CGPA: Coarse-Grained Pipelined Accelerators

Feng Liu Soumyadeep Ghosh Nick P. Johnson David I. August

Princeton University, Princeton, NJ
{fengliu, soumyade, npjohnso, august}@princeton.edu

Abstract

High-level synthesis (HLS) tools dramatically reduce the non-recurring engineering cost of creating specialized hardware accelerators. Existing HLS tools are successful in synthesizing efficient accelerators for program kernels with regular memory accesses and simple control flows. For other programs, however, these tools yield poor performance because they invoke computation units for instructions sequentially, without exploiting parallelism. To address this problem, this paper proposes Coarse-Grained Pipelined Accelerators (CGPA), an HLS framework that utilizes coarse-grained pipeline parallelism techniques to synthesize efficient specialized accelerator modules from irregular C/C++ programs without requiring any annotations. Compared to the sequential method, CGPA shows speedups of 3.0x–3.8x for 5 kernels from programs in different domains.

1. Introduction

Since technology scaling no longer provides previously seen rates of performance improvements, microprocessor designers have adopted a number of other techniques to improve processor performance at reasonable energy costs. An entire class of techniques improves performance by offloading computation intensive parts of programs from a general-purpose processor to customized accelerators [15, 17, 18, 24, 31, 34]. One approach to build these customized accelerators is to manually design dedicated circuit modules for each application using hardware description languages (HDLs) [24, 26]. While this approach often yields the best result, it requires significant non-recurring engineering costs to convert algorithms to HDL specifications. Another less labor-intensive approach is to use high-level synthesis (HLS) tools to generate customized accelerators from programs written in C or other high-level programming languages [4, 12, 13, 16, 30, 31, 32, 37].

HLS tools have been successful in creating customized accelerators for scientific computation [30] and digital signal processing (DSP) applications [12, 13, 16, 32, 37]. One typical feature of these applications is that their hot spots consist of affine loop nests. Such loop nests enable a series of loop transformations that expose loop-level parallelism to overlap instruction execution [7, 29, 35]. Additionally, the affine loops may also enable special circuit modules, such as systolic arrays and shift registers, to reduce memory traffic and improve performance [9, 10]. However, in the presence of loops with complex control flows or irregular memory accesses, these HLS tools do not extract loop-level parallelism well, leading to poorly performing accelerator designs.

DAC'14, June 01–05, 2014, San Francisco, CA, USA.

http://dx.doi.org/10.1145/2593069.2593105

In reality, a large portion of programs are implemented with non-array data structures and imperfect loop nests. This restricts the scope for parallelism within one or a few loop iterations, and generates hardware modules with limited performance improvement over general-purpose processors [31]. A new technique is required to generate efficient accelerators for a large class of programs with irregular memory accesses and imperfect loop nests.

To meet this requirement, this paper proposes Coarse-Grained Pipelined Accelerators (CGPA), an HLS framework which uses coarse-grained pipeline parallelism to generate efficient hardware accelerators for loops from unannotated C/C++ programs. CGPA leverages two distinct insights to improve efficiency and applicability for HLS. First, complex loop bodies with irregular memory accesses and imperfect loops usually contain coarse-grained code sections performing different tasks. HLS tools should separate and modularize these tasks to build an efficient system. Second, these complex loop bodies usually contain sections which are parallelizable. Coarse-grained decoupled pipelining techniques [28, 27] can exploit the presence of these parallelizable sections to enable a type of parallelism not exploited by existing HLS tools.

CGPA automatically partitions individual loops into separate pipelined stages and generates buffer-connected hardware modules for these stages. Pipelining enables the overlapping of execution of an earlier iteration of the loop with a later iteration and also allows the synthesized hardware to tolerate variable memory latency. CGPA also utilizes hidden data-level parallelism within the pipelined stages to achieve high performance. Experiments show that for 5 kernels from different domains, CGPA gives a 3.3x geomean performance improvement at 20% average energy overhead compared to non-parallelized hardware specialization results.

In summary, the contributions of this paper are:

1. the CGPA framework, which can identify and split parallelizable sections from non-parallelizable ones for individual loops with complex control flow and irregular memory access,
2. the CGPA tool, the first high-level synthesis tool to exploit coarse-grained pipeline parallelism for single loops to create highly efficient hardware accelerators, and
3. an experimental evaluation of the performance and energy efficiency of hardware accelerators generated by CGPA.

The rest of the paper is organized as follows: Section 2 provides background information on the loop parallelism techniques used in existing HLS tools and presents a motivating example to show the limitations of these tools. Section 3 describes the design and implementation of our pipelining technique. Section 4 contains the evaluation, followed by conclusions in Section 5.

2. Background and Motivation

2.1 Loop Parallelism in Existing HLS tools

Exploiting parallelism by finding independent operations in programs and overlapping their execution is an effective way to create high performance hardware for programs. Existing HLS tools

