# Performance and Productivity Evaluation of Hybrid-Threading HLS versus HDLs

Gongyu Wang, Herman Lam, Alan George
NSF Center for High-Performance Reconfigurable Computing (CHREC), Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL
{wangg, hlam, george}@chrec.org

Glen Edwards
Convey Computer Corporation
Richardson, TX 75081
gedwards@conveycomputer.com

*Abstract*—**FPGA-based reconfigurable computing is finding its way into a wide range of application areas in which high performance and low power consumption are paramount. However, FPGA-application development using hardware-description languages (HDLs) faces many productivity challenges that limit its wide adoption, including a steep learning curve and lengthy compilation. High-level synthesis (HLS) languages and tools aim to overcome these challenges by providing familiar high-level languages and tools for FPGA-application development. In using HLS, however, an important consideration is the cost-benefit tradeoff for performance and productivity. Hybrid-threading (HT) is a new open-source HLS toolset from Convey Computer, Corp. that features a programming language based on C/C++ and a set of tools for efficient compilation, verification, and implementation. In this paper, we present a performance and productivity tradeoff study of HT HLS versus HDLs using three RC-amenable kernels, each chosen for their distinctive computational requirements. Our results show that for all three kernels, HT achieved over 80% performance for a fraction of development time, in comparison to corresponding optimized HDL-based designs.**

***Keywords—FPGA; HLS; HDL; performance; productivity***

## I. Introduction

Reconfigurable computing (RC) technology using FPGAs provides opportunities to achieve better performance and lower power consumption than the corresponding software implementations for a wide range of high-performance applications such as bioinformatics, image processing, and graph processing (e.g. [6][15][24]). For FPGA-application development, the most widely used languages are hardware-description languages (HDLs) such as VHDL and Verilog. HDLs provide expressive specification capabilities down to the RTL level, which allows the design of an application to be mapped to FPGA hardware resources in an optimal manner. However, it is well known that the usage of HDLs poses significant productivity challenges [18][23]. Firstly, HDLs are cumbersome and have a steep learning curve requiring specifying algorithms at the cycle-accurate RTL level, resulting in more complex code and lengthier development time. Secondly, compilation time of non-trivial FPGA designs is typically hours or even days, which substantially reduces the achievable number of design iterations per day. Finally, compounded by first two problems, design validation is one of the most costly phases in an HDL-based design flow. Debugging HDL code relies heavily on cycle-accurate simulators (e.g., ModelSim [21]), which can be time-consuming due to the low abstraction level of the HDL design. A design change commonly involves many lines of HDL code; and if the change requires a complete re-run of the lengthy compilation process, the design validation cycles become even more painful.

Aiming to address the productivity problems of FPGA-application development, high-level synthesis (HLS) languages and tools (e.g., OpenCL [2], Vivado [14], BSV [22]) enable familiar high-level languages as design entry (often C or its variants), which alleviates the learning-curve problem of HDLs. The validation process is also improved because debugging via high-level simulation can detect and correct many functional errors before the lengthy hardware compilation. However, using HLS languages and tools commonly incurs certain overhead in the form of performance and hardware resources [12]. Thus, it is important to consider the performance/productivity tradeoff in the use of HLS languages and tools.

Hybrid-threading (HT) is a new open-source HLS toolset from Convey Computer, Corp. [7]. HT takes a unique approach to HLS by combining C/C++-like programming language, flexible architectural customization, thread-based execution model and an integrated-system approach to simulation using SystemC. Thus, HT holds the promise of high-level design entry and programming flexibility needed to achieve maximum performance for multiple application domains. In this paper, we present a comparative study between HT HLS and HDLs by exploring the productivity benefits of HT against the tradeoff in performance and resource usage, in the context of three RC-amenable kernels chosen for their distinctive computational requirements.

The rest of the paper is organized as follows. In the next section, we present related studies on HLS-HDL comparison. In Section III, we introduce the Convey HT toolset and identify the productivity benefits of HT by analyzing and comparing its design flow with a typical HDL design flow. Section IV describes the three computational kernels selected from distinctive domains of high-performance reconfigurable computing (image processing, graph processing, and bioinformatics) and the rationale for their selection. In Section V, comparative results on performance, resource utilization, and productivity are presented and discussed. For all three kernels, HT achieved over 80% performance while showing an order of magnitude productivity improvement in comparison to corresponding optimized HDL-based designs. Section VI concludes the paper and discusses future work.

