Abstract—OpenMP enables productive software development that targets shared-memory general purpose systems. However, OpenMP compilers today have little support for future heterogeneous systems – systems that will more than likely contain Field Programmable Gate Arrays (FPGAs) to compensate for the lack of parallelism available in general purpose systems.

We have designed a high-level synthesis flow that automatically generates parallel hardware from unmodified OpenMP programs. The generated hardware is composed of accelerators tailored to act as hardware instances of the OpenMP task primitive. We drive decision making of complex details within accelerators through a constraint-programming model, minimizing the expected input from the (often) hardware-oblivious software developer.

We evaluate our system and compare them to two state of the art architectures – the Xeon PHI and the AMD Opteron – where we find our accelerators to perform on par with the two ASIC processors.

I. INTRODUCTION

OpenMP is a standard programming model used to exploit parallelism within an application. It allows programmers to annotate source code with compiler directives, indicating where parallelism can be exploited. An OpenMP compiler can transform these directives to parallel, often high-performant, general-purpose processor code.

Currently OpenMP is a model to write software for use in general purpose computer architectures. The work presented in this paper enables OpenMP to be a model for writing software and hardware with little extra effort. The hardware generated can be synthesized and executed on field-programmable gate arrays (FPGAs).

The trend to overcome failing Dennard scaling is to exploit heterogeneity in computation through accelerators – today, acceleration is obtained through GPUs but it is likely that the far more customizable FPGAs will complement that role. In addition, the industry has also seen value in this approach and architectures such as Xilinx ZYNQ and the Intel/Altera platforms are emerging.

There are several reasons why enabling OpenMP-driven high-level synthesis (HLS) is worthwhile. Custom built FPGA hardware is more efficient than general purpose components [10], [7]. Power-consumption – a first-class constraint can also be reduced compared to CPUs or GPUs [11], [8]. FPGAs can be configured to have far more parallel throughput than their general purpose counterparts. Granted, FPGAs (or rather, their development boards) argue costs more than any general purpose CPU or GPU; however, it is likely that the growing interest of vendors will make FPGAs more readily available in the near future (for example Intel QuickAssist for FPGAs or Xilinx ZYNQ).

No matter what benefits FPGAs brings, building custom hardware by hand is time consume and prone to error. Moreover, there exists one hardware engineer for every ten software engineers [12], justifying the need for ease-to-use tools for hardware generation. Even existing HLS solutions needlessly expose too many options to the software developer.

This paper uses BŁYSK [19] – our prototype OpenMP framework, where we added a novel strategy for sharing resources across hardware instances of OpenMP tasks called hyper-tasks. We call these shared resources regional components and show how they can be used to increase performance through parallelism in hardware that is automatically generated. These hyper-tasks require little-to-no input from the developer – the application is analyzed and a constraint-based solver is used to decide the fine details other tools expects the user to supply.

Our framework accepts OpenMP annotated C-code and transforms it into a System-on-Chip composed of a soft-core and a number of hardware accelerators that are fully dedicated to perform OpenMP task-parallel applications as fast as possible. To our best knowledge this is the first time accelerators that utilize both MIMD and SIMD parallelism are automatically generated and modelled with component-sharing between asynchronous parallel computational units, closely tied to the parallel model. We show how both the hyper-tasks and the regional component can be implemented, how to decide on their usage and what constraints are imposed on the hardware and the compiler.

We evaluate hyper-tasks and their scalability and show that it is worthwhile sharing resources to gain performance using FPGAs. We also put our generated systems in perspective by comparing them to existing OpenMP-capable general purpose hardware.
II. RELATED WORK

Overall, high-level synthesis of parallel workloads can be divided into two categories: data-path and state-machine based.

Data-path based HLS involves creating a large data-path, into which data is being push into. The data-path is highly pipelined allowing a high degree of parallelism and throughput. Often the parallel concept of threads is used to decide what data to be fetched into the data-path. Examples of data-path parallel HLS today is for example the work of Halstead et al. [9], Altera’s OpenCL [5] or FCUDA [17]. The benefits of data-path based HLS is that a high-degree of throughput can be obtained. However, data-path construction typically limits the amount of control-flow that can exist in the kernel. Control-flow statements are often predicated and both paths (true and false) execute in parallel.