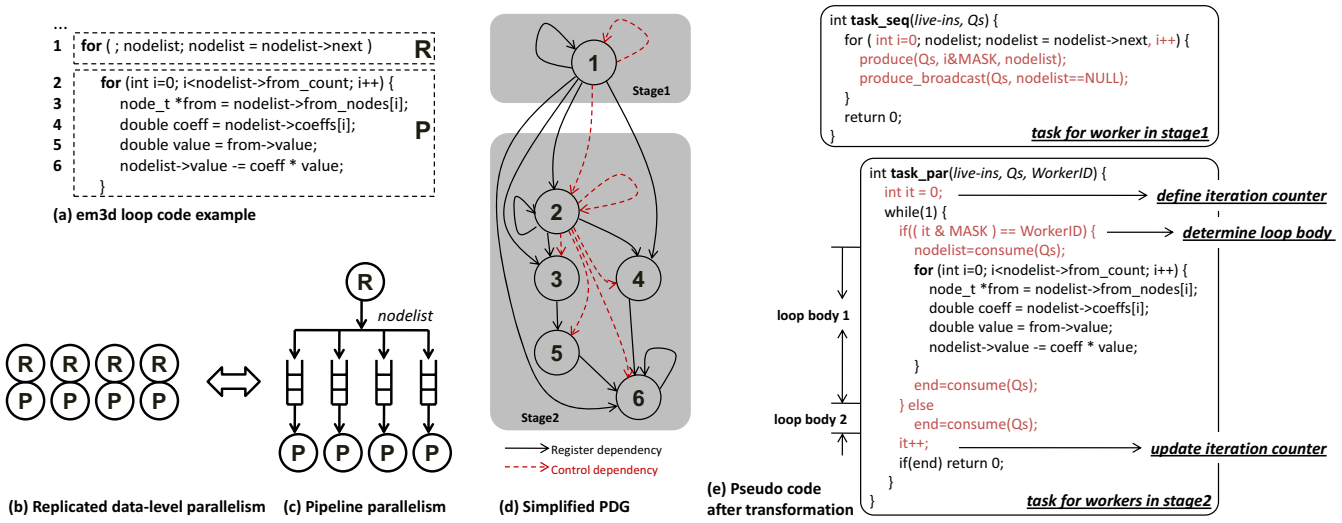

Figure 1: (a) Source code of the main loop of em3d program, with sections annotated as replicable or parallel; (b) Data-level parallelism is exploited by duplicating the replicable section; (c) Coarse-grained pipeline parallelism is exploited by separating the replicable section from the rest of the loop; (d) A simplified source-level Program Dependence Graph (PDG) of the main loop in em3d; (e) Pseudo-code of tasks for both hardware stages. The code in red is the overhead generated by the compiler.

exploit parallelism at both instruction-level and loop-level [2, 33, 29, 18, 15]. Unless combined with other techniques such as if-conversion or loop unrolling, the scheduling window for instruction-level parallelism (ILP) techniques is small. Thus, these techniques usually yield limited performance benefits [23]. As a result, modern HLS tools also utilize loop-level parallelism, which increases the scheduling scope across multiple loop iterations and can potentially yield higher performance.

Existing HLS tools target two main types of loop-level parallelism. One class of tools targets loops with **data-level parallelism** [3]. If each loop iteration operates on disjoint data, parallel hardware modules can be designed for to *fully* interleave loop iterations. In extreme cases, each iteration executes the same sequence of operations, leading to Single Instruction Multiple Data (SIMD) style parallelism [3]. In reality, however, few outer loops fit this pattern without additional transformations, thus limiting the applicability of HLS tools that exploit this kind of parallelism.

A second class of HLS tools exploits loop pipelining to *partially* interleave loop iterations [33, 29, 15, 18, 2]. Different pipelining schemes such as **pipeline vectorization** [33] and **software pipelining** [29, 15, 18, 2] have been adapted to HLS for this purpose. Pipeline vectorization is applicable to loops without true loop-carried dependencies or with only regular loop-carried dependencies. Software pipelining has been widely used to overlap computations from different iterations. However, complicated control and data dependencies existing in loops limit the number of independent operations found from different iterations.

Complementary loop transformations such as loop unrolling, flattening, permutation interchange and tiling can be used by HLS tools to expose loop-level parallelism for innermost loops [35, 36]. However, these transformations fall short when specializing programs with either complex control flows or irregular memory accesses. The following subsection uses an example to show the limitations of existing work and how CGPA address these issues.

2.2 Motivating Example

Figure 1 shows the source code of the main loop in em3d, which simulates electron microscope tomography by constructing two linked-lists to build a *N*-to-*N* bipartite graph [5]. This code has a number of features that motivate CGPA: recursive data structures, irregular memory accesses, and non-affine loop nests.

The input to the core em3d algorithm consists of nodes in a linked list. Each node has four data members: `value`, `from_count`, an array of `from_nodes` which points to the nodes of the other linked-list, and an array of `coeffs` (*coefficients*) for each `from_nodes`. The outermost loop *traverses* the linked list, and *updates* the `value` of each node by subtracting all the weighted `values` of its `from_nodes` using an inner loop. Even though this inner loop has some independent instructions across iterations, an attempt to exploit loop parallelism for this inner loop may fail because:

1. The iteration count, determined by `nodelist->from_count` (less than 10 for most cases), limits the amount of parallelism that can be exploited and also introduces the overhead of determining the control of loop exits. As a result, loop optimizations such as software pipelining cannot be applied.
2. The final weighted value reduction step in the loop induces a loop-carried dependency, which prevents a full application of data-level parallelism. Also the non-constant loop iteration numbers for each node disable the applicability of a circuit structure like the *reduce* module in [26].

Thus, the scheme for generating efficient accelerators for em3d should focus on its outer loop. At the algorithm level, we can divide the outer loop computations into two sections: *traversal* and *update*. The linked list traversal section (line 1 in Figure 1(a)) determines the address of the node used in an iteration and also the termination of the loop. We call this set of instructions a *Sequential Section* because fetching the node addresses must be serialized. Furthermore, because this section has no side-effects (for example, among other things, it does not store to the memory), we refer to this special sequential section as a *Replicated Section*, which means it is safe for multiple hardware modules to execute it redundantly. The *update* section for each node (lines 2-6 in Figure 1(a)) in one iteration is independent of updates to all other nodes, and can thus be executed in parallel (as long as the node addresses are known). We refer to this section as a *Parallel Section*. Existing HLS tools can not optimize this outer loop, due to the existence of the Sequential Section, which introduces a loop-carried dependency, non-affine memory access and non-constant loop boundary.