## II. Related Work

There has been much work published on HLS languages and tools. A recent survey [11] summarized and characterized a plethora of HLS tools qualitatively, but it does not provide any quantitative comparisons between them and HDLs. An overview of a number of HLS tools was presented in [20], in which HLS tools were compared to each other and to HDLs in the context of the Sobel edge detector. The comparison was limited to quantifying productivity-related features such as learning curve, documentation, ease of implementation, etc., and was lacking in performance results.

Although there were some comparison studies that considered multiple application domains (e.g., Cong et al. [9] compared AutoPilot (now Vivado) against HDL for video processing and cognitive radios), we observe that in most published studies, HLS tools were evaluated in a specific application domain. In [8], Impulse C was compared to HDL for a bioinformatics algorithm, where it showed better performance and hardware usage. A Scala-based HLS, Chisel [5], was compared to Verilog for the design of a RISC processor, showing better performance and productivity. Bluespec System Verilog (BSV) was compared to Xilinx IPs in [1] for a Reed-Soloman decoder, showing better performance and hardware usage. Altera OpenCL and Xilinx Vivado were shown to achieve comparable performance and resource utilization to HDLs for a linear algebra algorithm in [25]. In [3], four HLS tools (BSV, Chisel, LegUp, and OpenCL) were compared to each other and to Verilog for database algorithms. The results show that BSV and Chisel performed better than Verilog with minimum hardware overhead, but not so for the other two HLS tools.

In this paper, we explore the productivity benefits of HT, a promising HLS toolset, against the tradeoff in performance and resource usage, in the context of reference kernels from multiple application domains.

## III. Comparison of Design Flows

In this section, we analyze the design flow of the HT toolset in comparison with a typical HDL design flow. Since HT is a new toolset, a concise introduction of the HT toolset is presented before the comparative analysis.

### A. *Convey HT Toolset*

We introduce the HT toolset from three aspects: 1) the configurable HT infrastructure including the execution model; 2) the HT programming language; and 3) the HT design flow. More details on HT can be found in [13].

**HT infrastructure and execution model.** As shown in Fig. 1, the HT infrastructure (green blocks) is depicted in the context of a co-processing system. The HT host code can invoke the software part of HT host interface (HIF) to load the

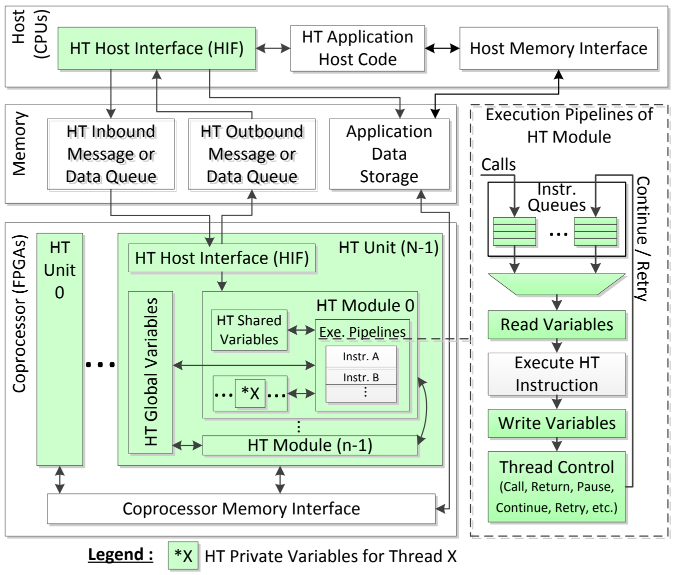

Fig. 1: HT infrastructure

FPGA images, allocate data structures in memory (e.g., message/data queues and application data storage), and dispatch the coprocessor FPGA logics. The coprocessor logics consist of one or more HT units. Each unit contains the hardware part of HIF, one or more HT modules, and optionally global variables accessible by all modules of the unit. The infrastructure manages and replicates the units by N times (N is configurable) so that the designer only needs to program for one unit. Also, the host code dispatch each unit independently via the HIF, so each unit can work on separate tasks.

An HT module consists of the user-programmable part and the infrastructure part. The former is not shown in details in Fig. 1 so we describe it first to facilitate understanding of the latter. Each HT module contains one or more HT instructions programmed by the user using the HT language. HT instruction is the smallest executable element of an HT module and it is a set of operations that can execute within a single clock cycle, such as external-memory interface calls, function calls, module-wise messaging, and thread-control routines. An execution instance of a sequence of HT instructions is defined as an *HT thread* of the module. The infrastructure part of the module manages its threads independently so that each thread has private variables (hence separate states). It is also configurable in terms of the number of supported threads and the shared variables accessible by all threads of the module.

