Accelerating Compute-Intensive Applications with GPUs and FPGAs

Shuai Che†, Jie Li‡, Jeremy W. Sheaffer†, Kevin Skadron†* and John Lach‡
{sc5nf, jl3yh, jws9c, skadron, jlach}@virginia.edu
Departments of Electrical and Computer Engineering‡ and Computer Science†, University of Virginia

***Abstract*—*Accelerators* are special purpose processors designed to speed up compute-intensive sections of applications. Two extreme endpoints in the spectrum of possible accelerators are FPGAs and GPUs, which can often achieve better performance than CPUs on certain workloads. FPGAs are highly customizable, while GPUs provide massive parallel execution resources and high memory bandwidth.**

Applications typically exhibit vastly different performance characteristics depending on the accelerator. This is an inherent problem attributable to architectural design, middleware support and programming style of the target platform. For the best application-to-accelerator mapping, factors such as programmability, performance, programming cost and sources of overhead in the design flows must be all taken into consideration. In general, FPGAs provide the best expectation of performance, flexibility and low overhead, while GPUs tend to be easier to program and require less hardware resources.

We present a performance study of three diverse applications—*Gaussian Elimination*, *Data Encryption Standard (DES)*, and *Needleman-Wunsch*—on an FPGA, a GPU and a multicore CPU system. We perform a comparative study of application behavior on accelerators considering performance and code complexity. Based on our results, we present an application characteristic to accelerator platform mapping, which can aid developers in selecting an appropriate target architecture for their chosen application.

I. Introduction

Difficulties in scaling single-thread performance without undue power dissipation has forced CPU vendors to integrate multiple cores onto a single die. On the other hand, *GPGPU* (general purpose computing on graphics processing units) and FPGA (field-programmable gate array)-based software/hardware co-design are becoming increasingly popular means to assist general purpose processors in performing complex and intensive computations on accelerator hardware. GPUs and FPGAs, together with other accelerators such the vector processors of IBM's Cell [11], *DSPs* (digital signal processors), media processors and network processors, can process work offloaded by the CPU and send the results back upon completion. Accelerators range from general purpose processors optimized for throughput over single-thread performance, through programmable, domain-specific processors optimized for characteristics of a particular application domain, to custom, application specific chips which are possibly implemented with reconfigurable hardware such as FPGAs. Accelerators' vast parallel computing resources and increasingly friendly programming environments make them good fits to accelerate compute-intensive—and especially data-parallel—parts of applications.

∗ Kevin Skadron is currently on sabbatical with NVIDIA Research

Future computer systems will certainly include some accelerators, with the GPU and video processor the most common. Today, accelerators are primarily available as add-in boards. In the future they will probably be located on-chip with the CPU, thus reducing communication overhead.

Different applications place unique and distinct demands on computing resources, and applications that work well on one processor will not necessarily map to another; this is even true for different phases of a single application. Accelerators that are designed independently by different vendors exhibit significant differences in hardware architecture, middleware support and programming models, which causes the processors designed for the same special task to favor differing subsets of applications. For example, programming methodologies range from direct hardware designs for FPGAs, through assembly and domain specific languages, to high level languages supported by GPUs [10]. These are widely different technologies and currently it is unclear which one is best suited to a given task.

Except for FPGAs—and recently GPUs—there is little research on how to use special purpose processors as accelerators for general-purpose computations, how accelerators and tasks map, and—for future heterogeneous multicore chips—which accelerators warrant on-die circuitry. Also, a challenge facing developers is to understand application behavior on different accelerators to determine how to partition the applications into phases that can execute on available accelerators in the most efficient and cost-effective way. To understand these issues first requires an understanding of which application characteristics map well to which accelerators, and what issues arise in an acceleration model of computing. As a first step in advancing our understanding of these issues, this paper studies several different applications on FPGAs and GPUs and compares them to single- and multi-core CPUs.

This work makes the following contributions:

- An understanding of the pros and cons of FPGA hardware platforms and programming models as they compare to GPU platforms.
- An analysis of three diverse applications—*Gaussian Elimination*, *DES*, and *Needleman-Wunsch*—that is not focused on speedup, but rather on those diverse characteristics of their performance that *allow* speedup.
- Our mapping of application characteristics to a preferred platform, taking into account various trade-offs of metrics including programming cost and performance.

II. Related Work

Use of existing accelerators, such FPGAs and GPUs, has demonstrated the ability to speed up a wide range of applications. Examples include image processing [6], data mining [2] and bioinformatics [9] for FPGAs, and linear algebra [12], database operations [7], K-Means [4], AES encryption [19] and n-body simulations [15] on GPUs. Other work has compared GPUs with FPGAs for video processing applications [5], and similarly analyzed the performance characteristics of applications such as Monte-Carlo simulations and FFT [10].