CGPA can apply two novel loop parallelism techniques to build high performance hardware modules for this outer loop. One technique, called **replicated data-level parallelism** leverages the insight that computations from replicated sections can be safely executed as multiple parallel copies [19]. For example, CGPA could create four identical copies of the traversal section — one for each hardware module, as shown in Figure. 1(b). During execution, each hardware module executes both fetch and update in one iteration. In the next three iterations, the module skips update and only executes fetch. By replicating fetch and distributing updates in a round-robin manner across the four hardware modules, CGPA can create replicated data parallel accelerators for em3d. The redundant fetching is useful to calculate the correct node addresses for the corresponding updates. However, it causes unnecessary memory access overhead in each module.

CGPA can also adopt an approach, called **decoupled pipeline parallelism** to improve outer loop performance and generate accelerators similar to the results of manual accelerator designs [27]. This approach uses a set of FIFO buffers to separate the linked list traversal module from that for node updates. Since the traversal section only goes over the linked list and fetches node addresses, it can progress much faster than the update section. Thus one *sequential* traversal module can supply node addresses to multiple *parallel* update modules in another stage, as shown in Figure. 1(c).

With this decoupled pipeline design, when control enters the loop, the hardware modules for both stages are invoked by the same start signal. The module in the first (sequential) stage begins fetching node addresses one by one, and assigns the node address values to the FIFO buffers of the parallel modules in the parallel stage in a round-robin fashion. The sequential stage stalls when there are cache misses or the corresponding buffers are full. Each module in the second (parallel) stage waits until there are node addresses in its buffer, and starts to process the update by fetching the node address directly from the buffer. After completing one iteration, the module in the parallel stage can get another pointer from the buffer, or stalls if the buffer is empty. This pipelining execution method brings the following two benefits:

1. **Tolerating Variable Latency**: In this example, memory accesses during the linked list traversal are irregular and might have variable latency due to cache misses. However, the buffers between stages ensure that the impact of variable latency is limited to one stage and does not cause stalls in the subsequent stages as long as the buffers are not empty.
2. **Enabling More Parallelism**: Since the sequential stage is split from the parallel stage by FIFO buffers, the updates of nodes from different iterations become completely independent of each other, thus enabling data-level parallelism within the parallel stage.

The decoupled pipeline model has proved efficient for streaming applications [16], and has also been used to design accelerators for applications such as hash indexing [21]. CGPA is the first HLS tool to automatically extract this type of parallelism from a *single loop* and generate efficient pipelined hardware modules for it.

3. Exploiting Coarse-Grained Pipeline Parallelism

3.1 CGPA Accelerator Architecture Overview

Figure 2 shows a logical view of coarse-grained pipelined accelerators with a Sequential–Parallel–Sequential (S-P-S) pipeline. The number of stages for different applications is not fixed, and is determined automatically for each application by the CGPA compiler's partition algorithm. The significant difference between the CGPA designs and existing accelerator designs is that there are multiple stages of the accelerator for one *single loop* and they are separated by FIFO buffers. Each hardware module with independent control that implements instructions from the original loop is called a *worker*. Each worker has its own independent control circuit and dedicated memory ports to the cache.

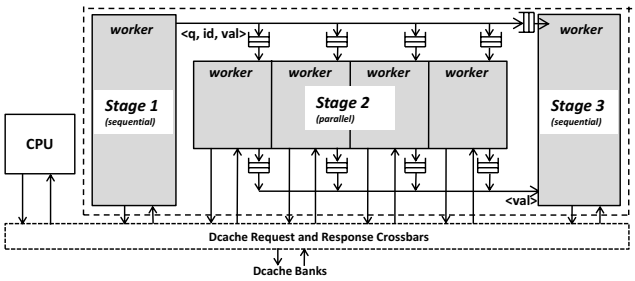

Figure 2: A logical view of CGPA architecture within the dashed box. Each grey box contains circuit modules customized for the targeted loop and generated by CGPA compiler. In this figure, a Sequential–Parallel–Sequential (S-P-S) 3-stage pipeline is shown.

The design of this pipelined accelerator can be embedded in a general-purpose co-processor such as conservation cores [31] and Legup accelerator [4] to improve the performance of these co-processors. It can alternatively be implemented as a standalone accelerator to improve the performance of targeted loops. This paper explores the first configuration.

3.2 CGPA Workflow

Figure 3 shows the workflow of the CGPA tool. The tool is built on top of the LLVM compiler infrastructure [22] (revision 164307). The tool accepts unannotated sequential C/C++ programs. The LLVM frontend (clang) translates the source code to intermediate representation (IR) for further analyses and optimizations. The compiler identifies hotspots in the code via a simple profiling step. A series of analyses and code transformations are applied to create the pipeline specifications from the IR. Subsequently, the compiler splits the program into two parts: one to be executed on the general-purpose processor and the other (containing the transformed pipelines) to be implemented as accelerators. Additionally, the compiler generates wrapper functions to invoke the accelerators from the processor and pass it the necessary arguments. The code to be executed on the general-purpose processor is then compiled to binary, and a hardware backend translates the second part to Verilog descriptions and finally the device programming files.

3.3 Pipeline Generation

The specification of the pipeline is generated during the analysis and optimization phase. In this phase, a set of common optimization passes such as dead code elimination, strength reduction, and scalar optimizations are applied before generating the actual pipeline. There are three main steps in the pipeline generation stage: building the program dependence graph (PDG), pipeline partition, and pipeline transformation.