Multiple HT threads are time-division multiplexed on the module by the infrastructure. On every clock cycle, each HT module takes an instruction X from its infrastructure-maintained instruction queues and pushes it into the execution pipelines of variable read, instruction execution, variable write, and thread control. The variable reads and writes are for global, shared, or private variables accessed in X. The execution stage executes the user-programmed operations in X. At the thread-control stage, X can call another module, return to the calling module, pause the current thread until awaken by another thread, continue execution of another instruction, or retry X if the current execution stage failed for any reason (e.g., busy external memory).

```
#define EX_HTID_W 7
typedef ht_uint48 MemAddr_t;
dsnInfo.AddModule(name=addOne, htIdW=EX_HTID_W);
addOne.AddInst(name=EX_LD);
addOne.AddInst(name=EX_ST);
addOne.AddInst(name=EX_RTN);
addOne.AddHostMsg(dir=in, name=OP1_ADDR)
  .AddDst(var=op1Addr);
addOne.AddPrivate()
  .AddVar(type=uint32_t, name=vecIdx)
  .AddVar(type=uint64_t, name=result);
addOne.AddShared().AddVar(type=MemAddr_t, name=op1Addr);
addOne.AddEntry(func=addOne, inst=EX_LD)
  .AddParam(type=uint32_t, name=vecIdx);
addOne.AddReturn(func=addOne)
  .AddParam(type=uint64_t, name=result);
addOne.AddReadMem().AddDst(var=op1);
addOne.AddWriteMem();
```

(a)

```
#include "Ht.h"
#include "PersAddOne.h"
#define BUSY_RETRY(b) {if (b) {HtRetry(); break;}}

void CPersAddOne::PersAddOne() {
  if (PR_htValid) {
    switch (PR_htInst) {
      case EX_LD: BUSY_RETRY(ReadMemBusy());
        ReadMem_op1((MemAddr_t)(S_op1Addr+(P_vecIdx*8)));
        ReadMemPause(EX_ST); break;
      case EX_ST: BUSY_RETRY(WriteMemBusy());
        WriteMem((MemAddr_t)(S_op1Addr+(P_vecIdx * 8)),
                 (P_op1 + 1));
        WriteMemPause(EX_RTN); break;
      case EX_RTN: BUSY_RETRY(SendReturnBusy_addOne());
        SendReturn_addOne(); break;
      default: assert(0);
}}}
```

(b)

Fig. 2: HT language example, addOne, showing (a) HT description (HTD) file, and (b) HT instruction file

**HT Language.** HT modules are programmed using the HT programming language, a subset of C/C++ with configurable runtime libraries of commonly used functions. The coprocessor code consists of two types of source files: HT description (HTD) file and HT instruction files. The HTD file contains the declarations of the modules. It also has definitions of call/messaging interfaces, instructions, and variables (global, shared, or private) for each module. Each module has its own HT instruction file, which contains the module's behavioral code in C/C++ syntax.

In Fig. 2, we show a simple HT code example – addOne. Its HTD file as shown in Fig. 2(a), includes declaration of the module, three instructions (EX_LD, EX_ST, EX_RTN), one input host-messaging interface, two private variables (vecIdx, result), one shared variable (op1Addr), the module entry interface, the module return interface, and the module's memory access interfaces. The HT instruction file of the example code is shown in Fig. 2(b), which includes the programmed operations for each instruction. Three important aspects of the instruction file should be noted: 1) each HT instruction corresponds to a case of the switch statement over the infrastructure-managed PR_htInst variable; 2) the user-defined variables within the HTD file are accessible in each instruction by prefixing the variable name with P_ (for private variable) or S_ (for shared variable); and 3) APIs (e.g., ReadMem_op1, ReadMemPause, etc.) are generated by HT to facilitate thread controls and the usage of memory interfaces and entry/return interfaces. More code examples of HT programs can be found in [13] or the OpenHT repository [7].

**HT design flow.** The HT design flow is shown in Fig. 3. The green-colored blocks are the compiler tools for HT: HT linker (HTL) and HT Verilog generator (HTV). HTL parses the HTD file and generates the runtime libraries in the form of SystemC code, which is then compiled with the HT instruction files and the host code by GCC into an integrated SystemC simulator. Using this SystemC-based simulator, designers can quickly debug a design using familiar techniques (e.g., GDB, printf, or assertions) and conduct design-space exploration using the trace of the cycle-accurate simulation. After the design is verified, the HTV tool is used to generate Verilog files, which are then compiled by the FPGA vendor tools to generate an FPGA image.

