From Software Threads to Parallel Hardware in High-Level Synthesis for FPGAs

Jongsok Choi, Stephen Brown, and Jason Anderson
ECE Department, University of Toronto, Toronto, ON, Canada
legup@eecg.toronto.edu

***Abstract*—We describe the support within high-level *hardware* synthesis (HLS) for two standard *software* parallelization paradigms: Pthreads and OpenMP. Parallel code segments, as specified in the software, are automatically synthesized by our HLS tool into parallel-operating hardware sub-circuits. Both data parallelism and task-level parallelism are supported, as is the combined use of *both* Pthreads and OpenMP. Moreover, our work also provides automated synthesis for commonly occurring synchronization constructs within the Pthreads/OpenMP library: mutual exclusion (mutex) and barriers. Essentially, our framework allows a software engineer to specify parallelism to an HLS tool using methodologies they are likely to be familiar with. An experimental study considers a variety of parallelization scenarios, including demonstrated speedups of up to 12.9× in circuit wall-clock time for the 16-thread case and area-delay product as low as 12% (∼8× improvement) when using 4 pipelined hardware threads.**

I. Introduction

The inherent parallelism of FPGA hardware has been applied to provide orders of magnitude speedup vs. software for implementing computations [5]. FPGA hardware is difficult to design however, requiring the use of hardware description languages (HDLs), such as VHDL or Verilog. Moreover, hardware expertise is comparatively rare vs. software expertise [26]. To deal with these trends, high-level synthesis (HLS) continues to gain traction as a design methodology for FPGAs, both to ease hardware development, and ultimately, to make the performance advantages of FPGA hardware accessible to those with software skills. There is a gap, however, between HLS-generated hardware and human-designed hardware, partly due the inability of HLS to fully exploit the parallelism available in the target FPGA fabric. We addresses this challenge by presenting a methodology through which an engineer may use software techniques to specify parallelism to an HLS tool, and providing an HLS tool that implements the specified parallelism in a hardware circuit.

Software compilers have a limited ability to infer parallelism automatically, and as such, it is normally incumbent on the programmer to specify parallelism within the code. Common parallelization approaches include using parallel programming languages such as OpenCL [19] or CUDA [22], or the use of libraries like Pthreads [21] and OpenMP [23]. A question that naturally arises then, is how does one specify parallelism to an HLS tool? There appears to be no single common approach among current HLS tool offerings. To specify fine-grained loop-level parallelism, such as loop pipelining or loop unrolling, some tools require the use of vendor-specific directives (pragmas) within the source code [27], while other tools use constraint side files [6]. Coarse-grained parallelism is often realized by using an HLS tool to synthesize a single hardware core, and then manually instantiating multiple instances of the core in structural HDL – an approach which requires knowledge of hardware design.

In this paper, we provide HLS support for standard parallel programming methodologies that software engineers are already likely familiar with. In particular, we propose using Pthreads and OpenMP for the specification of parallelism to an HLS tool. We provide an HLS framework wherein the parallelism described in the software code is automatically synthesized into parallel hardware accelerators that perform the corresponding computations concurrently. Writing deterministic parallel software often requires the use of synchronization constructs that, for example, manage which threads may execute a given code segment at any given moment. Recognizing this, we also provide HLS support for two key thread synchronization constructs in the Pthreads/OpenMP library: mutexes and barriers. Furthermore, to meet memory bandwidth constraints, our tool automatically synthesizes arbiters among parallel accelerators that share memory. Our Pthreads and OpenMP support in HLS has been implemented within the LegUp open-source high-level synthesis framework developed at the University of Toronto [6].

In an experimental study, we demonstrate the capabilities of our HLS tool across a broad range of parallelization scenarios applied over a suite of benchmark circuits, including the use of Pthreads in isolation, combined use of both Pthreads and OpenMP, as well as Pthreads in conjunction with loop pipelining. Results show that the proposed techniques can provide up to 12.9× speedup in wall-clock time with 16 hardware threads, and show an area-delay product as low as 12% (over 8× improvement) when using 4 pipelined hardware threads.

The contributions of this work are: 1) Automatic generation of parallel hardware using *both* Pthreads and OpenMP. We believe this to be the first work of its kind, and not found in prior literature. 2) Automatically generating "nested" parallel hardware using Pthreads and OpenMP, or Pthreads and loop pipelining. 3) Quantitatively analyzing the three parallelization techniques, Pthreads, OpenMP, and loop pipelining, in isolation as well as in tandem. 4) Providing an *open-source* framework which can be freely used by the research community.

Our work represents a key step towards improving the performance of hardware that can be produced by an engineer who solely possesses software skills. The remainder of this paper is organized as follows: Section II presents related work. Section III provides an overview of parallel programming in software using Pthreads and OpenMP. Section IV describes how these parallel software threads are compiled to concurrent hardware accelerators. Section V provides an overview of the targeted system. An experimental study is described in Section VI and Section VII concludes this work.

II. Related Work

Several prior works consider the use of OpenMP for FPGAs. The work in [11] implements an extension to OpenMP so that computations can be off-loaded to an FPGA, although it was not in the context of high-level synthesis and required that the FPGA hardware be designed manually using HDL. The work in [17] describes a framework that generates Handel-C and VHDL from programs that use OpenMP, where the generated code is implemented on an FPGA. Although the

978-1-4799-2198-0/13/$31.00 ©2013 IEEE

work bears some similarity to our own, it has significant limitations, namely, it only supports the *integer* data type, the target hardware has no memory subsystem, the hardware FSM allows only one statement in each state, and there is no support for nested parallelism. Our framework does not have such limitations. The work in [7] describes a source-to-source translator that accepts a `C` program that uses OpenMP as input, and generates source files to be passed to an HLS tool. The authors do not provide the HLS ability themselves, but instead use a commercial tool – Impulse CoDeveloper [18].

Concerning Pthreads, [25] describes a framework which employs Pthreads to generate hardware accelerators at *runtime* using on-chip CAD tools. This framework requires an OS running on an embedded processor (ARM11) to manage the scheduling and execution of threads. In our work, no OS is required and the thread support is completely integrated into the HLS tool itself, permitting auto-generation of a complete system, synthesizeable by commercial RTL synthesis tools. HybridThreads (hthreads) is a real-time embedded operating system which allows programmers to run threads simultaneously on a CPU and on an FPGA [15, 9, 10]. The authors provide an hthreads API, which, while similar to the Pthreads API, is nonetheless a non-standard approach to parallelization.