State-machine based HLS is more generic than the data-path based version. Here, each instruction is mapped into a finite-state machine that governs how data flows through the units and state is maintained globally. There are fewer limitations concerning control-flow but comes at the cost of performance. Examples of state-machine based parallel HLS-tools include the work by Leow et al. [13] and Nowak et al. [16] (albeit microcoded).

Our approach combines both data-path and state-machine execution and is similar to LegUP [3]. Instructions are mapped into a state-machine while basic blocks without control-flow are decomposed into data-paths. Additionally, unlike existing work, our work promotes sharing of expensive resources. We formally model the problem of sharing as a constraint optimization problem unlike related work that uses heuristics [3], [4]. Our work also allows resource sharing across multiple hardware instances while related work primarily considers re-usage of components within one hardware instance.

Cabrera et al. studied FPGA co-acceleration in OpenMP from a run-time system perspective using OmpSs. Their work primarily concerned quantifying overheads in using FPGAs from the runtime-system’s perspective (such as DMA transfers and reprogramming the FPGA) – no high-level synthesis was performed and existing IP cores were used. Later, Filgueras et al. [6] continued to work by introducing directives into their Mercurium compiler, which in turn were used to synthesize VHDL code using Xilinx Vivado. Our work differs in that our tool both source-to-source compiles and performs the high-level synthesis of the OpenMP application.

DURASE [14] is a system for automatically generating hardware for certain portions of a serial applications code. As with the present work, DURASE model the problem using constraint programming. However, DURASE is limited to serial application and only focuses on matching tree’s in the intermediate code to decide what portions should be accelerated – they are oblivious to the problem of sharing resources across several asynchronous processing units, which is what the present paper tackles.

III. OPENMP-DRIVEN HLS OVERVIEW

The steps performed by our framework are seen in Figure 1:a. The user provide an OpenMP annotated source code, which is the basis for the custom system-on-chip that our framework generates. We focus on exploiting parallelism using asynchronous tasks (#pragma omp task) with (OpenMP 4.0) or without (OpenMP 3.0) data-flow information. Every detected OpenMP task will be converted into what we call a hyper-task – a custom hardware unit that perform the work within a particular task. Several hyper-tasks are clustered together for resource efficiency and are then called an accelerator (Figure 1:b). Once accelerators have been created, they are bundled together with a NiosII soft-core and a system-on-chip is created. The system-on-chip is very similar to the IBM Cell processor, where the NiosII acts as a master towards the OpenMP accelerators. Knowing the layout of the SoC, the compiler now can proceed to translate remaining parts of the source-code to exploit the newly generated SoC. All detected OpenMP constructs related to the tasks are source-to-source translated into using the generated accelerators. The output is an Altera QSYS hardware description file and an associated C file for the NiosII processor.

IV. ACCELERATORS AND HYPER-TASKS

The hyper-tasks generated by our method are composed of a number of primitive components (such as adders or multipliers). Each hardware component is associated with a component declaration (ports), number of pipeline stages and a tree-match it represents in the intermediate dependency-graph. We refer to components that are shared within one hyper-task as private. A private component can be promoted to become shared across hyper-tasks and is then called a regional component (section IV-A).

A private hardware component can be used by several intermediate instructions (within one hyper-task) as long as they start at different times. When a component is used by several instructions, multiplexers will be added to choose inputs into the component. Components that are private require the scheduled instructions to capture the results according to the component’s latency (the result is unbuffered).

We support arbitrary wide vectorization instructions that can be used from within the C source code. The vector components are automatically formed by linking several smaller components together. Exploiting vectorization requires the programmer to use certain pre-defined built-in functions (conceptually similar to [21]).

A. Regional Components

Original to this paper are components that can be shared between hyper-tasks. We call them regional components. Any component, from the smallest of adders to the largest of cross-products can be a regional component. Figure 2 illustrates the internals of a regional component.