Currently, HT only supports platforms that implement the Convey hybrid-core architecture [19]. However, Convey has released the source code of HT [7], allowing anyone to build the toolset and add support for other RC platforms.

## B. *Design-flow Comparison with HDLs*

A typical HDL design flow is shown in Fig. 4. The host code is written in a high-level language such as C/C++ and the coprocessor code is in HDLs (e.g., VHDL, Verilog). Designers are required to write testbench code in HDLs for functional simulation. Additionally, functional co-simulation of both host

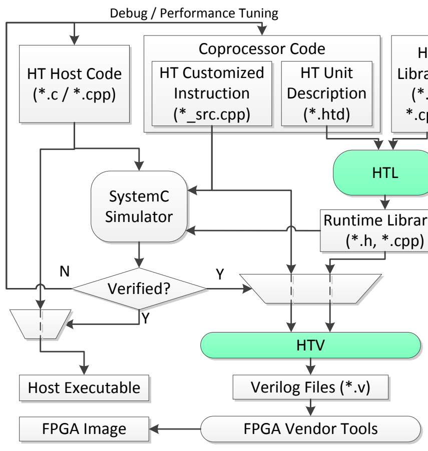
Debug / Performance Tuning
HT Host Code (*.c / *.cpp)
Coprocessor Code
HT Customized Instruction (*_src.cpp)
HT Unit Description (*.htd)
HT Libraries (*.h, *.cpp)
HTL
SystemC Simulator
Runtime Libraries (*.h, *.cpp)
N
Verified?
Y
Y
HTV
Host Executable
Verilog Files (*.v)
FPGA Image
FPGA Vendor Tools

Fig. 3: HT design flow

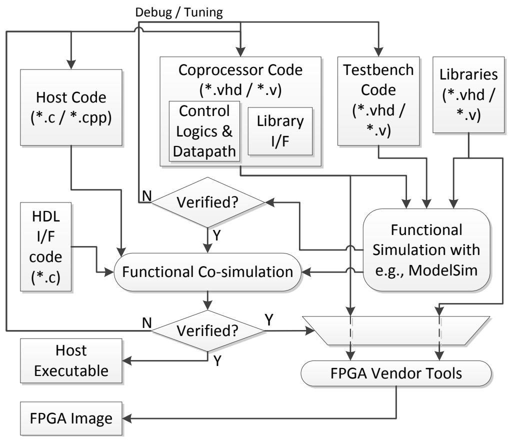

Fig. 4: HDL design flow

and coprocessor code requires designers to develop even more code – e.g., foreign language interface (FLI) code for VHDL or programming language interface (PLI) code for Verilog. Finally, the coprocessor code is compiled by the FPGA vendor tools to produce an FPGA image. To meet performance requirements, the above process is generally repeated to optimize the design.

Comparing the HT design flow with the HDL design flow, we observe several productivity improvements. Firstly, all the source code of an HT design are written in a high-level language, which generally result in less code and faster code modifications than HDLs. Secondly, validating an HT design is much simpler than an HDL design. No HDL testbench or FLI/PLI code is required for HT. Instead, SystemC connects the host and coprocessor code in an integrated-system simulation, in which more comprehensive input data can be used for design validation since SystemC simulation is found to be orders of magnitude faster than HDL simulations (e.g., using ModelSim) [17]. Moreover, cycle-accurate SystemC simulation can be used to conduct design-space exploration, which saves even more design time. Finally, the HT infrastructure provides consolidated FPGA design patterns. For example, memory partitioning is concretized and abstracted in the form of global, shared, and private variables. In comparison, memory partitioning in HDL designs needs much attention of the developer and has proven to be one of the major productivity challenges [9].

In addition to the qualitative productivity analysis above, we also conducted quantitative productivity evaluation of HT. However, a rigorous productivity study involves independent testing of a large number of developers of various backgrounds and skill levels. Conducting such a productivity evaluation is beyond the scope of our work. Instead, we use two commonly used productivity metrics: lines of code (LoC) and anecdotal design time. Design time of the HDL designs was given to us by their developers and we estimate the design time for HT designs. LoC is a simple line count of the source files. The results are summarized in Section V.