Building the PDG: The compiler first builds a program dependence graph (PDG) for each program. Each node in the PDG represents an instruction in LLVM IR and each edge between the nodes represents a control or data dependency. During this process, a set of alias analyses can be applied to remove dependency edges between two memory instructions. For example, in the example of

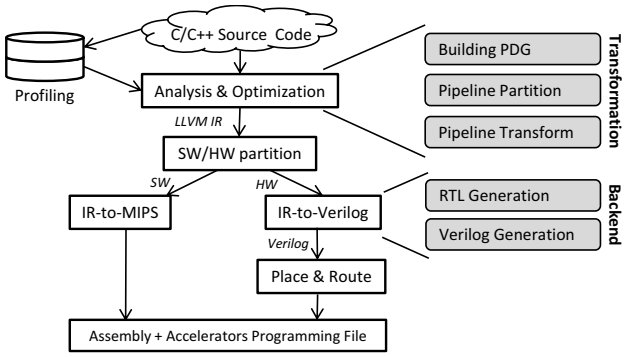

Figure 3: Workflow for CGPA HLS Framework

Section 2.2, several static analysis algorithms can determine that `from` and `nodelist` nodes are from different linked-lists and disjoint from each other [14]. After the PDG is built, the compiler consolidates all the strongly connected components (SCCs) in the PDG to create a directed acyclic graph (DAG) [11, 20]. Each component is classified as: parallel, replicable, or sequential [19]. Parallel SCCs contain no loop-carried dependencies and thus, can be executed in parallel. The other SCCs are either replicable (do not contain an instructions with side-effects) or sequential (may contain instructions with side-effects).

Pipeline Partition: This is a coarse-grained instruction scheduling step that assigns instructions to different pipeline stages. The partitioning algorithm is adapted from the Parallel Stage Decoupled Software Pipelining (PS-DSWP) [27] algorithm. The main difference between CGPA and PS-DSWP lies in the identification of replicable sections, and deciding whether it should be inserted in the parallel stage as replicas or into the sequential stage. Generally, inserting replicable sections into a sequential stage will increase communication, because the results of the replicable sections must be sent explicitly to the following stages. Conversely, duplicating the replicable section in parallel stages will increase the amount of computation and memory accesses. The CGPA framework only duplicates lightweight replicable sections which do not contain load and multiply instructions. This is based on the intuition that time and resources required for the replicable section without these instructions are less than those for communicating results via FIFO buffers. Figure 1(d) shows that replicable sections with heavyweight load instructions are identified and inserted into a sequential stage (Stage 1), and the remaining instructions are grouped as one parallel stage (Stage 2). The required number and stage connections of the buffers are also determined at this step.

Pipeline Transform: This step forms a set of control-equivalent loops for the workers based on the results of the partition step. Control-equivalent means that all workers from all stages have the same loop iterations and exit points as the original loop, even though their bodies have been assigned different instructions. Then the compiler creates tasks for each stage, with the loop live-ins and buffers as arguments. Tasks for parallel stages have one additional argument to indicate the unique identification (ID) for the worker (relative to all other workers in the same stage). For the body of the tasks, the instructions of the original loop are distributed according to the result of the pipeline partition. Moreover, loop control branch instructions of the targeted loop are also duplicated across the tasks and the destination of original branch instructions are modified to recreate the original loop structure in each task.

Both register and control dependencies between stages are communicated via FIFO buffers, and the compiler automatically inserts communication primitives into tasks. For data dependencies, if the *definition* and *use* of a variable are in different stages, a *produce* primitive is inserted after the definition, and a *consume* primitive is inserted at the point in the later stage which corresponds to the definition. Also for control dependencies, the compiler needs to *broadcast* the condition of the branch instructions to all the workers in the following stages as a data dependency, as shown in Figure 1(e).

One significant difference between CGPA partitioning and previous work [27] is the handling of the replicable section and loop termination. Duplicating lightweight replicable sections introduces loop-carried dependencies in the parallel stage. To solve this, the CGPA compiler creates two copies of the loop body in the tasks for workers in a parallel stage, as shown in Figure. 1(e). One copy (loop body 1) has real computations which are the instructions (both parallel and replicable sections) assigned to this worker; another copy (loop body 2) is only for the replicable section. The compiler creates a new basic block to use the worker ID and iteration counter to decide which loop body the control should enter at the beginning of each iteration. For the loop termination of workers in the parallel stage (when the parallel stage is not the first stage), we adopt the same strategy, and consume the branch condition from the previous stage in both loop bodies, and guarantee that it can exit in any iteration, as shown in Figure 1(e).

Once tasks are generated, the original loop in the parent function can be replaced by a set of calls to the generated tasks. This is done by inserting *parallel_fork* and *parallel_join* primitives with the corresponding arguments. The live-outs are communicated back to the original parent function by inserting *store_liveout* exactly before the exits of the tasks, and *retrieve_liveout* before the uses of the liveouts. Table 1 summarizes all the primitives inserted by the compiler during the pipeline transformation.