Unlike private components, regional components are extended with logic to support sharing. An arbiter is integrated into the component. The arbiter will choose one of the many possible hyper-tasks (HTs) that are requesting to use the regional component. We chose a fixed-priority based arbitration scheme.
The fixed-priority arbitration assumes that a number of requesters $HT_1, HT_2, \ldots, HT_n$ attempt to use the regional component. The fixed-priority arbitration will prioritize the lowest ranked requesters, meaning that $HT_1$ will have uncontested access to the component, $HT_2$ will acquire it only if $HT_1$ does not request it and so forth.

The arbiter’s decision is a bit-vector, where each bit is sent to each of the hyper-tasks. The arbiter’s decision is saved and pipelined alongside the computation and decides which of the FIFO buffers sitting at the end of the pipeline the result should be captured into. The latency of a regional component is always one clock cycle longer than its private counterpart because of the FIFO buffer.

**B. Generating Accelerators and Hypertasks**

The back-end of our compiler will generate an accelerator for the OpenMP-task given its intermediate representation code. The generated accelerators are memory-mapped, sitting on an Avalon [1] interconnect. They act as slaves towards the host processor (the NiosII) and as master towards other units (such as RAM). Communication between the Host and the Accelerator is performed through control-registers and includes functionality such as associating a hyper-task with an address, assigning identifiers to each hyper-tasks or accelerators (which allows SPMD/SIMT execution or redundancy computing if desirable) and status registers for starting task execution.

1) **Building the Accelerator:** Building the accelerator involves mapping the intermediate representation into the hyper-tasks as well as deciding what resources to be shared within an accelerator. Intuitively, large components should be made regional in order to conserve FPGA resources, while small components should be made private to reduce multiplexer cost. This intuition does however come with problems. Depending on what operands are used for the components, it might still be worthwhile to make large components private. For example, a kernel that consists of a large number of duplicated `fadd` $s1, s2, s3$ instructions can have the floating-point adder held private in each hypertask, reuse the adder continuously with minimal multiplexer pressure (since operands do change).

Another problem is the DSP-block pressure. FPGA resources can be divided into two types: ALMs (or LUTs) which is the internal block that logic operations are mapped to, and DSP which are hard-wired components used to assist for example multiplications. A kernel that is large on for example multiplications (which is an ALM friendly component because
it uses DSP blocks) will therefore benefit from having the multiplication unit regional rather than private.

Transforming the intermediate representation into hardware is a known NP-hard problem. Unlike related work that use heuristics to solve above-mentioned problems, we solve it as an optimization problem through a constraint satisfac-
tory problem (CSP). Constraint Programming is appealing: the model is formed and use a branch-and-bound engine to repeatedly search for a solution, adding constraints that monotonically improve the solution. The problem formulation was implemented through Gecode [20].

Our model uses the following variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G \in \mathbb{Z} )</td>
<td>Maximum number of resource of a given type</td>
</tr>
<tr>
<td>( n \in [1,16] )</td>
<td>Number of Hyper-tasks per accelerator</td>
</tr>
<tr>
<td>( i \in I )</td>
<td>An instruction</td>
</tr>
<tr>
<td>( R_i \in [-G, +G] )</td>
<td>What resource instruction ( i ) will use.</td>
</tr>
<tr>
<td>( S_i \in [0, +G] )</td>
<td>Scheduled state of ( i ) in the FSM</td>
</tr>
<tr>
<td>( C_i \in [0, 1] )</td>
<td>Commutative property of instruction ( i )</td>
</tr>
<tr>
<td>( Y_i \in [0.6] )</td>
<td>Which component instruction ( i ) uses (we have 60 different components)</td>
</tr>
<tr>
<td>( O_{ij} \in [0, G] )</td>
<td>Operand ( j ) used in instruction ( i )</td>
</tr>
</tbody>
</table>

1, \( O \) and \( Y \) are known a-priori, obtained from the intermediate representation (the number of instructions, their type and which operands they use). Variable \( G \) is bound through statistical-analysis a-priori to the computation. For the kernels we have used, \( G \) is often bound to 4 and \( O \) bound to the total number of different registers/constants used in program.