## IV. Reference Computing Kernels

In this section, we introduce the selected reference kernels, present their HT designs, and justify our selection of the kernels. The reference computing kernels are the Sobel edge detector from the image processing domain, breadth-first search from the graph processing domain, and Smith-Waterman sequence alignment from the bioinformatics domain.

### *A. Sobel Edge Detector*

The Sobel edge detector is a basic image-processing algorithm. It is based on the gradient calculation of pixel intensities in an image, which is essentially a 2D convolution of the image with two sets of 3-by-3 filters known as Sobel filters:

$$Gx = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, Gy = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

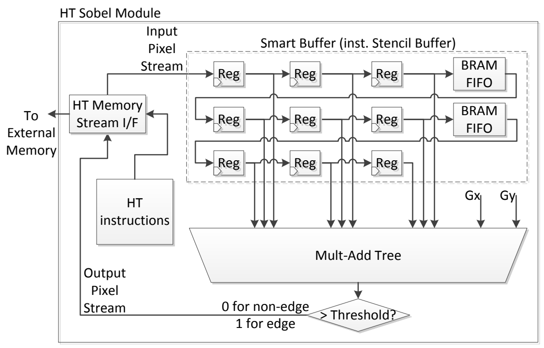

Fig. 5: HT design of Sobel edge detector

We ported the VHDL code of the Sobel edge detector (which was developed for another project in our lab [16]) to our target platform to create the reference HDL design. We follow that design closely to create the HT design, the hardware structure of which is shown in Fig. 5. It has a single HT unit with one module inside. In the module, a smart buffer is created as an instance of the stencil buffer from the HT libraries. HT instructions are defined to create, read from, and write to an HT streaming interface for external memory. The multiply-add tree is also defined using the HT programming language. The Sobel module has 221 lines of HT code (including both the HTD file and the instruction file) as compared to 2078 lines of HDL code.

### *B. Breadth-first Search (BFS)*

Breadth-first search (BFS) is an important building block for graph processing algorithms. It is used in many application domains that require high-performance traversal of large-scale graphs (e.g., social networking). CyGraph is an optimized HDL implementation of the BFS algorithm presented in [4] and is among the best performing BFS implementations published so far. The authors of CyGraph kindly shared the VHDL source code with us so we can use it as our reference HDL design for the BFS kernel.

We created the HT design of BFS following CyGraph as closely as possible. One unit of the design is depicted in Fig. 6. The Master module forks threads onto Kernel and NextEnq modules. Each kernel thread handles one node from the current-level queue. The Kernel module is functionally similar to the kernel in CyGraph, except the process of pushing results into the next-level queue. That process corresponds to the NextEnq module. Similar to CyGraph, we replicate the HT unit 64 times to utilize all memory ports on the target platform.

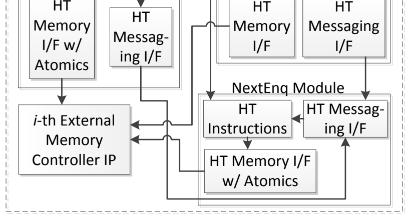

Fig. 6: HT design of BFS kernel

Our HT design differs from CyGraph in three ways. Firstly, explicit FIFOs are not needed by the HT design because the HT infrastructure provides built-in FIFOs in the module-call and messaging interfaces. Secondly, memory management logics of CyGraph (e.g., request multiplexer and responses decoders) are not needed by the HT design because the HT memory interfaces provide simple APIs for memory management. Finally, the version of HT that we use (version 1.3) has a limitation: no messaging interfaces across units. Consequently, we cannot implement the efficient token-ring synchronization mechanism of CyGraph. Instead, we resort to a synchronization mechanism based on memory atomics.

## C. *Smith-Waterman Sequence Alignment*

Smith-Waterman is a fundamental algorithm in the bioinformatics domain which performs local sequence alignment to determine similar regions between two strings of nucleotide or protein sequences. A detailed description of the algorithm can be found in [15].

An optimized Verilog implementation of Smith-Waterman from Convey is overviewed in [10]. It features optimized Verilog code and detailed area constraints using manual placement and routing. Since such detailed constraints are not generally available for HDL-based designs, we recompiled the optimized Verilog code without the manual placement and routing to derive the reference HDL design.

The longest query that the reference HDL design can handle in hardware is 480, limited by the number of PEs. HT's ease of programming for complex control logics enabled us to easily overcome this limitation by processing a long query in segments so that queries of arbitrary lengths can be handled. The hardware structure of the HT design is shown in Fig. 7. Each HT unit (64 units in total) has one Control module and one Query-database module. The Query-database module has 128 threads, each thread handling a pair of query and database sequence. The module also has a pipelined datapath of 15 stages organized into an 8-by-8 array of PEs. The 128 threads in module share the HT memory interface and the 8-by-8 array of PEs. Each thread processes its query and database sequence in segments of 8 protein residues and the database segments are double-buffered to hide the latency of database loads.