3.4 CGPA **Compiler Backend**

RTL Generation: The backend of the compiler builds the RTL description of the hardware modules from IR. We adapt an open-source Verilog backend of LLVM for this purpose [4]. The instruction scheduling phase creates a control flow graph (CFG) of the offloaded program. The nodes of the CFG can be split into multiple states of a finite state machine (FSM), after scheduling instructions at different clock cycles (represented by one state in the FSM). Since our compiler generates pipelined parallel modules, scheduling constraints for the new primitives should be added to ensure correctness and performance. We use the notations from [8] to express the set of new constraints that must be preserved: $sv_{beg}(v)$ represents the scheduling variable associated with the starting state of instruction v; $C_k(l)$ are primitives from Class k with LoopId l (Table 1), M is the set of memory access instructions, B represents branch instructions. Then the following additional scheduling constraints are introduced:

$$\forall f_i, f_j \in C_1(l): \quad sv_{beg}(f_i) - sv_{beg}(f_j) = 0 \quad (1)$$

$$\forall f_i \in C_1(l1), f_j \in C_1(l2): \quad |sv_{beg}(f_i) - sv_{beg}(f_j)| \geq 1 \quad (2)$$

$$\forall m \in M, \forall pc \in C_2: \quad |sv_{beg}(m) - sv_{beg}(pc)| \geq 1 \quad (3)$$

$$\forall b \in B, \forall lo \in C_3: \quad sv_{beg}(b) - sv_{beg}(lo) = 0 \quad (4)$$

Constraint 1 guarantees that the parallel invocation of hardware modules from the same loop will be within the same cycle, and only when all the modules finish, the state machine of the parent module will continue. Constraint 2 guarantees that the parallel invocation of hardware modules for different loops will not be invoked in the same cycle. Constraint 3 makes sure the produce and consume primitives will not be scheduled with memory operations, since memory operations can stall the circuit, and cause multiple

Class	Primitive	Arguments	Description
1	parallel_fork	*LoopID*, *Task*, *Liveins*, *Buffer*, *WorkerID*	In the current state, invoke a hardware module for *Task* (associated with *LoopID*) and read register values of *Liveins* as input. If this is a parallel worker, *WorkerID* is used as one extra input.
	parallel_join	*LoopID*	Stall in current state until all the workers related to *LoopID* have raised the finish signal
2	produce	Buffer, WorkerID, Value	Push *Value* to the FIFO *Buffer* with index *WorkerID*
	produce_broadcast	Buffer, Value	Push *Value* to all the workers connected to the FIFO *Buffer*
	consume	Buffer	Pop a value from the connected FIFO *Buffer*
3	store_liveout	LiveoutID, Value	Store a liveout value, *Value* with ID *LiveoutID* in a register
	retrieve_liveout	LiveoutID	Read value for a liveout with ID *LiveoutID* from the corresponding register

Table 1: New primitives added to LLVM IR to support worker invocation, dependency communication, and register value passing across hardware modules.

pops/pushes to the FIFO buffers if they are scheduled within the same state. Finally, constraint 4 allows stores of live-out values only when the loop exits. The backend in [4] is fully utilized to translate the FSM with scheduled instructions to the RTLs of datapath and control circuits.

Verilog Generation: Given the RTL specification, the backend generates the Verilog code automatically. We also create a hardware circuit library to support all the primitives shown in Table 1. During the Verilog generation phase, the wire connections between the generated modules and the hardware module in the library is completed automatically. Besides the Verilog code, the compiler also generates a testbench to verify the design. All the Verilog designs of our benchmarks passed the verification.

4. Experiments

4.1 Methodology

Evaluation Framework: The CGPA compiler is based on LLVM (revision 164307). We adapt the backend for CGPA from an open source Verilog backend [4], which generates both design and testbench files. The CPU/accelerators heterogeneous system from [4] is based on Altera DE4 which features a Stratix IV FPGA [1]. A 32-bit MIPS software core executes the CPU part of the program. Both the instruction cache and the data cache are directly-mapped with 512 lines and 128 byte blocksize. The instruction cache and the data cache have 1 and 8 ports, respectively. For all the pipelined accelerators, we fixed the width of FIFO buffers to 32 bit, the depth to 16 entries and the number of workers in the parallel stage to 4. We use Quartus II 11.0 to synthesize, fitter and simulate the accelerator part of the program. We set the targeted synthesis frequency to 200MHz. The accelerators generated by CGPA have all been implemented and verified. For performance, we measured the number of cycles for which the kernels of various programs were running. The ALUT resource usage is obtained after place and route, and the power estimated is given by PowerPlay [1] with obtained activity files from a post-fitter simulation.

Benchmarks: Table 2 shows the descriptions of a set of kernels that were accelerated using CGPA and also the corresponding pipelined stages generated by CGPA. We chose these benchmarks because they belong to different domains such as machine learning, graph partitioning, and image processing. Additionally, we are not aware of HLS synthesized accelerators for some of these kernels (hash indexing, em3d, and ks).

4.2 Results

Performance: We consider three data points for each loop kernel: (1) Performance on a MIPS software core; (2) Performance of hardware accelerators generated by Legup [4]; and (3) Performance of hardware accelerators generated by CGPA. Legup was chosen for comparison because it could generate accelerators for each kernel mentioned in Table 2. Additionally, other HLS tools targeting general-purpose programs use a framework similar to Legup [31]. Figure 4 shows the loop speedup numbers for the kernels, relative to the performance on the MIPS software core. The Legup HLS tool gives a 1.85x geomean speedup over the software core. CGPA gives a geomean of 3.3x speedup over the performance achieved by Legup and a geomean of 6.0x speedup over the MIPS software core.