Our goal is to maximize the performance-metric \( M \). The performance metric is calculated using an approximation of the number of instruction executed per cycle (IPC) linearly scaled by the number of hyper-tasks in the design over the area estimated for the design:

\[
M = \max \left( \frac{\text{IPC}}{A} \right)
\]

Estimating the cycle time to execute the kernel \( t \) is calculated by summing up the last finishing instruction in each basic-block. Estimating the kernel execution time this way is often used. The metric \( M \) is scaled to avoid the need for floating-point variables in the constrained model. The model has one hardwired constraint: the total number of DSP blocks used in the design must not exceed 256 (which is what our Stratix V FPGA contains).

2) Instruction scheduling constraints: Each instruction \( i \) is scheduled on a state in the hyper-task's FSM, on a particular resource \( R_i \) and uses a certain component group \( Y_i \).

Two independent instructions \( i \) and \( j \) may not be scheduled on the same state because they use the same resource:

\[
\forall i, j \in I : (R_i = R_j \land Y_i = Y_j) \implies S_i \neq S_j
\]

Similarly, an instruction \( j \) which is dependent on an instruction \( i \) must be scheduled after instruction \( j \) have finished. An extra cycle is added should instruction \( i \) be scheduled on a regional component:

\[
\forall i \in \text{dep}(j) : R_i < 0 \implies S_i + \text{latency}(Y_i) + 1 < S_j
\]

Two instructions \( i \) and \( j \), which both use a regional component, may neither be scheduled on the same state (due to arbitration requests) nor finish when the other one is awaiting arbitration decision:

\[
\forall i, j \in I : R_i < 0 \land R_j < 0 \implies (S_i \neq S_j) \land (S_i + \text{latency}(Y_i) \neq S_j) \land (S_j + \text{latency}(Y_j) \neq S_j)
\]

An instruction \( i \) which uses a regional resource may not be scheduled on a state where another instruction \( j \) that uses a private component is in flight

\[
\forall i, j \in I : R_i < 0 \land R_j \geq 0 \implies (S_i < S_j) \land (S_i \geq S_j + \text{latency}(Y_j))
\]

3) Area constraints: Modelling the area is crucial to get a metric that reflects the hardware that is generated (and synthesized). We empirically derived individual multiplexer, components, arbiter and fifo area (ALM) and DSP cost through isolated synthesis. Isolated synthesis provides us with an upper-bound on the area consumption because there will always be ALMs that are not fully utilized – ALMs that could be packed with logic in a larger design. This can be seen in the area-evaluation for one of the benchmarks, where our estimated area is larger than that of the synthesized version for a number of solutions (Figure 3). While there is a large deviation between the synthesized- and the model-area, they correlate well, meaning that the performance metric we optimize over is accurately reflected in the generated hardware. We call the functions that estimates the area: \text{cost(), cost_mux(), cost_fifo()} and \text{cost_arbiter()}.

The area for the accelerator is calculated from the area estimated by the regional components (including arbiters and FIFOs), the privatized components (multiplied with the number of hyper-tasks) and the total multiplexer area.

\[
A = \sum \text{Regional} + n \times \sum \text{Private} + \text{Max_private} + \text{Max_regional} + \text{Max_output}
\]

The \text{Regional} and \text{Private} components are constrained primarily through channeling. A boolean array \( B \) is created of size \( n \times 2 \times G \) – large enough to hold all information regarding the potential resource usage for each of the instruction. A ‘1’ in the row of this array indicate that a particular resource has been decided for the instruction with the same row index, modeled (through channeling constraints) as:

\[
\exists i \in I, \forall j \in R : R_i = j \iff B(Y_i, R_i) = 1
\]

\[
\forall j \in Y : \text{Regional}_j = \sum_{i=-G}^{i+c} B(j, i) \times \text{cost}(Y_j)
\]

\[
\forall j \in Y : \text{Private}_j = \sum_{i=0}^{i+c+G} B(j, i) \times \text{cost}(Y_j)
\]

