3</td>
</tr>
<tr>
<td>Floyd-Warshall</td>
<td>Con.ACC</td>
<td>2391</td>
</tr>
<tr>
<td>DPP</td>
<td>7659</td>
<td>7210</td>
</tr>
<tr>
<td>% change</td>
<td>+207.5</td>
<td>+104.3</td>
</tr>
<tr>
<td>DFS</td>
<td>Con.ACC</td>
<td>4870</td>
</tr>
<tr>
<td>DPP</td>
<td>8500</td>
<td>7813</td>
</tr>
<tr>
<td>% change</td>
<td>+76.9</td>
<td>+53.5</td>
</tr>
</tbody>
</table>