The HT design has 1409 lines of code as compared to 11461 lines of HDL. Note that we use for-loops to replicate the PE arrays which are translated into for-generates in HDLs.

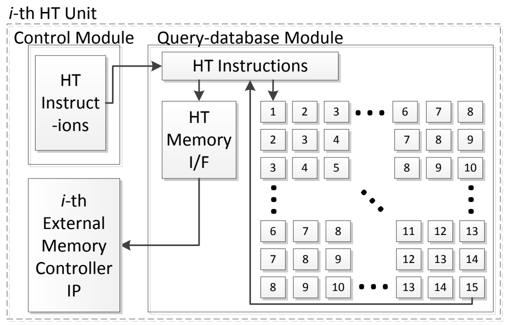

Fig. 7: HT design of Smith-Waterman kernel

## D. *Rationale for Kernel Selection*

The Sobel edge detector is a streaming kernel with limited amount of computation (12 multipliers, 13 adders) and fixed input data rate (one pixel per clock cycle). Such streaming algorithm is amenable for FPGAs and thus serves as a basic test for HLS tools. The BFS algorithm has limited computation but much external memory operations. Such memory-bound algorithm can test an HLS in terms of memory-interface efficiency and the ease of programming external memory operations. Finally, the Smith-Waterman algorithm represents the computation-bound algorithms which require massive parallelism to achieve high performance. Thus, such algorithm can test an HLS's capability to efficiently replicate processing elements. These three types of algorithms represent common computation and communication patterns of many algorithms targeting FPGAs for acceleration and serve as a good test suite for evaluating the efficiency of an HLS tool against corresponding optimized HDL designs.

## V. RESULTS AND DISCUSSIONS

In this section, we first describe the experiment setup. Then, the performance, resource usage, and productivity results of the reference computing kernels are presented and compared between the HT HLS implementations and the HDL ones.

## A. *Experiment Setup*

Our target platform is the Convey HC-2ex system. This platform features two Xeon X5670 processors (12 cores in total) on the host side and four Xilinx Virtex-6 LX760 FPGAs on the coprocessor side. The system has 64GB host memory and 16GB coprocessor memory. HT toolset version 1.3 was used to implement the reference kernels.

Real-world input data were used for the tests. The Sobel kernel's input images can have pixels up to 16 bits wide and three resolutions: 640x480 (VGA), 1280x720 (720p), and 1920x1080 (1080p). The inputs to the BFS kernel are random graphs with 2^23 vertices and N*2^23 edges. N is the average degree of the vertices, taking values of 8, 16, 32, and 64. The Smith-Waterman kernel is tested with the Swiss-Prot protein database, which features 195,014,757 residues in 547,599 sequences. The query sequences (code-name: P07327, P01008, P03435, and Q9UKN1) are selected so that their lengths surround 480, the upper limit of query length of the HDL design. Note that Q9UKN1 has 5478 residues and it is selected to verify that the HT version can handle very long query.

## B. *Performance and Productivity Comparison*

The results of performance, resource utilization, and productivity are summarized in Table 1. The Sobel kernel shows similar performance for both HT and HDL designs, which indicates that HT is excellent for the first type of algorithms (streaming). The HT design uses more resources, which is due to the overhead of the HT infrastructure (e.g., HIF, module wrapper, automatic global/shared variables, etc.) and the HT streaming interface (e.g., aggressive prefetching logics). However, productivity metrics such as lines-of-code (LoC) and design time show an order of magnitude advantage for HT.

Table 1: HT-HDL comparison of performance, resource utilization, and productivity