Constraining the component’s input multiplexers is done similarly, but with operands instead of component-types. We are calculating the number of unique operands used for a

\(^1\)Private instruction always capture the value according to their components latency. Any stalls (by waiting for arbitration) will make the private instruction capture an incorrect value.
Fig. 3: Comparison between our area-model and the area resulting from synthesis of the hardware using Altera Quartus II.

particular resource $R$. If the same operand is used for a particular resource, then our backend will not duplicate the input but rather re-use an input.

A function, $\text{count}(y, r, p)$ is used, which calculates the number of unique operands used for a particular component ($y \in Y$) and its resource ($r \in R$) for one of its input ports ($p$). The total multiplexer cost is calculated as:

$$Mux_{\text{private}} = \sum_{i=0}^{60} \sum_{j=k < \text{ports}(Y)} \sum_{k=0} \text{cost}_{\text{max}}(\text{count}(i, j, k)) \cdot n$$

$$Mux_{\text{regional}} = \sum_{i=0}^{60} \sum_{j=k < \text{ports}(Y)} \sum_{k=0} \text{cost}_{\text{max}}(\text{count}(i, j, k)) + \text{cost}_{\text{arbiter}}(n) + \text{cost}_{\text{fifo}}(n)$$

Constraining the multiplexer area for the output ($Mux_{\text{output}}$) of operations is done similarly as for the input, albeit now the matrix $\text{count}(\cdot)$ is defined on the output operands rather than on the component/operand pair.

We model commutative properties of operations and operands and the model can choose to switch operands to potentially reduce the area (should the input already exist on one of the multiplexer to the component):

$$\forall i \in I : C_i = 1 \implies \text{switch_operand}(O_{i1}, O_{i2})$$

C. System-Generation

We compose the entire system-on-chip once the accelerators have been described. We use the Avalon interconnect as a medium for communication. The programmer can choose how many accelerators will be mapped onto the Avalon interconnect. Each mapped accelerator will have its memory-mapped address recorded for further use by the compiler. Aside from the accelerators, a timer, block RAM holding instructions and data and a NiosII f-type processor will be mapped. The output will be given as a system (QSYS) description, with which existing Altera tools can be used to synthesize the design down to hardware.

D. Source-to-Source Translation

 Aside from generating the system, our compiler needs to modify the original source-code so that it is capable of exploiting the auto-generated accelerators. Parts of this flow is illustrated in Figure 4 and involves transforming the two task-parallel constructs in OpenMP: `omp task`, `omp taskwait`.

In our case, the work-sharing construct `omp parallel/single` deviates from the OpenMP specification since it initializes the system for parallel execution but the accelerators do not execute the parallel region– instead they sit idle and wait for work to be scheduled to them.

Booting accelerators (Figure 4:b) requires knowledge of where those accelerators are memory-mapped in the system-generation step (see section IV-C). The boot-up sequence allocates memory holding the arguments to a task for each hyper-task in an accelerator and sets the accelerator’s `device_address` to point into this shared memory position (Figure 4:c).

Creating and exposing a task involves translating the OpenMP directive so that a structure holding the task’s arguments is allocated. The structure is initialized accordingly to the task’s function arguments (Figure 4:b). Calling the `submit_try_run()` function (Figure 4:d) will finally submit the task to the run-time system. `submit_try_run()` is automatically generated and attempts to find an idle hyper-task to schedule the work on. The idle hyper-task returned is based on the scheduling policy. Scheduling a task onto an accelerator involves copying the arguments of the task into where the hyper-task is expected to find the arguments according to the previously set `device_address`. Execution starts by asserting the correct bit in the `Start control register belonging to the accelerator where the hyper-task is. Should no hyper-task be found, the task is pushed onto a centralized queue, thus postponing the execution.