Benchmark	Domain	Description	P1	P2
K-means [6]	Machine Learning	finding the nearest cluster for each node and updating its position	P-S	-
Hash-indexing [21]	Database	computing hash key for each node and indexing it in a linked-list	S-P-S	-
ks	Graph Partition	traversing doubly-nested linked-lists to find a max grain of swapping	S-P-S	-
em3d [5]	3D Simulation	updating value for each node in a linked-list by subtracting weighted value in from_nodes	S-P	P
SIFT 1D-Gaussblur [25]	Image Processing	1D row Gaussian blurring; pipeline vectorization has been applied to reduce memory access	S-P	P

Table 2: Descriptions of benchmarks used. P1: Pipeline Partition with Replicable Section in Sequential Stage; P2: Pipeline Partition with Replicable Section in Parallel Stage

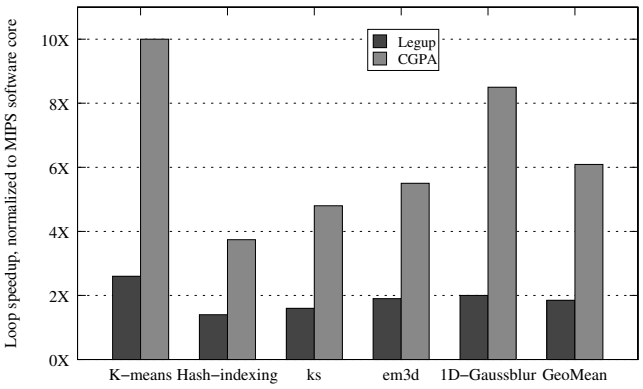

Figure 4: Speedup Results for Loop Kernels of the Benchmarks

Area and Power: Table 3 shows the area and energy overheads for each kernel. For most benchmarks, the ALUT usage relative to Legup is approximately 4.1x. This is not surprising since CGPA creates four parallel workers in the parallel stage to increase performance. Another type of overhead comes from the usage of BRAM to build the FIFO buffers, which are not included in the ALUT usage. Table 3 also shows the power and energy dissipation for each benchmark. The results show that a geomean of 20% energy dissipation overhead is generated by CGPA, over the accelerators generated by Legup for the benchmarks. The sources of the energy overhead are passing values via the FIFO buffers, multi-port cache support and other computation overhead.

Tradeoff: To explore the tradeoff between computation and communication in the presence of replicable sections, we enable replicated data-level parallelism for em3d and 1D-Gaussblur by duplicating the replicated stage in the parallel workers (reported as P2 in Table 3). The pipelining method (P1) outperforms the duplicated replicable section method (P2) by 6% and 15% for em3d and 1D-

Benchmark	Type	ALUT #	power (mW)	energy (uJ)	energy efficiency
K-means	Legup	1696	46	22.1	7.3
	CGPA (P1)	7197	162	22.9	6.9
Hash-indexing	Legup	421	47	12.1	6.7
	CGPA (P1)	2052	150	14.6	5.5
ks	Legup	1371	60	104.5	6.7
	CGPA (P1)	5741	233	131.7	5.3
em3d	Legup	623	72	1.66	6.4
	CGPA (P1)	2842	292	2.24	4.7
	CGPA (P2)	2624	305	2.49	4.2
SIFT 1D-Gaussblur	Legup	1319	53	1.27	7.4
	CGPA (P1)	3806	183	1.35	6.9
	CGPA (P2)	4168	194	1.55	6.0

Table 3: Comparison between CGPA and related frameworks

Gaussblur, respectively. Moreover, as shown in Table 3, the pipelining method can reduce energy dissipation by 11% and 14% respectively for these two benchmarks. For the other benchmarks, replicated data-level parallelism was not found applicable.

The appendix shows case studies of two benchmarks and also includes a discussion of the scalability of CGPA.

5. Conclusion

This paper presents Coarse-Grained Pipelined Accelerators (CGPA), an HLS framework that synthesizes efficient specialized accelerator modules for individual loops by utilizing coarse-grained pipeline parallelism techniques. The combination of coarse-grained pipelining and exploitation of parallelism within each pipelined stage allows CGPA to design efficient accelerators for C/C++ programs with irregular memory accesses and complex control flows. Compared to the unparallelized version, CGPA shows speedups of 3.0x–3.8x for 5 kernels from programs in different domains.

Acknowledgments

We thank the entire Liberty Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments. This research was funded in part by National Science Foundation grants 0964328 and 1047879 and by DARPA contracts FA8750-10-2-0253 and HR0011-13-C-0005. All opinions, findings, conclusions, and recommendations expressed throughout this work are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

[1] Altera corp. Web site: http://www.altera.com.

[2] Autoesl. http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, 2008.

[4] 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. In *FPGA*, 2011.

[5] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *PPoPP*, 1995.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[7] J. Cong, P. Zhang, and Y. Zou. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. In *DAC*, 2012.

[8] J. Cong and Z. Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. In *DAC*, 2006.

[9] P. Diniz and J. Park. Automatic synthesis of data storage and control structures for FPGA-based computing engines. In *FCCM*, 2000.

[10] B. Draper, W. Nar, W. Bohm, J. Hammes, B. Rinker, C. Ross, M. Chawathe, and J. Bins. Compiling and optimizing image processing algorithms for fpgas. In *CAMP*, 2000.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[12] E. Fiksman, Y. Birk, and O. Mencer. ASC-Based Acceleration in an FPGA with a Processor Core Using Software-Only Skills. In *FCCM*, 2006.

[13] J. Frigo, M. Gokhale, and D. Lavenier. Evaluation of the streams-C C-to-FPGA Compiler: An Applications Perspective. In *FPGA*, 2001.

[14] R. Ghiya and L. J. Hendren. Is it a Tree, DAG, or Cyclic Graph? In *POPL*, 1996.

[15] M. Gokhale and J. Stone. NAPA C: Compiling for a Hybrid RISC/FPGA Architecture. In *FCCM*, 1998.

[16] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented fpga computing in the streams-c high levellanguage. In *FCCM*, 2000.

[17] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, and R. Laufer. Piperench: a coprocessor for streaming multimedia acceleration. In *ISCA*, 1999.