| | | Performance | | | | | Resource Utilization (number and percentage) | | | | | | | | Freq. (MHz) | Productivity | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sobel | | Frames per second (FPS) | | | | | FF | | LUT | | DSP | | BRAM (Kb) | | | LoC | Design time (hrs) |
| Image resolution | | VGA | 720p | 1080p | | | | | | | | | | | | | |
| Design entries | VHDL | 321 | 122 | 63 | | | 70,434 | 7% | 41,977 | 8% | 22 | 2% | 1789 | 6% | 150 | 2078 | 60 |
| | HT | 312 | 114 | 60 | | | 108,666 | 11% | 86,527 | 18% | 22 | 2% | 2376 | 9% | 150 | 221 | 5 |
| BFS | | Billion traversed edges per second (GTEPS) | | | | | | | | | | | | | | | |
| Data size: node x avg. degree | | 2^23 x 8 | 2^23 x 16 | 2^23 x 32 | 2^23 x 64 | | | | | | | | | | | | |
| Design entries | VHDL | 2.00 | 2.19 | 2.30 | 2.35 | | 127,119 | 13% | 126,525 | 26% | 0 | 0% | 8640 | 33% | 150 | 4273 | 240 |
| | HT | 1.19 | 1.57 | 1.85 | 2.02 | | 201,019 | 21% | 232,097 | 48% | 0 | 0% | 6048 | 23% | 150 | 773 | 40 |
| Smith-Waterman | | Billion cell updates per second (GCUPS) | | | | | | | | | | | | | | | |
| Query name, query length | | P07327, 375 | P01008, 464 | P03435, 567 | Q9UKN1, 5478 | all 4 queries | | | | | | | | | | | |
| Design entries | Verilog | 179.81 | 223.16 | 50.95 | 67.11 | 130.05 | 112,431 | 11% | 147,785 | 31% | 0 | 0% | 2844 | 10% | 110 | 11461 | 350 |
| | HT | 189.23 | 194.70 | 206.23 | 212.25 | 201.17 | 259,874 | 27% | 337,834 | 71% | 16 | 1% | 10656 | 41% | 125 | 1409 | 80 |

For the BFS kernel, the HT design achieves from 60% performance of the HDL design for the smallest graph to 86% performance for the largest graph, indicating good performance for memory-bound algorithms. The main performance overhead is from the atomic memory operations for synchronization among HT units. This overhead is not directly affected by graph size so its percentage of the execution time decreases for larger graphs. Comparing resource utilization, HT uses more flip-flops (FF) and look-up tables (LUT) but fewer block RAMs (BRAM) than HDL. The FF and LUT overhead are due to the HT infrastructure and the atomic memory interfaces. As for BRAM, since the FIFOs within the HT infrastructure are implicitly used, fewer BRAM-based FIFOs are required as compared to the HDL design. Again, the productivity metrics show a great advantage for the HT design.

For Smith-Waterman, the performance of the HDL design drops dramatically for queries that are longer than 480. This is expected since the HDL design cannot fit more than 480 PEs on the FPGA and the application resorts to a software-based alignment instead. The HT design is slower than the HDL design for query P01008 (13%). The performance overhead mainly comes from the additional execution time required to switch from one query segment to the next and the extra memory operations for buffering intermediate results between the switching. However, since the HT hardware works for long queries (>480) as well, HT's overall performance across all four queries surpasses that of the HDL design. Also, the HT design has a higher clock frequency.

Regarding the resource utilization results of Smith-Waterman, in addition to the infrastructure overhead, the HT design required more resources for the following two reasons. Firstly, we were able to fit 1024 PEs (vs. 480) onto the FPGA due to the regular HT unit/module structures. Secondly, extra logics are needed to handle arbitrarily long queries (even longer than the 5478 residues that we tested).

In terms of productivity for Smith-Waterman, HT's LoC number is an order of magnitude smaller than HDL. The design time comparison is less dramatic but the development process of the HDL design involved more than one developer, while a single developer is what we had for all three HT designs. In summary, the Smith-Waterman results indicate that HT also works well for the computation-bound algorithms.

## VI. Conclusions and Future Work

As FPGA technology sees rapid growth in many high-performance computing domains, productivity challenges of the traditional HDL-based design flow become a major obstacle to wide adoption of the technology. HLS provides familiar high-level languages as design entry and tools to compile, verify, and implement designs to greatly increase design productivity. However, HLS commonly incurs overhead in the form of performance and hardware resource.

The Convey HT toolset is a new and unique HLS that features familiar design entry, customizable infrastructure, and fast validation. In this paper, we evaluate HT using designs of three reference kernels (Sobel edge detector, breadth-first search, and Smith-Waterman sequence alignment) by comparing performance, resource usage, and productivity against optimized HDL designs. The results show that HT can achieve over 80% performance of optimized HDL designs for the reference kernels. Although there is non-negligible resource overhead, the dramatic reduction of LoC and design time may justify the tradeoff in many applications.

We plan to explore additional productivity benefits of HT in future work. The focus is on early design-space exploration using HT's SystemC-based simulation.