Finally, the `omp taskwait` function that synchronizes (blocks) execution until all previously spawned tasks have finished is translated into a `__sync_all_tasks()` function call (Figure 4:c). `__sync_all_tasks()` loops through all accelerators, scheduling tasks onto them until no more tasks exists.

E. Runtime-System Scheduling Policy

Our OpenMP runtime system’s scheduling policy decides on which hyper-task in which accelerator an OpenMP task is exposed by the application. Scheduling policies in modern general-purpose systems are complex, taking architectural details such as caches and memory hierarchy into account. Fortunately, such complex thinking is needless for the system-on-chips that our framework generates.

The one architectural detail that we need to take into account when scheduling is the arbitration details that regulate access to regional components. It can be crucial for performance reasons to schedule tasks onto the OpenMP accelerator with the least amount of tasks currently running (the accelerator that is least loaded). The reason is illustrated in Figure 5. Here, an
The auto-generated system always generated four banks of on-chip memory to increase the memory bandwidth. The per-bank data-width varied between 32- and 512 bits and depends on the vector width used in the OpenMP tasks. The clock-frequency was set according to Altera Timing Analyzer in order to meet all timing requirements (to meet critical path on-chip memory). The hardware components in our study were taken from the Altera library and all floating-point components are using single-precision.

We compared our generated OpenMP accelerators with the 1.1 GHz, 57-core Xeon PHI coprocessor and a 2.7 GHz AMD Opteron 48-core SMP (AMD Opteron 6172) system.

V. RESULTS

A. Experimental Setup

1) System: Evaluation was done using the Altera/Terasic DE5-Net board that contains an Altera Stratix-V GX FPGA (5SGXEA7N2F45C2). The auto-generated system contains the NiosII/F processor, a JTAG-UART interface and a timer, all connected through the Avalon interconnect. Altera Quartus II v14.0 was used for synthesis of the generated systems.

The auto-generated system always generated four banks of on-chip memory to increase the memory bandwidth. The per-bank data-width varied between 32- and 512 bits and depends on the vector width used in the OpenMP tasks. The clock-frequency was set according to Altera Timing Analyzer in order to meet all timing requirements (to meet critical path for 80°C). The hardware components in our study were taken from the Altera library and all floating-point components are using single-precision.

We compared our generated OpenMP accelerators with the 1.1 GHz, 57-core Xeon PHI coprocessor and a 2.7 GHz AMD Opteron 48-core SMP (AMD Opteron 6172) system.

Fig. 4: Source-to-source transformation of a OpenMP task-parallel application (a) into a set of run-time system calls (b): booting a number of thread (c), creating tasks (d) and a task-barrier (e).
We used the identical input source-code for all benchmarks except the matrix multiplication. For the matrix multiplication, we both used the naïve implementation that our compiler compiles with, but we also include the ATLAS (for the Opteron) and Intel MKL (for the Xeon PHI) versions. For the AMD Opteron system, we used the GNU C Compiler (4.9.2) with -O2 with OpenMP support enabled (-fopenmp). For the Xeon PHI system we used the Intel C Compiler (14.0.2) with -O2 and OpenMP support enabled (-openmp). All Xeon PHI experiments were executed natively on the device (no transfer overheads). We hand-optimized our intermediate code to accommodate loop-unrolling that our compiler does not yet have. The hardware generated was the best versions found by the constraint solver after 15 minutes of runtime.

We compared the performance of all three systems by normalizing the performance against the uni-accelerator OpenMP accelerator version:

\[
\text{speedup} = \frac{t_{\text{serial}}(\text{ACC})}{t_{\text{parallel}}(\text{ACC})}
\]

2) Benchmarks: We used the following kernels to evaluate our OpenMP accelerators:

- **PI-calculation** iteratively approximates the constant \(\pi\) by calculating the continued fraction (1 billion iterations)
- **Mandelbrot Fractal** draws the Mandelbrot fractal on a 4096x4096 pixel-map
- **MatMul** calculates the product of two matrixes (1024x1024 with block-size: 64x64)
- **N-Body simulation** simulates the Newtonian forces between 16384 heavenly bodies
- **Raytrace** tests for 680 million ray-triangle intersection (based on [15])