[18] J. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *FCCM*, 1997.

[19] T. B. Jablin. *Automatic Parallelization for GPUs*. PhD thesis, 2013.

[20] N. P. Johnson, T. Oh, A. Zaks, and D. I. August. Fast condensation of the program dependence graph. In *PLDI*, 2013.

[21] O. Kocberber, B. Grot, P. J., F. B., L. K., and R. P. Meet the walkers: Accelerating index traversals for in-memory databases. In *MICRO*, 2013.

[22] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[23] H. H. Lee, Y. Wu, and G. Tyson. Quantifying Instruction-Level Parallelism Limits on an EPIC Architecture. In *ISPASS*, 2000.

[24] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: designing SoC accelerators for memcached. In *ISCA*, 2013.

[25] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, 2004.

[26] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing. In *ISCA*, 2013.

[27] E. Raman. *Parallelization Techniques with Improved Dependence Handling*. PhD thesis, Princeton University, 2009.

[28] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *CGO*, 2008.

[29] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *MICRO*, 1994.

[30] J. Tripp, K. Peterson, C. Ahrens, J. Poznanovic, and M. Gokhale. Trident: an fpga compiler framework for floating-point algorithms. In *FPL*, 2005.

[31] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: Reducing the energy of mature computations. In *ASPLOS*, 2010.

[32] J. Villarreal, A. Park, W. Nar, and R. Halstead. Designing Modular Hardware Accelerators in C with ROCCC 2.0. In *FCCM*, 2010.

[33] M. Weinhardt and W. Luk. Pipeline vectorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):234–248, 2001.

[34] M. Wirthlin and B. Hutchings. A dynamic instruction set computer. In *FCCM*, 1995.

[35] M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *MICRO*, 1996.

[36] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[37] J. Xu, N. Subramanian, A. Alessio, and S. Hauck. Impulse C vs. VHDL for Accelerating Tomographic Reconstruction. In *FCCM*, 2010.

APPENDIX

A. Case Studies

This section shows the applicability of CGPA by describing two interesting cases from the evaluated set of benchmarks.

A.1 K-means

```
for (int i = 0; i < numNodes; ++i ) {                                  R
  int index = findNearestPoint (nodes[i], nFeatures, clusters, nClusters);  P
  if(membership[i] != index)
    delta += 1;
  membership[i] = index;
  new_centers_len[index] += 1;                                         S
  for(int j = 0; j < nFeatures; ++j)
    new_centers[index][j] += nodes[i][j];
}
```

Figure 1: The targeted loop for K-means algorithm, along with the identification of different sections.

For K-means, the CGPA compiler builds a pipelined accelerator by targeting the loop for deciding cluster membership for each point and updating new cluster centers. Figure 1 shows the source code for the loop. This loop first finds the *membership* of each input data point by using the `findNearestPoint` function. This function calculates the Euclidean distance between a data point and the center of each cluster, then returns the *index* of the cluster corresponding to the minimum distance. The returned *index* is used to assign membership to the data point and to update the positions of the corresponding cluster center.

The CGPA framework indicates that the targeted loop can be separated into three unique sections (shown in Figure 1). The induction variable calculation, which determines the index of data points used in an iteration and also the termination of the loop, is identified as a *Replicable Section*. The call to the `findNearestPoint` function can be executed independently for each data point, and thus is identified as a *Parallel Section*. The rest of the loop contains updates to three objects: `membership`, `new_center_len`, and `new_centers`. These updates are executed for each iteration, and cannot be overlapped with the updates from the other iterations. The CGPA framework identifies these updates as belonging to a *Sequential Section*.

In the pipeline transformation step of CGPA, the Parallel Section is deployed as the parallel stage in the pipeline, which then is translated into parallel hardware modules as shown in Figure 2. The Sequential Section is transformed to one hardware worker, which is connected to the parallel workers in the earlier stage via FIFO buffers. The compiler also identifies that the Replicable Section is lightweight, so it is duplicated across all workers. Thus, each worker has its own induction variable calculation. One 4-channel FIFO buffer is generated to hold the index from the workers in the parallel stage. The sequential worker completes its task by fetching index values from the buffers on a round-robin basis. By separating the sequential worker from the main parallel computations in the algorithm, maximum parallelism can be exploited while ensuring correctness of execution.

Some existing HLS tools have targeted inner loop for the K-means kernel. For example, Gokhale et al. propose the use of a systolic array structure for the calculation of Euclidean distances within the `findNearestPoint` function, based on an language extension of C, called Stream-C [16]. However, the CGPA framework differs from this approach in the following ways:

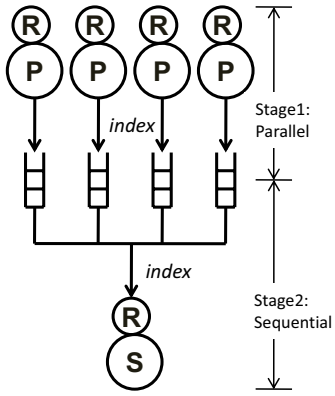

Figure 2: Pipeline generated for K-means by CGPA.

- The results of CGPA are more general (in terms of the number of different clusters and input data points), since CGPA does not assume a fixed number of clusters; and
- CGPA targets the outer loop, which potentially has a higher degree of parallelism, since the number of data points is typically orders of magnitude larger than the number of cluster centers.

CGPA is a coarse-grained pipeline parallelism technique for individual loops, and its transformation technique usually does not change the code structure of inner loops, which can be targeted by existing HLS techniques. In this example, the calculation of the Euclidean distances between the nodes and the centers can still be optimized by applying systolic array structures if the number of clusters is a constant. Thus, CGPA can be seen as complementary to existing work.