## Acknowledgment

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant Nos. EEC-0642422 and IIP-1161022.

## References

[1] A. Agarwal, M. C. Ng et al., "A comparative evaluation of high level hardware synthesis using reed–solomon decoder," Embedded Sys. Letters, IEEE, vol. 2, no. 3, pp. 72–76, 2010.

[2] Altera SDK for OpenCL Programming Guide Version 13.0 SP1, 2013.

[3] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, M. Lujan, "An empirical

evaluation of High-Level Synthesis languages and tools for database acceleration," In Proc. 24th FPL, pp. 1-8. 2014.

[4] O.G. Attia, T. Johnson, K. Townsend, P. Jones, J. Zambreno, "CyGraph: A Reconfigurable Architecture for Parallel Breadth-First Search," In Proc. IPDPSW, 2014, pp. 19-23.

[5] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: constructing hardware in a scala embedded language," in Proc. 49th DAC, 2012, pp. 1216–1225.

[6] B. Betkaoui, Y. Wang, D. Thomas, W. Luk, "A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration," in Proc. ASAP, 2012, pp. 8–15.

[7] T. Brewer, OpenHT. https://github.com/TonyBrewer/OpenHT.

[8] A. Cornu, S. Derrien, and D. Lavenier, "HLS tools for FPGA: Faster development with better performance," in Reconfigurable Computing: Architectures, Tools and Applications. Springer, 2011, pp. 67–78.

[9] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," IEEE Trans. on Computer-Aided Design of ICs and Systems, vol. 30, no. 4, pp. 473–491, 2011.

[10] Convey Computer, Convey Computer Smith-Waterman personality. http://www.conveycomputer.com/files/1113/5085/5467/ConveySmithWaterman6202011.pdf

[11] L. Daoud, D. Zydek, and H. Selvaraj, "A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing," in Advances in Systems Science, 2014, pp. 483–492.

[12] B. da Silva, E. D'Hollander, D. Stroobandt, A. Touhafi, "Exploiting High-level Synthesis Tools for High-performance Applications on FPGAs," 15th FEA Research Symposium Faculty of Engineering and Architecture, Ghent, Belgium: UGent, 2014.

[13] G. Edwards, Convey HT overview. http://www.conveycomputer.com/files/7914/0539/0689/Convey_HT_Overview.pdf, 2013.

[14] Tom Feist, Vivado design suite. Technical report, Xilinx, Inc., 2012.

[15] L. Hasan, Z. Al-Ars, "An Overview of Hardware-Based Acceleration of Biological Sequence Alignment," in Computational Biology and Applied Bioinformatics, pp. 187, Dec. 2011.

[16] K. Hill, S. Craciun, A.D. George, H. Lam, "Comparative Analysis of OpenCL vs. HDL with Image-Processing Kernels on Stratix-V FPGA," in 26th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), July 2015.

[17] A. Hoffman, T. Kogel, H. Meyr, "A framework for fast hardware-software co-simulation," In Proc. DATE, pp. 760-765. 2001.

[18] Y. Iskander, C. Patterson, S. Craven, "High-Level Abstractions and Modular Debugging for FPGA Design Validation," ACM Trans. Reconfigurable Technol. Syst. 7, 1, Article 2, February 2014.

[19] B. Klauer, "The Convey Hybrid-Core Architecture," In High-Performance Computing Using FPGAs, pp. 431-451. 2013.

[20] W. Meeus, K. VanBeeck, T. Goedem, J. Meel, and D. Stroobandt, "An overview of todays high-level synthesis tools," Design Automation for Embedded Systems, vol. 16, no. 3, pp. 31–51, 2012.

[21] Mentor Graphics. "ModelSim." 2007.

[22] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on , vol., no., pp.69,70, 23-25 June 2004.

[23] R. Rashid, J.G. Steffan, V. Betz, "Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS," Field-Programmable Technology (FPT), 2014 International Conference on, vol., no., pp.20,27, 10-12 Dec. 2014.

[24] S. Singh, S. Saurav, R. Saini, A. K. Saini, C. Shekhar, A. Vohra, "Comprehensive Review and Comparative Analysis of Hardware Architectures for Sobel Edge Detector," ISRN Electronics, vol. 2014, Article ID 857912, 9 pages, 2014.

[25] D.J. Warne, N.A. Kelson, R.F. Hayward, "Comparison of High Level FPGA Hardware Design for Solving Tri-diagonal Linear Systems," Procedia Computer Science 29: 95-101. 2014.