NVIDIA's *Compute Unified Device Architecture*, or *CUDA*, and AMD's *Compute Abstraction Layer*, or *CAL*, are new language APIs and development environments for programming GPUs without the need to map traditional OpenGL and DirectX APIs to general purpose operations. Domain specific parallel libraries, such as a recent scan primitives implementation [18] can be used as building blocks to ease parallel programming on the GPU. On the other hand, FPGA applications are mostly programmed using hardware description languages such as VHDL and Verilog. Recently there has been a growing trend to use high level languages such as *SystemC* and *Handel-C* [8] which aim to raise FPGA programming from gate-level to a high-level, modified C syntax. Calazans et al. provide a comparison of system design using SystemC and VHDL [3]. But there are still some limitations for these languages. For example, Handel-C does not support pointers and standard floating point. In this initial study, we only use VHDL for the FPGA implementations.

III. FPGA and GPU comparisons

A. Platforms Overview

GPUs are inexpensive, commodity parallel devices with huge market penetration. They have already been employed as powerful coprocessors for a large number of applications including games and 3-D physics simulation. The main advantages of the GPU as an accelerator stem from its high memory bandwidth and a large number of programmable cores with thousands of hardware thread contexts executing programs in a single program, multiple data (SPMD) fashion. GPUs are flexible and easy to program using high level languages and APIs which abstract away hardware details. In addition, compared with hardware modification in FPGAs, changing functions is straightforward via rewriting and recompiling code, but this flexibility comes at a cost.

Compared to the fixed hardware architecture of the GPU, FPGAs are essentially high density arrays of uncommitted logic and are very flexible in that developers can directly steer module-to-module hardware infrastructure and trade-off resources and performance by selecting the appropriate level of parallelism to implement an algorithm. In the FPGA co-processing paradigm, the hardware fabric is used to approximate a custom chip, i.e. an *ASIC* (application specific integrated circuit). This eliminates the inefficiencies caused by the traditional von Neumann execution model and the pipelined implementations of GPUs and CPUs, and can achieve vastly improved performance and power efficiency. Though vendors provide IP cores that offer the most common processing functions, programming in VHDL or Verilog and creating the entire design from scratch is a costly and labor intensive task.

B. CUDA and the GeForce 8800 GTX GPU

CUDA is an extension of C and an associated API for programming general purpose applications for all NVIDIA's *Tesla*-architecture GPUs, including their GeForce, Quadro, and Tesla products. CUDA has the advantage that is does not require programmers to master domain-specific languages to program the GPU. In CUDA, the GPU is treated as a co-processor that executes data-parallel kernels with thousands of threads. Threads are grouped into *thread blocks*. Threads within a block can share data using fast shared-memory primitives and synchronize using hardware-supported barriers. Communication among thread blocks is limited to coordination through much slower global memory. Note that the programming model for a CUDA kernel is *scalar*, not vector. The current Tesla architecture combines 32 scalar threads into SIMD groups called *warps*, but the programmer can treat this as a performance optimization rather than a fundamental aspect of the programming model, similar to optimizing for cache line locality.

The NVIDIA GeForce 8800 GTX GPU is comprised of 16 *streaming multiprocessors* (SMs). Each SM has 8 *streaming processors* (SPs), with each group of 8 SPs sharing 16 kB of per-block shared memory [13] (a private scratchpad in each SM). Each SP is deeply multithreaded, supporting 96 *co-resident* thread contexts with zero-overhead thread scheduling.

C. VHDL and the Xilinx Virtex-II Pro FPGA

VHDL is one of the most widely used hardware description languages. It supports the description of circuits at a range of abstraction levels varying from gate level netlists up to purely algorithmic behavior [3]. Very efficient hardware can be developed in VHDL but it requires a great deal of programming effort.

FPGAs consist of hundreds of thousands of programmable logic blocks and programmable interconnects that can be used to create custom logic functions, and many FPGA products also include some hardwired functionality for common functions. For example, the Xilinx Virtex II Pro FPGA also integrates up to two 32-bit RISC PowerPC405 cores.

D. Application Domains

A technical report from Berkeley [1] argued that successful parallel platforms should strive to perform well on 13 classes of problems, which they termed *dwarves*. Each dwarf represents a set of algorithms with similar data structures or memory access patterns. By examining applications from different dwarves, we can find common characteristics of applications from that dwarf on a specific hardware platform. We chose three applications from three different dwarves.

Our first application, Gaussian Elimination, comes from the *Dense Linear Algebra* dwarf. The applications in this dwarf

use strided memory accesses to access the rows and columns in a matrix [1]. Gaussian Elimination computes result row-by-row, solving for all of the variables in a linear system. The algorithm must synchronize between iterations, but the values calculated in each iteration can be computed in parallel.

DES is a member of the *Combinational Logic* dwarf. Applications in this dwarf are implemented with bit-level logical functions [1]. DES is a cryptographic algorithm, making heavy use of bit-wise operations. It encrypts and decrypts data in groups of 64-bit blocks, using a 64-bit key. For encryption, groups of 64-bit blocks of plaintext are fed into the algorithm to produce groups of 64-bit blocks of ciphertext. This application exhibits massive bit-level parallelism.