It is also worthwhile to comment on the parallelization capabilities of the HLS tools offered by the two main commercial FPGA vendors. Vivado [27] is Xilinx's HLS tool, which provides a rich set of features such as pipelining, memory partitioning/restructuring, arbitrary precision types, as well as supporting other user-specified pragmas to control the generated hardware. Hardware knowledge is needed to fine-tune the hardware using pragmas and there is currently no support for standard software APIs to specify parallel execution. Altera's OpenCL compiler [3] permits the compilation of OpenCL kernels to FPGA hardware. Parallelism is explicitly specified by the programmer in OpenCL, which is compiled to pipelined hardware units. Related to the Altera effort, [8] provides a source-to-source compiler which translates CUDA code and produces annotated C code that can be input into another HLS tool, AutoPilot [14].

To our knowledge, there is no prior work that offers an open-source HLS tool with support for parallelism expressed using the Pthreads or OpenMP standards.

III. Parallel Programming in Software

This section briefly describes Pthreads and OpenMP and illustrates how they can be used to express parallelism. We focus on the most widely-used aspects of the two parallelization approaches, which are the same aspects we have selected for automated synthesis to hardware. Consider the following code snippet that uses Pthreads:

```
for (i=0; i<4; i++) { //fork threads
  pthread_create(&threads[i], NULL, add, &data[i]);
}

for (i=0; i<4; i++) { //join threads
  pthread_join(threads[i], NULL);
}
```

The `pthread_create` function invokes a new thread to execute a function, whose name is passed as an argument. Here, four parallel threads are invoked, each of which executes the `add` function. As its arguments, `pthread_create` takes a pointer to a Pthread thread variable, a pointer to a Pthread attribute variable (`NULL` in this case), the name of the function to execute on the thread, and a pointer to an argument variable (`data[i]`). Using `pthread_create`, one can also execute different functions in parallel, by passing different functions as the third argument.

The `pthread_join` function waits for a specified thread to terminate. In the above example, it is used to wait until all four threads have finished executing the `add` function. As its arguments, it accepts a thread variable, and a pointer to a return value variable (`NULL` in this case).

In parallel programming, synchronization mechanisms are used to prevent race conditions and ensure correctness of data, with locks and barriers being the most commonly used mechanisms. A lock is used to specify mutual exclusion to a critical section of code, ensuring that at most one thread can execute the code at a given time. With Pthreads, locks are specified with the `pthread_mutex_lock` and the `pthread_mutex_unlock` functions. Both functions take a pointer to a special *mutex* variable as their only argument. Multiple locks can be specified by using different mutex variables. A barrier is used to synchronize threads at a specific point in a program. When a barrier is used in a program, all threads must stop at the barrier point and wait until all other threads have reached the barrier. In Pthreads, `pthread_barrier_init` is used to initialize the barrier with the number of threads that must wait at the barrier. The `pthread_barrier_wait` function synchronizes participating threads at the barrier which blocks the threads until the required number of threads have reached the barrier.

Turning now to OpenMP, consider the following code segment:

```
#pragma omp parallel for num_threads(2) private(i)
for (i = 0; i < SIZE; i++) {
  output[i] = A_array[i]*B_array[i];
}
```

The loop performs a dot product on two arrays, `A_array` and `B_array`. To parallelize this loop using OpenMP, one simply puts an OpenMP pragma before the loop, as shown in the example. The OpenMP pragma, `#pragma omp parallel for`, is used to parallelize a `for` loop. The pragma uses a number of clauses. The `num_threads` clause sets the number of threads to use in the parallel execution of the `for` loop. The `private` clause declares the variables in its list to be private to each thread. In the above example, two threads will execute the loop in parallel, with one thread handling the first half the array, and the other handling the second half of the array. When `#pragma omp parallel for` is used on a `for` loop, the `gcc` compiler outlines the body of the loop to a function, and inserts a call to a library function called `GOMP_parallel_start`, marking the start of a parallel section, and then forks threads to execute the outlined function. This is immediately followed by a call to another library function, `GOMP_parallel_end`, which marks the end of a parallel section and makes the threads wait for all other threads to finish execution. Note that the parallel pragma in OpenMP is blocking – all threads executing the parallel section need to finish before the program execution continues. `#pragma omp parallel` can also be used to parallelize a section of code which is not in a loop.

As illustrated, OpenMP provides a simple and a high-level approach for parallelization. With a single pragma, the user is able to parallelize a section of code without complicated code changes. On the other hand, Pthreads requires explicit forks and joins of threads. Pthreads generally requires more work from the programmer but it also gives more fine-grained control.

IV. Parallel Threads to Parallel Hardware

Prior to this work, LegUp HLS was only able to exploit instruction-level parallelism, and loop-level parallelism via

loop pipelining. The current work greatly expands the extent to which a user may specify parallelism to the HLS tool. The approach we take is to automatically instantiate parallel hardware for parallel threads. That is, each software thread is mapped automatically into a hardware accelerator[1]. The remaining (sequential) portions of the program are executed in software on a MIPS *soft* processor. The MIPS processor invokes parallel accelerators and retrieves their return values by using *wrapper functions*, which replace the original software versions of the parallel code.

A. Wrapper Functions for Parallel Accelerators

The MIPS soft processor communicates with hardware accelerators over memory-mapped addresses defined in wrapper functions. As described above, Pthreads and OpenMP use the `pthread_create/join` and `GOMP_parallel_start/end` functions to create and manipulate threads. When these functions are called in a program, our framework automatically generates wrapper functions. The wrapper functions are used by the MIPS processor to send function arguments to accelerators, start the accelerators, poll to check if the accelerators are done, and retrieve any return values. Two types of wrapper functions are created by our tool: *calling* wrappers are generated to replace software code that launches threads, namely, `pthread_create` and `GOMP_parallel_start`. *Polling* wrappers are generated to replace code that joins threads, namely, `pthread_join` and `GOMP_parallel_end`. Example wrapper functions for OpenMP are shown below.

```
1: #define fct_DATA          (volatile int*)0xf000000
2: #define fct_STATUS        (volatile int*)0xf000008
3: #define fct_ARG1          (volatile int*)0xf00000c
4: #define fct_THREADID      (volatile int*)0xf000010
5: #define fct_THREADMUTEXID (volatile int*)0xf000014
6:
7: void legup_ompcall_fct(char* _omp_data_i) {
8:   int i;
9:   for (i=0; i<4; i++) {
10:    *(fct_ARG1+i*8) = (volatile int)_omp_data_i;
11:    *(fct_THREADID+i*8) = i;
12:    *(fct_THREADMUTEXID+i*8) =
                        (volatile int)fct_DATA+i*8;
13:    *(fct_STATUS+i*8) = 1;
14:  }
15:}
16:
17:void legup_omppoll_fct() {
18:  int i;
19:  for (i=0; i<4; i++) {
20:    while (*(fct_STATUS+i*8) == 0){}
21:  }
22:}
```