B. Results

The performance of our OpenMP task accelerators are plotted in Figure 6. For the PI-kernel, Figure 6:a, our OpenMP accelerators performs significantly better than both the Xeon PHI and the AMD Opteron on an accelerator to core basis. Performance can reach as high as 20x faster than one AMD Opteron, requiring several AMD Opteron sockets to reach similar performance levels. Xeon PHI performs slightly better than the AMD Opteron but is 17% slower than our OpenMP accelerators. The PI OpenMP accelerator generated by our model is limited by the amount of DSP-blocks available on the device.

The OpenMP accelerator performs the poorest on the N-Body kernel in Figure 6:b. Both the Xeon PHI and the AMD Opteron outperforms the accelerator with almost a magnitude of performance improvements. There are two reasons for this: our OpenMP accelerators lacks caches and since the N-Body kernel is memory-intensive, there is a large contention on the shared bus. The second problem is that our N-Body kernel is not vectorized, while the versions of both AMD Opteron and Xeon PHI are.

The Raytrace benchmark (Figure 6:c) is another case where the OpenMP accelerator perform worse than the AMD Opteron. Surprisingly, the OpenMP accelerators are still on par with the Xeon PHI cores with near equivalent performance. The main limitation for the accelerators is the memory bandwidth, where we found significant time spent blocking on memory reads/writes.

The matrix multiplication case shows how well the OpenMP accelerators can perform, even without caches (Figure 6:d). Here the OpenMP accelerators had 512-bit vector floating point units, exploiting the massive SIMD parallelism available in the multiplications. The performance is on par with both the Intel MKL library on Xeon PHI and ATLAS on AMD Opteron on an accelerator to core basis. The main limitation to performance is the amount of DSP-blocks in our FPGA and the lack of caches. When using the identical source code (marked -identical in the graph), our accelerators outperform the general purpose systems.

The OpenMP accelerators perform on par with the Xeon PHI for the Mandelbrot fractal generator, while the AMD Opteron outperforms both by a factor of 5x. The limiting factor for the OpenMP is the area it occupies on the FPGA. Also, our OpenMP accelerators does not use vectorizing support. Interestingly, the Mandelbrot fractal kernel was the only kernel the constraint programming model deemed worth of having more than eight hyper-tasks per accelerators.

Overall, the automatically generated systems fare well compared to modern ASIC processors. Our OpenMP accelerators executed under a clock frequency of between 90 MHz-100 MHz, which is far below the several GHz that the two other machines used, yet yielded comparable performance.

VI. CONCLUSION AND DISCUSSION

In this paper, we have shown design direction for an OpenMP driven hardware synthesis. We introduce the hyper-tasks – instances of the OpenMP task primitive – that reside within accelerators and that can exploit MIMD, SIMD and SPMD capabilities. We introduced regional components – components that can be shared within an accelerator across several hyper-tasks. We have shown how to formulate and solve hardware generation through constraint-programming.

We showed that our OpenMP accelerators can provide comparable performance-levels to modern high-end systems running the very same OpenMP application. The methods described are portable – every new generation of FPGAs that are released can support them. As the capacity of future FPGAs grows, so will the performance obtained through our methods. Additionally, our method minimizes the expected input from the user (letting most of the work be performed by the model).

The main limitation is that we do not model the memory hierarchy – there are no per-accelerator caches. This limits the performance, as every memory access will have to go out on the shared bus. While adding a cache would be trivial, it is also counter-intuitive as we would expect the user to provide the type and size for the cache. Our intention for the future is to model the caches and the rest of memory hierarchy.

ACKNOWLEDGEMENTS

The reviewers, for valuable comments that helped making the paper better. Christian Schulte, Gabriel Hjort Blindell and Roberto Castaneda Lozano, for providing feedback during early stages of the paper.
Fig. 6: Performance comparison for the various benchmarks running on our OpenMP accelerators.