A.2 1D Row Gaussian Blur

```
for (int i = 0; i < height; ++i ) {
  float img0 = img[i][0];
  float img1 = img[i][1];
  float img2 = img[i][2];
  float img3 = img[i][3];
  float img4 = img[i][4];
  for (int j = 0; j < width-4; ++j) {                                  R1
    intermediate[i][j] = coef0*img0 + coef1*img1 + coef2*img2
                         +coef3*img3+ coef4*img4;                      P
    img0 = img1;
    img1 = img2;                                                       R2
    img2 = img3;
    img3 = img4;
    img4 = img[i][j+5];                                                R3
  }
}
```

Figure 3: Source code for targeted loop in 1D Gaussian Blur, along with the identification of different sections.

CGPA is able to generate parallel accelerator designs for both the 1D row and column Gaussian Blur kernels in SIFT program. This section only shows the result of the 1D row Gaussian Blur kernel, whose code is shown in Figure 3. For a certain row *i*, the loop has a window of size 5 moving from the left to the right of that row, and calculates a weighted sum reduction of all the image points within the window. In prior work, a series of optimizations, namely scalar replacement and pipeline vectorization [9], have been utilized to

optimize the number of memory accesses for the loop. In our evaluation, these optimizations are applied for the CPU baseline, Legup and CGPA.

The CGPA framework is able to identify four different code sections for the targeted loop. Three of the four sections are identified as Replicable Sections. The first replicable section (labelled R1) performs induction variable calculations, which gives out the column index of the image data. The second replicable section (labelled R2) performs data swaps. The third replicable section (labelled R3) fetches new image data. The fourth section is a parallel section, which performs a weighted sum reduction of each window position. The sum reduction for each position can be performed independently of that for other positions, thus explaining why this section is identified as a parallel section.

CGPA handles the three replicable sections in different ways, according to their features. Since R1 and R2 are lightweight, they are replicated in the workers. R3 has a load instruction, thus it is inserted into a separated sequential pipeline stage. As a result, R1 is replicated in both the sequential stage and parallel stage, because it performs all the induction variable calculations. R2 is only replicated in the parallel stage, because its result is only used in the parallel stage. For each iteration, R3 fetches image data and broadcasts it to all four shift register chains (R2) in the second stage. A final pipeline partition and transformation generated by CGPA is shown in Figure 4.

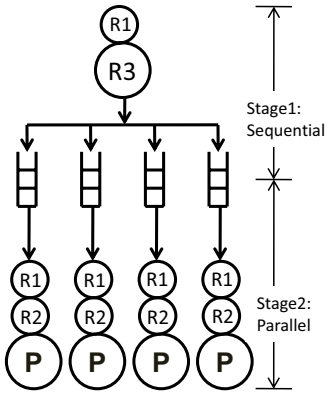

Figure 4: Pipeline generated for 1D Row Gaussian Blur by CGPA.

Compared to the pipeline vectorization technique which can only generate one reduction window [9], CGPA builds four parallel reduction windows to generate four intermediate data concurrently. This is not just a simple duplication of reduction windows that allows them to work independently (like the results generated by inserting R3 in a parallel stage), and experimental results show that the decoupled pipelining technique improves the performance by 15% and reduces the energy cost by 14%. Furthermore, the *map-reduce* style circuit in [26] can also be applied in the Parallel stage to improve the sum reduction performance. Again, that is complementary to CGPA.

B. Discussions

This section discusses the scalability of CGPA and the potential improvements that could be achieved by combining it with prior work.

B.1 Scalability

Due to the limitations of the experimental platform, this paper only shows cases with a maximum of 4 parallel workers in the parallel stage. The degree of parallelism that can be potentially exploited by CGPA is larger than this number for all the benchmarks. In the ideal case, the scalability of CGPA depends on three issues:

Workloads of the Sequential Stage: Increasing the workload on the sequential stages has two effects: (1) it may stall the parallel stage through the FIFO buffers, and (2) it may limit the overall speedup in accordance with Amdahl's law. The pipeline partition algorithm in CGPA tries to find maximum parallel section of the loop body, thus reducing the workloads of the sequential stages.

Workloads of the Replicable Sections in the Parallel Stage: If the number of workers in the parallel stage is increased, there is an increase in the chance that execution enters the loop body which contains only the replicable section. This could result in higher overheads for the whole accelerator system. Thus, it is important for efficiency and scalability to decide where the replicable section must be inserted. In CGPA, the partition algorithm can intelligently calculate the pipeline balance and decide which replicable sections should be inserted in the parallel stage.

Memory system support: As shown in the experiments, CGPA tries to reduce memory access by reusing input data from a sequential stage. However, in CGPA, since each worker in the parallel stage has its own memory ports, the overhead of building shared memory system becomes large if the number of parallel workers increases To solve this problem, some existing memory optimizations, such as private cache and memory partition techniques can be applied. CGPA's pipeline partition design enforces an assignment of aliasing memory instructions to the same stage (by creating SCCs), and this indicates that there are no data access conflicts from different stages and this helps the application of existing memory optimizations.

B.2 Potential Improvements

The CGPA compiler focuses on outer loops and relies on the adaptive backend to optimize inner loops which are suitable for targeting by traditional methods. Since CGPA is a coarse-grained parallelism technique, it usually keeps the code structure of inner loops, which enables additional optimizations using the existing research results such as prefetching and other loop-level parallelism techniques. Normally, these additional techniques can be applied to each stages seperately. Additionally, the bandwidth of the memory system can be increased by synchronizing the memory accesses from parallel stage workers [19]. We leave these potential improvements as future work.