Prior to the wrapper functions, the address map of the first hardware accelerator is defined using `#define` statements (lines 1-5). Only addresses associated with the first accelerator are defined per parallel section (for compactness and readability of code), and pointer arithmetic is performed to calculate the addresses associated with the remaining accelerators belonging to the same parallel section. The pointers are used to send arguments to accelerators (function arguments, thread ID, and thread mutex ID, as shown on lines 10-12), start accelerators (line 13), and check if accelerators are done (line 20). The thread ID is a unique number within a parallel section that is assigned to each thread to determine the portion of work the thread is responsible for in the particular parallel section. The thread *mutex* ID is a unique number across all parallel sections that is assigned to each thread that accesses a mutex. This thread mutex ID is used to lock/unlock a mutex. In the above example, observe that the calling wrapper invokes four parallel OpenMP accelerators. After executing the calling wrapper, the processor proceeds to call the polling wrapper (lines 17-22). The polling wrapper queries each accelerator and checks the `STATUS` register in a `while` loop (line 20). An accelerator returns a '1' when it is done, which terminates the `while` loop.

With Pthreads, the generated wrappers are different from the OpenMP wrappers, as thread creation with Pthreads is non-blocking, and it involves a thread variable. An example set of calling and polling wrapper functions for Pthreads are:

```
1: #define add_DATA          (volatile int*)0xf000000
2: #define add_STATUS        (volatile int*)0xf000008
3: #define add_ARG1          (volatile int*)0xf00000c
4: #define add_THREADMUTEXID (volatile int*)0xf000010
5:
6: int legup_count_add=0;
7: void legup_pthreadcall_add(char *threadarg,
                              pthread_t *threadVar) {
8:   int legup_accelID_add=legup_count_add++;
9:   int LEGUP_ADDR_OFFSET=legup_accelID_add*8;
10:  *threadVar=(int)(add_STATUS+LEGUP_ADDR_OFFSET);
11:  *(add_ARG1+LEGUP_ADDR_OFFSET)=
         (volatile int)threadarg;
12:  *(add_THREADMUTEXID+LEGUP_ADDR_OFFSET)=
         (volatile int)(add_DATA+LEGUP_ADDR_OFFSET);
13:  *(add_STATUS+LEGUP_ADDR_OFFSET)=1;
14:}
15:
16:char *legup_pthreadpoll(pthread_t threadVar) {
17:  volatile int* STATUS=(volatile int*)(threadVar);
18:  volatile int* DATA=STATUS-2;
19:  while (*STATUS == 0){}
20:  return (char*)*DATA;
21:}
```

For each different `C` function intended for execution in a thread, a new calling wrapper (lines 1-14) is created, whereas, only one polling wrapper (lines 16-21) is created for all `pthread_join` calls. The reason for this is that the `pthread_join` function uses the thread variable to determine, at runtime, which thread to join. Therefore, the polling wrapper must also determine at runtime which hardware accelerator to "join" using the thread variable.

The calling wrapper receives the index of the accelerator (line 8), for when the same function is executed on multiple threads, and it calculates the memory offset of the accelerator based on the index (line 9). This offset is used to calculate the memory-mapped address of the accelerator, which is saved into the Pthread variable (line 10), to be used later in the polling wrapper. Since the thread variable already exists as an argument to `pthread_create`, it can be conveniently "re-purposed" automatically by our system without requiring any code changes by the user. This is "legal" as the thread variable itself does not have any other uses. Once the address is saved, function arguments and the thread *mutex* ID are sent to the hardware accelerator (lines 11-12) and the start signal is given (line 13).

At run-time, the polling wrapper determines which accelerator it needs to communicate with by loading its `STATUS` address from the thread variable (line 17). Once the address is retrieved, the polling wrapper gets the `DATA` address (line 18, `STATUS` address is always offset by 2 words from the `DATA` address, as shown on lines 1-2), then proceeds to poll the `STATUS` register to check if the accelerator is done (line 19). It then retrieves the return value if necessary (line 20).

[1]Note that the number of software threads needs to be fixed at compile time as LegUp currently does not support dynamic creation of hardware accelerators at runtime.

B. Synchronization Mechanisms

We now describe how we support thread synchronization mechanisms (locks and barriers) in HLS-generated hardware. In our framework, a lock is created in hardware by using a special *hardware mutex core*, which is also mapped to an address. An instance of the core is instantiated for each mutex variable used in the parallel program, and a round-robin arbiter is instantiated for each core to ensure atomic access. The core itself is quite simple: it contains a register that holds the unique thread ID of the hardware accelerator that holds the lock and has a flag to indicate its state (locked/unlocked).

The operation of a lock is as follows. Our framework automatically replaces all calls to `pthread_mutex_lock` and `pthread_mutex_unlock` with calls to our own lock/unlock functions. Our lock/unlock functions perform memory-mapped reads and writes to acquire and release a lock. Both functions have two arguments: the thread ID, and the mutex index. The mutex index is used to differentiate between multiple locks – requests which must be steered to different instances of mutex cores. Our replacement lock/unlock functions can be executed both in software on the MIPS processor and in hardware on accelerators.

When a processing element (the MIPS or a hardware accelerator) calls the lock function, it first tries to write its thread ID to the mutex core corresponding to the mutex index. If the mutex is free, the write is successful and the thread ID is stored. If the mutex is already locked, the mutex core retains the previously stored thread ID. After the write, the processing element reads from the mutex to check if the stored thread ID matches its own ID. If there is a match, this indicates that the processing element has acquired the lock and is free to enter the critical section. If the processing element fails to get the lock, it repeats the locking procedure until it gets the lock (a behaviour akin to "spin locks").

In our unlock function, the processing element again writes to the mutex core with its thread ID. If the mutex is locked with the matching thread ID, this unlocks the mutex.

OpenMP provides mutual exclusion capability with two pragmas: `#pragma omp atomic` and `#pragma omp critical`. The `atomic` pragma permits the specification of a single-statement critical section, whereas a multiple-statement critical section can be specified with the `critical` pragma. Both of these pragmas are supported in our tool by using a similar approach to Pthread locks.

Pthread barrier functions, `pthread_barrier_init` and `pthread_barrier_wait`, are likewise automatically replaced with our own barrier init/wait functions, and a hardware barrier core is created. The barrier core contains a register, which is initialized when the barrier init function is called to store the number of threads to wait at the barrier. The barrier core also contains a counter, which is incremented each time a processing element reaches the barrier. When the barrier wait function is called, the processing element first writes to the hardware barrier core to increment its counter, then it keeps polling on the barrier core, which returns a one until enough threads have reached the barrier. When the counter equals the number of threads, the barrier core returns a zero, at which point the processing elements can continue to execute. The barrier core also resets its counter to zero, so that it can be used again for the same barrier object.

Table I shows a list of Pthreads and OpenMP library functions which are currently supported in our framework. In addition to those listed in the table, OpenMP clauses to set the number of threads (`num_threads`), the scopes of variables (`public`, `private`, `firstprivate`, `lastprivate`) and the division of work among threads (`static` scheduling of any chunk size) are also supported. Note that all of the original calls to OpenMP/Pthreads functions are automatically replaced with corresponding functions in our framework, requiring no manual code changes by the user. Meaning that, the input `C` program with calls to the Pthreads/OpenMP API can be compiled to a hybrid processor/accelerator system *as is*.

TABLE I. SUPPORTED PTHREADS FUNCTIONS/OPENMP PRAGMAS.

Pthreads Functions	**Description**
pthread_create(..)	Invoke thread
pthread_join(..)	Wait for thread to finish
pthread_exit(..)	Exit from thread, can be used to return data
pthread_mutex_lock(..)	Lock mutex
pthread_mutex_unlock(..)	Unlock mutex
pthread_barrier_init(..)	Initialize barrier
pthread_barrier_wait(..)	Synchronize on barrier object
OpenMP Pragmas	**Description**
omp parallel	Parallel section
omp parallel for	Parallel for loop
omp master	Parallel section executed by master thread only
omp critical	Critical section
omp atomic	Atomic section
reduction(operation: var)	Reduce a var with operation
OpenMP Functions	**Description**
omp_get_num_threads()	Get number of threads
omp_get_thread_num()	Get thread ID

C. Automatic System Generation Flow

An overview of our automatic system generation flow in shown in Fig. 1. Our flow is integrated with the LegUp HLS framework, which itself is implemented within the LLVM compiler framework [20]. The input is a `C` program containing calls to the Pthreads and/or OpenMP API. The program is first compiled using `llvm-gcc`[2] which translates the program into the LLVM IR (intermediate representation). LegUp implements HLS as a set of back-end compiler passes which operate directly on the optimized LLVM IR. A detailed description of LegUp's algorithms is outside the scope of this paper, and the interested reader is directed to [6].

We wrote an LLVM pass, SW Pass, which replaces all of the Pthreads/OpenMP functions with our own functions as described previously (left side of Fig. 1). It also removes the functions to be implemented as hardware accelerators from the software portion of the program. The modified software IR is compiled to a MIPS executable using the LLVM/MIPS toolchain. The SW pass also creates a `Tcl` file, which will be used to control Altera's SOPC Builder to automatically generate the system (described in the next section).

The HW Pass (right side of Fig. 1) operates on the functions intended to be realized as hardware accelerators, replacing all calls to Pthreads/OpenMP functions (e.g. locks, barriers) and removing all software functions that are executed on the processor. This modified HW IR is passed to LegUp HLS to produce synthesizeable Verilog. Each thread in software becomes its own accelerator in hardware. LegUp HLS algorithms have also been modified to support nested parallel hardware (described in the next section).

Once both software and hardware components have been generated, Altera's SOPC Builder uses the `Tcl` file to generate the complete system. This generated system can either be simulated with ModelSim (a testbench with input vectors is also automatically generated), or compiled to bitstream using Altera's Quartus II. This entire flow is automated so that the user only has to run a *single* Makefile target.

V. SYSTEM ARCHITECTURE

The system architecture is shown in Fig. 2 and we target the Altera DE4 board with a Stratix IV 40nm FPGA. The

[2]Clang, the native front-end for LLVM, does not have OpenMP support.

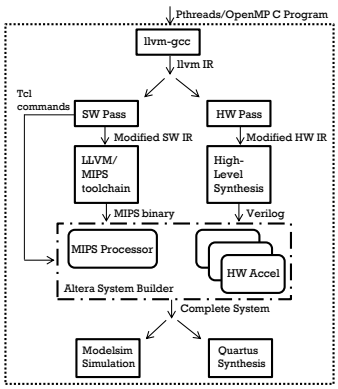

Fig. 1. Automatic system generation flow.

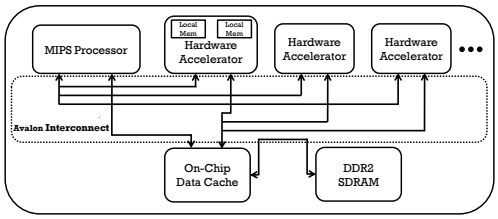

Fig. 2. Default system architecture.

system comprises the MIPS soft processor, hardware accelerators, mutex/barrier cores (if used in the program), on-chip cache, as well as off-chip DDR2 memory. An accelerator may also have local memories for local data not shared with the processor or other accelerators. The local memories are implemented in on-chip Block RAMs, instantiated within a hardware accelerator. Shared data, on the other hand, is stored in off-chip DDR2 memory (on the DE4 board), which can be fetched into the on-chip cache. The components of the system communicate via the Avalon Interconnect, which is generated by Altera's SOPC Builder [1]. Avalon is a point-to-point network, which allows multiple independent transfers to occur simultaneously, via memory-mapped addresses. When multiple components are connected to a single component, such as the on-chip data cache, a round-robin arbiter is created by SOPC Builder to arbitrate among simultaneous accesses. For memory-intensive applications, the default dual-port cache can be replaced with a multi-ported cache (controlled by a `Tcl` parameter), which allows multiple accelerators to access the cache concurrently [13].

A. Parallel Accelerator Architecture

Parallel accelerators (corresponding to parallel threads) are connected to the system as shown in Fig. 2. Our framework permits multiple calls to `pthread_create/join` with the same and/or different functions to be executed on the threads, as well as multiple sections of code parallelized with the OpenMP parallel pragmas.

In this work, we also allow *nested parallelism* – threads forking threads. Consider the case of there being multiple functions executed in parallel with Pthreads – a first level of parallelism. These functions could have one or more loops, some of which could be parallelized with OpenMP – a second level of parallelism. Currently, we only permit up to two levels of parallelism for automated hardware synthesis, with Pthreads being the first level and OpenMP being the second. OpenMP can also be used as the first level of parallelism, though we do not consider this case in our experimental study. We refer to the second-level accelerators as *internal accelerators*. The internal

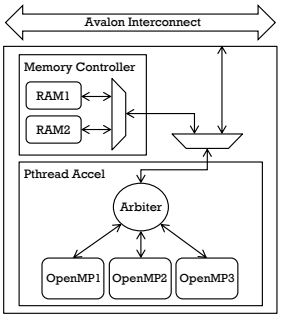

Fig. 3. Detailed accelerator architecture.

accelerators are created inside their corresponding first-level accelerators. Internal accelerators can access the local variables of their first-level accelerators, which are created in local RAMs, as well as global/stack variables stored in the shared memory space.

Fig. 3 shows the architecture of a hardware accelerator with internal accelerators. Internal accelerators, corresponding to OpenMP threads, are instantiated multiple times, each of which executes a portion of work in parallel. The internal accelerators are invoked simultaneously by an FSM, which also controls the execution of the first-level accelerator. As parallelization with OpenMP is blocking, after invoking the internal accelerators, the FSM waits until all of them have finished execution before continuing on to the next state (adhering to the semantics of the `GOMP_parallel_start/end`). Note that wrapper functions are not generated for internal accelerators, because they are invoked by the FSM (not the MIPS).

Since internal accelerators can access memory, and because they execute in parallel, a round-robin arbiter is created inside the first-level accelerator to arbitrate among internal accelerators as they access memory. The round-robin arbiter has a single-cycle latency to grant an access. When an internal accelerator makes a memory request, if it is not granted access by the arbiter, it stalls. Similarly, if an internal accelerator accesses shared memory and experiences a cache miss, it has to wait until the data is fetched from off-chip memory; however, other internal accelerators which are not accessing memory are free to execute. Hence, the internal accelerators do not execute in lockstep but work independently of one another. Although not shown in the figure, additional control logic is created to steer data from memory accesses back to internal accelerators.

VI. Experimental Study

We study the performance and area of several different hardware configurations, each of which corresponds to a different parallelization scenario. The baseline configuration is the default LegUp hybrid processor/accelerator system, with no accelerators that operate in parallel – the processor and accelerators operate sequentially and the processor is stalled while an accelerator does its work.

The parallel configurations fall into 3 classes: 1) a single-level of parallelization using Pthreads, 2) Pthreads combined with loop pipelining – a form of nested parallelism, and 3) Pthreads combined with OpenMP – nested parallelism as described in the previous section. For class #1, the Pthreads may execute the same function or different functions, depending on the benchmark (see below). When executing the same function, each accelerator performs a portion of the total work. Configurations in classes #2 and #3 can be used when the first-level threads contain loops – such loops can be parallelized by loop pipelining or OpenMP pragmas. Note that the current loop pipelining capabilities of the LegUp HLS tool are limited

and can only be applied to loops with bodies that contain no function calls or branches. With the OpenMP support, however, loops with branches and function calls can be parallelized.

For class #3 (Pthreads combined with OpenMP), we experiment with various numbers of Pthreads and OpenMP internal accelerators, for a total of 8 different configurations. The largest configuration has 30 Pthreads with 4 OpenMP internal accelerators, which means that there are essentially a total of 120 accelerators. Note that it does not necessarily mean that all 120 accelerators are *identical*, as the 4 OpenMP accelerators only parallelize the *loop* inside a Pthread function, and there can be other operations done outside the loop. We label architecture configurations as follows: `S` denotes the sequential baseline case, `4L1` denotes the 4 first-level Pthread accelerators architecture, `4L1-P` denotes the 4 first-level Pthread accelerators with loop pipelining, and `nL1-mL2` denotes the architecture with `n` first-level Pthread accelerators with `m` second-level OpenMP accelerators.

A. Benchmarks

We use 7 benchmarks, each of which includes built-in inputs and golden outputs, with the computed result checked against the golden output at the end of the program to verify correctness. The inputs, golden outputs, and the computed results are held in global variables and stored in the shared memory space (off-chip DDR2 SDRAM). Even though using on-chip memory yields better speedup results, off-chip memory is used to model real world applications with big data sets. The benchmarks are:

- Black-Scholes: performs options pricing via a Monte Carlo approach. Computations are done in fixed-point.
- MCML: simulates light propagation from a point source in an infinite medium with isotropic scattering. The benchmark has been adopted from the Oregon Medical Laser Centre [24] with the computations done in fixed-point.
- Mandelbrot: an iterative mathematical benchmark which generates a fractal image.
- Line of Sight: uses the Bresenham's line algorithm [4] to determine whether each pixel in a 2-dimensional grid is visible from the source.
- Division: divides a set of integers in an array by another set of integers.
- Hash: uses four different integer hashing algorithms to hash a set of numbers, and compares the number of collisions caused by the four different hashes.
- dfsin: adopted from the CHStone benchmark suite [16], it implements a double-precision floating-point sine function using 64-bit integers.

We synthesized each benchmark into each of the parallel architecture configurations and simulated the synthesized circuit using ModelSim to extract the total number of execution cycles and verify correctness. Note that we used an accurate Altera-provided simulation model of the off-chip DDR2 SDRAM on the DE4 board. Following the cycle-accurate simulation, each benchmark was synthesized to the Altera Stratix IV (EP4SGX530KH40C2) with Quartus II (ver. 11.1SP2) to obtain area and critical path delay ($Fmax$) numbers. Execution time for each benchmark is computed as the product of execution cycles and post-routed clock period.

B. Results

The performance results for all benchmarks and all architectures are presented in Tables II and III, and the area results are shown in Table IV. For those benchmarks where loop pipelining could not be used, or those which could not be parallelized with more accelerators (due to resource limits on the Stratix IV or due to the nature of the benchmark), the results are shown as "N/A".

Tables II and III show the number of execution cycles it takes to execute each benchmark, the $Fmax$, as well as the wall-clock time (in μs) based on the $Fmax$ and clock cycle results. The speedups and ratios of each parallel architecture relative to the sequential case are also shown. As expected, performance generally improves as the degree of parallelism is increased, up to a certain point. For most benchmarks, the total number of execution cycles decreases as more parallel accelerators are used. For Black-Scholes, MCML, and Mandelbrot, which are computationally intensive rather than memory intensive, the number of clock cycles scales well with the number of accelerators, especially up to the architecture 4L1-4L2 (16 accelerators). For 4L1-4L2, Black-Scholes, MCML, and Mandelbrot show 15.2, 15, and 13.6$\times$ speedup, respectively. For other benchmarks, which are more memory intensive, such as Line of Sight, Division, and Hash, the performance improvement is less. For the Line of Sight and Division benchmarks, a 4-ported cache [13] was used, which allows higher memory bandwidth, though it consumes more resources than the default cache, and hence it is not used for other benchmarks. For the Hash benchmark, locks are used to prevent race conditions between the internal accelerators. With more internal accelerators, the contention to access the mutex core increases, and thus the execution cycles actually increase as the number of internal accelerators is increased from three to four. Similarly for other benchmarks, as the total number of accelerators is excessively increased (up to 120 accelerators in Line of Sight!), the reduction in clock cycles is either small, or the number of clock cycles even increases. With too many accelerators, the work assigned to each accelerator becomes smaller yet the memory contention increases, hurting performance.

The $Fmax$ of the systems is also affected as the degree of parallelism is varied. Overall, $Fmax$ is negatively impacted with more accelerators, mainly due to the arbitration and the stall logic, needed to manage memory contention. Some benchmarks, such as dfsin, Black-Scholes, and MCML, exhibit more rapid reductions in $Fmax$ than others. We believe that, as the utilization of the Stratix IV becomes close to full, the Quartus II synthesis tool has more difficulty optimizing the implementation. For example, for the 4L1-4L2 architecture, dfsin showed 96% logic utilization, and Black-Scholes and MCML showed 85% logic utilization and used 100% of the Stratix IV DSP blocks. However, for the other benchmarks, the $Fmax$ reduction for 4L1-4L2 is $\sim$10% when compared to the baseline.

Fig. 4 shows the geometric mean speedup (in wall-clock time) of the different architectures normalized to the baseline case. Since not all parallelization configurations could be used for all benchmarks, multiple lines are plotted, with each line showing the geomean speedup for a subset of circuits in which the particular configuration could be used[3]. The legend shows which benchmarks are included for each line on the graph. The geometric mean across *all* benchmarks (first line of the legend) shows that the best speedup of 7.6$\times$ is observed with the 4L1-4L2 architecture. The 4L1-P configuration is not included in this case, since loop pipelining could not be applied in all benchmarks.

For the benchmarks where loop pipelining could be used (Division/Mandelbrot/Hash), 4L1-P shows 6.17$\times$ speedup over baseline, and 4L1-4L2 still shows the best result with 7.51$\times$ speedup. There are cases where loop pipelining can perform

[3]If we had used a single line, each data point would represent the average for potentially different sets of circuits.

far better, however. For instance, for the Division benchmark, Table III shows that 4L1-P outperforms all other architectures, even the 20L1-4L2 architecture which has 80 accelerators. This is because a 32-bit division takes 32 cycles in LegUp, using Altera's divider core pipelined to achieve the highest-possible $Fmax$. Since the divider itself is pipelined, it can accept a new input every clock cycle, which is very well suited to loop pipelining. With 4 Pthread accelerators, each of which has only one hardware instance of the loop body, this 4L1-P architecture shows 12.5× speedup over the baseline architecture for the Division benchmark. The biggest speedup in Fig. 4 is 12.9× with the 12L1-4L2 architecture for three benchmarks. Mandelbrot shows the largest single benchmark speedup with 17.2× with the 12L1-4L2 architecture, and 16.6× with the 8L1-4L2 architecture. Overall, as the number of accelerators is increased excessively, the geomean speedups decrease due to reductions in $Fmax$ and diminishing returns in clock cycle reduction.

Table IV shows the area results in terms of Stratix IV logic utilization, M9K blocks, and DSP blocks. The logic utilization metric reported by Quartus II is an estimate of how full the device is, calculated from the number of half-ALMs (adaptive logic modules) used in the design. M9Ks are Altera's on-chip RAMs which can hold up to 9 Kbits of data including parity bits [2]. M144Ks, which are much larger RAMs that can hold up to 144 Kbits, are only used by one benchmark, Division, and hence are not shown on the table for space reasons. Note that usage of M144K blocks is taken into consideration when calculating the total area of the systems (see below). The area results presented in Table IV are for the entire system, which includes the MIPS processor, the on-chip cache, the DDR2 controller, the interconnection network, as well as the hardware accelerators. In general, as expected, the area increases as parallelism is increased, both in terms of Pthreads and OpenMP accelerators. Mathematically intensive benchmarks, such as Black-Scholes, MCML, Mandelbrot, and dfsin, show significant increases in the number DSP blocks. In fact, even though the logic utilization for Mandelbrot is only ∼19%, it could not be parallelized more than 12L1-4L2, since DSP usage was at 95%. Note that, parallelization with Pthreads does not necessarily increase circuit area if different functions are executed in parallel (vs. when these functions are executed sequentially). Thus, for the Hash benchmark, the circuit area is roughly the same for the S and 4L1 configurations.

Area-delay product is another important metric when evaluating the efficiency of different hardware architectures. Calcuating the total circuit area of an FPGA can be particularly challenging since modern FPGAs consist of different types of blocks, such as logic blocks, memory blocks, DSP blocks, and routing, each of which consumes different amount of chip area. To account for this fact, we use the data from [12], which gives the chip tile area for each type of block[4]. Using this data and the results in Table IV, we calculate the total circuit area for each architecture configuration for each benchmark. With this area, and using the wall-clock time results from Tables II and III, we computed the geometric mean area-delay product, shown in Fig. 5 as a percentage compared to the baseline architecture. Similar to Fig. 4, multiple lines are shown, each of which includes results for different architectures/benchmarks. Looking at the geomean result for *all* benchmarks (first line of the legend), the 4L1 architecture shows the best result with 53.5% area-delay product of the baseline case, and the 4L1-2L2 and the 4L1-4L2 architectures follow with 58% and 63% respectively. For the three benchmarks where loop pipelining

[4]Note that although [12] provides detailed area data for the types of tiles in Stratix III, Stratix IV contains the same types of tiles, so we believe the data can be used for this relative area comparison.

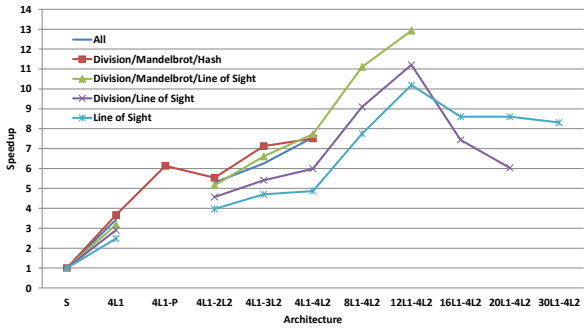

Fig. 4. Geomean speedup ratios.

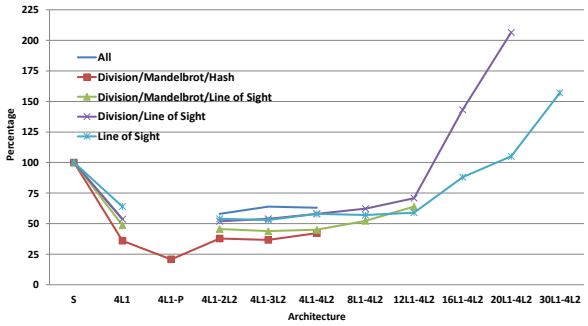

Fig. 5. Geomean area-delay ratios.

could be used, 4L1-P shows the best result with 20.8% and 4L1 follows with 36.7%. For the Division benchmark, 4L1-P showed 12% area-delay product when compared to the sequential case. Similar to the speedup results, as the degree of parallelism is excessively increased, the area-delay product is also significantly increased, with the 20L1-4L2 architecture showing 206% over baseline for the Division/Line of Sight plot.

In summary, the results above demonstrate the capabilities of our HLS tool to synthesize circuits automatically into a variety of architectures with dramatically different degrees of parallelization. Promising results are observed in terms of execution time and area-delay product. It is worthwhile to reiterate that with our approach, changing the parallelization configuration is straightforward, with Pthreads requiring only a few lines of code changes, and OpenMP requiring a *single* number (`num_threads` clause) to be changed. This enables wide design space exploration with ease, which is certainly not feasible with manually designed hardware. With our framework, exploring with different parallelization schemes in hardware is no more difficult than the analogous exploration in software.

VII. Conclusions and Future Work

In this paper, we presented a framework which can automatially compile software threads to parallel hardware accelerators. Two standard software parallelization techniques, Pthreads and OpenMP, are used to generate hardware accelerators which execute concurrently in a shared memory system. OpenMP allows a section of code, such as a loop, to be automatically compiled to parallel accelerators, whereas Pthreads allow the same and/or different functions to be synthesized to concurrently operating hardware accelerators. Our framework allows nested parallelism, where Pthreads can invoke OpenMP threads to allow 2 levels of parallel accelerators. Loop pipelining can also be used in conjunction with Pthreads. A key advantage of our framework is that software engineers

TABLE II. PERFORMANCE RESULTS OF ALL ARCHITECTURES FOR BLACK-SCHOLES, MCML, AND MANDELBROT BENCHMARKS.

	Black-Scholes			MCML			Mandelbrot		
	Time (Speedup)	**Cycles (Speedup)**	**Fmax (Ratio)**	**Time (Speedup)**	**Cycles (Speedup)**	**Fmax (Ratio)**	**Time (Speedup)**	**Cycles (Speedup)**	**Fmax (Ratio)**
S	2058 (1)	437947 (1)	130.41 (1)	22317 (1)	1876187 (1)	84.07 (1)	18724 (1)	2523455 (1)	134.77 (1)
4L1	615 (3.9)	108654 (4.0)	127.39 (0.98)	6308 (3.5)	482855 (3.9)	76.55 (0.91)	4930 (3.8)	627059 (4.0)	127.18 (0.94)
4L1-P	N/A	N/A	N/A	N/A	N/A	N/A	4248 (4.4)	434533 (5.8)	102.28 (0.76)
4L1-2L2	488 (6.6)	56344 (7.8)	110.68 (0.85)	3613 (6.2)	246094 (7.6)	68.12 (0.81)	2849 (6.6)	355224 (7.1)	124.67 (0.93)
4L1-3L2	397 (6.9)	42410 (10.3)	87.3 (0.67)	3635 (6.1)	246094 (7.6)	67.71 (0.81)	1904 (9.8)	241805 (10.4)	127 (0.94)
4L1-4L2	364 (10.7)	28736 (15.2)	91.83 (0.7)	1961 (11.4)	125463 (15.0)	63.97 (0.76)	1455 (12.9)	185863 (13.6)	127.73 (0.95)
8L1-4L2	N/A	N/A	N/A	N/A	N/A	N/A	1131 (16.6)	126828 (19.9)	112.17 (0.83)
12L1-4L2	N/A	N/A	N/A	N/A	N/A	N/A	1089 (17.2)	113593 (22.2)	104.31 (0.77)

TABLE III. PERFORMANCE RESULTS OF ALL ARCHITECTURES FOR LINE OF SIGHT, DIVISION, HASH, AND DFSIN BENCHMARKS.

	Line of Sight			Division			Hash			Dfsin		
	Time (Speedup)	**Cycles (Speedup)**	**Fmax (Ratio)**	**Time (Speedup)**	**Cycles (Speedup)**	**Fmax (Ratio)**	**Time (Speedup)**	**Cycles (Speedup)**	**Fmax (Ratio)**	**Time (Speedup)**	**Cycles (Speedup)**	**Fmax (Ratio)**
S	4146 (1)	556933 (1)	134.34 (1)	2947 (1)	407837 (1)	138.37 (1)	2654 (1)	325859 (1)	122.76 (1)	2058 (1)	265513 (1)	129.02 (1)
4L1	1672 (2.5)	197625 (2.8)	118.22 (0.88)	867 (3.4)	112466 (3.6)	129.7 (0.94)	696 (3.8)	86378 (3.8)	124.16 (1.01)	615 (3.4)	67507 (3.9)	109.78 (0.85)
4L1-P	N/A	N/A	N/A	235 (12.5)	29457 (13.9)	125.13 (0.9)	635 (4.2)	83748 (3.9)	131.8 (1.07)	N/A	N/A	N/A
4L1-2L2	1045 (4.0)	127471 (4.4)	121.94 (0.91)	560 (5.3)	67237 (6.1)	120.11 (0.87)	540 (4.9)	64588 (5.1)	119.67 (0.97)	488 (4.2)	47533 (5.6)	97.5 (0.76)
4L1-3L2	882 (4.7)	102313 (5.4)	115.98 (0.86)	473 (6.2)	51349 (7.9)	108.59 (0.78)	449 (5.9)	50917 (6.4)	113.3 (0.92)	397 (5.2)	34520 (7.7)	86.89 (0.67)
4L1-4L2	851 (4.9)	98708 (5.6)	115.97 (0.86)	400 (7.4)	46854 (8.7)	117.12 (0.85)	594 (4.5)	66219 (4.9)	111.56 (0.91)	364 (5.7)	27463 (9.7)	75.55 (0.59)
8L1-4L2	535 (7.6)	59361 (9.4)	111.05 (0.83)	276 (10.7)	27627 (14.8)	100.08 (0.72)	N/A	N/A	N/A	N/A	N/A	N/A
12L1-4L2	407 (10.2)	43540 (12.8)	106.87 (0.80)	239 (12.3)	23133 (17.6)	96.64 (0.7)	N/A	N/A	N/A	N/A	N/A	N/A
16L1-4L2	482 (8.6)	49664 (11.2)	102.94 (0.77)	458 (6.4)	42663 (9.6)	93.09 (0.67)	N/A	N/A	N/A	N/A	N/A	N/A
20L1-4L2	482 (8.6)	44036 (12.7)	91.37 (0.68)	697 (4.2)	57042 (7.2)	81.87 (0.59)	N/A	N/A	N/A	N/A	N/A	N/A
30L1-4L2	499 (8.3)	42558 (13.1)	85.34 (0.64)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

TABLE IV. AREA RESULTS OF ALL ARCHITECTURES FOR ALL BENCHMARKS.

	Black-Scholes			MCML			Mandelbrot			Line of Sight			Division			Hash			Dfsin		
	Logic Util	**M9K**	**DSP**	**Logic Util**	**M9K**	**DSP**	**Logic Util**	**M9K**	**DSP**	**Logic Util**	**M9K**	**DSP**	**Logic Util**	**M9K**	**DSP**	**Logic Util**	**M9K**	**DSP**	**Logic Util**	**M9K**	**DSP**
S	43474	140	88	38810	136	120	22111	134	28	23550	134	12	24527	134	8	31015	262	36	51763	136	52
4L1	111156	142	328	87874	150	456	26869	134	88	40385	134	24	43134	134	8	31210	262	36	138380	142	184
4L1-P	N/A	N/A	N/A	N/A	N/A	N/A	27208	138	56	N/A	N/A	N/A	43680	138	8	32414	278	36	N/A	N/A	N/A
4L1-2L2	198126	174	648	152660	166	904	34187	158	168	54926	158	40	79208	158	8	49550	326	64	276756	158	360
4L1-3L2	276243	174	968	264559	166	1020	37476	158	248	62988	158	56	98317	158	40	58328	326	92	387958	375	568
4L1-4L2	361327	443	1020	355873	419	1020	40890	158	328	70834	158	72	119359	158	72	69031	326	120	408169	451	776
8L1-4L2	N/A	N/A	N/A	N/A	N/A	N/A	61251	182	648	108737	182	136	204230	182	136	N/A	N/A	N/A	N/A	N/A	N/A
12L1-4L2	N/A	N/A	N/A	N/A	N/A	N/A	81224	206	968	146380	206	200	275185	635	200	N/A	N/A	N/A	N/A	N/A	N/A
16L1-4L2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	181166	230	264	340888	1280	264	N/A	N/A	N/A	N/A	N/A	N/A
20L1-4L2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	214658	254	328	400613	1280	328	N/A	N/A	N/A	N/A	N/A	N/A
30L1-4L2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	310013	324	488	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

without hardware knowledge can use standard software APIs to obtain significant speedup through parallel hardware systems. Geometric mean results over 7 benchmarks showed that using 4 Pthreads accelerators, each of which contains 4 OpenMP accelerators, provides the best performance results, with 7.6× speedup and 63% area-delay product when compared to the single threaded system. The highest speedup was 17.2× in wall-clock time with the 12L1-4L2 architecture, and the best area-delay product was 12% (over 8× improvement) with the 4L1-P architecture.

Future work includes adding support for fast estimation of speed and area with different configurations of Pthreads and OpenMP accelerators for rapid design space exploration.

REFERENCES

[1] Altera, Corp., San Jose, CA. *SOPC Builder User Guide*, 2010.

[2] Altera, Corp., San Jose, CA. *TriMatrix Embedded Memory Blocks in Stratix IV Devices*, 2011.

[3] Altera, Corp., San Jose, CA. *Implementing FPGA Design with the OpenCL Standard*, 2012.

[4] J. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4, 1965.

[5] J. Cong and Y. Zou. FPGA-based hardware acceleration of lithographic aerial image simulation. *ACM Trans. Reconfigurable Technol. Syst.*, 2(3):1–29, 2009.

[6] A. Canis et. al. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *ACM Int'l Symp. on FPGAs*, pages 33–36, 2011.

[7] A. Cilardo et. al. Efficient and scalable OpenMP-based system-level design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 988 – 991, 2013.

[8] A. Papakonstantinou et. al. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *IEEE Symposium on Application Specific Processors*, pages 35–42, 2009.

[9] D. Andrews et. al. Programming models for hybrid FPGA-cpu computational components:a missing link. In *IEEE Micro*, pages 42–53, 2004.

[10] D. Andrews et. al. hthreads: a hardware/software co-designed multithreaded rtos kernel. In *IEEE Conference on Emerging Technologies and Factory Automation*, pages 330–338, 2005.

[11] D. Cabrera et. al. OpenMP extensions for FPGA accelerators. In *IEEE Int'l Conference on Systems, Architecture, Modeling and Simulation (SAMOS)*, pages 17–24, 2009.

[12] H. Wong et. al. Comparing FPGA vs. custom cmos and the impact on processor microarchitecture. In *ACM Int'l Symnp. on FPGAs*, pages 5–14, 2011.

[13] J. Choi et. al. Impact of Cache Architecture and Interface on Performance and Area of FPGA-Based Processor/Parallel-Accelerator Systems. In *IEEE Int'l Symposium on FCCM*, pages 17–24, 2012.

[14] J. Cong et. al. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[15] W. Peck et. al. Hthreads: A Computational Model for Reconfigurable Devices. In *Int'l Conference on FPL*, pages 1–4, 2006.

[16] Y. Hara et. al. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[17] Y.Y Leow et. al. Generating hardware from OpenMP programs. In *IEEE Int'l Conference on FPT*, pages 73–80, 2006.

[18] Impulse. *Impulse CoDeveloper – Impulse accelerated technologies (http://www.impulseaccelerated.com)*, 2010.

[19] The Khronos Group. *OpenCL The open standard of parallel programming of heterogeneous systems. (http://www.khronos.org/opencl)*.

[20] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE Int'l Symp. on Code Generation and Optimization*, pages 75–86, 2004.

[21] Lawrence Livermore National Laboratory. *POSIX Threads Programming. (https://computing.llnl.gov/tutorials/pthreads)*, 2013.

[22] NVIDIA. *CUDA Parallel Programming and Computing Platfrom. (http://www.nvidia.ca/object/cuda_home_new.html)*.

[23] The OpenMP Architecture Review Board. *The OpenMP API specification for parallel programming. (http://www.openmp.org)*, 2013.

[24] Oregon Medical Laser Center. *Monte Carlo Simulations. (http://omlc.ogi.edu/software/mc/)*.

[25] G. Stitt and F. Vahid. Thread warping: a framework for dynamic synthesis of thread accelerators. In *IEEE/ACM international conference on CODES+ISSS*, pages 93–98, 2007.

[26] United States Bureau of Labor Statistics. *Occupational Outlook Handbook 2010-2011 Edition*, 2010.

[27] Xilinx, Inc., San Jose, CA. *Vivado Design Suite User Guide*, 2012.