Our third application is the Needleman-Wunsch algorithm, which is a representative of the *Dynamic Programming* dwarf. Needleman-Wunsch is a global optimization method for DNA sequence alignment. The potential pairs of sequences are organized in a 2-D matrix. The algorithm fills the matrix with scores, which represent the value of the maximum weighted path ending at that cell. A traceback process is used to find the optimal alignment for the given sequences. A parallel Needleman-Wunsch implementation processes the score matrix in diagonal strips from top-left to bottom-right.

In our GPU implementations, we implement the data-parallel portions of these applications by assigning each thread the task of processing one data point. These threads are independent and can execute in parallel. Datapoints in each iteration of Gaussian Elimination, each permutation of DES, or each diagonal strip of a score matrix in Needleman-Wunsch can be all processed simultaneously. Our implementations of these algorithms on FPGAs and GPUs are similar; we use the same algorithms, and do not employ any optimizations on one unless we can also employ it on the other. In Needleman-Wunsch and Gaussian Elimination, our FPGA implementation uses IP cores for floating point operations. In those cases, floating point multiplication requires 6 cycles and division 24 cycles. We will make the codes used in this study available online at http://lava.cs.virginia.edu/wiki/rodinia.

Our code for the FPGA implementations is in continuing development. So far, we are not using all the FPGA die area. This must be taken into consideration when evaluating the quantitative results in this paper. On the other hand, CUDA implementations leave large portions of the GPU hardware – the rendering specific portions – idle.

E. Methodology and Experiment Setup

Our experiments are performed on representative commercial products from both of the GPU and FPGA markets. The GPU is an NVIDIA GeForce 8800 GTX (128 stream processors clocked at 575 MHz with 768 MB of GPU device memory and a 16 kB per-block shared memory per SM) with NVIDIA driver version 6.14.11.6921 and CUDA 1.1. The FPGA is a Xilinx Virtex-II Pro, which is based on a 130 nm process, clocked at 100 MHz. We also compare with a multicore CPU based system with an Intel Xeon processor (3.2 GHz with two hyperthreaded dual-cores, 4 GB main memory and 2 MB L2).

We developed our GPU code using NVIDIA's CUDA API. Our FPGA code was developed in VHDL under Xilinx ISE 9.2i, and our multithreaded CPU code was compiled with the Intel C compiler, ICC, version 9.1 and uses OpenMP. In all cases, we made no specific effort to tune our implementations to reduce cycle-counts. All the results are compared in terms of cycle counts, eliminating scaling and frequency issues. Note that we are not comparing the theoretical program cycles, but values returned by performance counters via library functions (such as the `clock()` entry point provided by the CUDA API). We measure the total cycle counts from the beginning to the end of program execution on the accelerators, thus offload overhead is not included.

We use cycle count as a metric to examine what application characteristics map well to FPGAs or GPUs. However, in reality, we must take clock frequency into account when choosing accelerators, because they generally run under different frequencies which will determine how fast an application runs. For example, the Xilinx Virtex-II Pro FPGAs can run at a frequency of 100MHz, the NVIDIA Geforce 8800 GTX GPU has a system clock frequency of 575MHz (the SMs process instructions at 1.35GHz) and the Intel Xenon CPU we used is running at 3.2GHz. The Virtex-II FPGA is not as state-of-the-art as the other processors. Given these clock frequencies, the GPU is generally fastest on our computations, followed by the CPU, with the FPGA being slowest. However, we focus on clock cycles to understand organizational tradeoffs, to reduce sensitivity to timing closure on the FPGA, and because at the time of this preliminary study, we only had a Virtex-II FPGA fabric available.

IV. Parallel Programmability

A. Parallel Execution

The powerful compute capabilities of FPGAs and GPUs stem from their vast available parallelism. Of course, programming gates is quite different from programming microprocessors, domain-specific or otherwise. In CUDA, developers write programs with a C-like language. CUDA is currently best suited for a SPMD programming style in which threads execute the same kernel but may communicate and follow divergent paths through that kernel.

On the other hand, the dataflow of an application is exploited in FPGAs through parallelism and pipelining. Designers have the flexibility to trade-off performance for resources. For example, in massively parallel algorithms, hardware programmers might duplicate the same functional units many times, with only the die area limiting the level of parallelism.

B. Synchronization

Placing barriers in CUDA is quite different from using similar such synchronization mechanisms in FPGA hardware. CUDA's runtime library provides programmers with a specific barrier statement, `syncthreads()`, but the limitation of this function is that it can only synchronize all the threads

within a thread block. To achieve *global barrier* functionality, the programmer must allow the current kernel to complete and start a new kernel. This is currently fairly expensive, thus rewarding algorithms which keep communication and synchronization localized within thread blocks as long as possible.

In the case of the FPGA, it is flexible enough to implement different types of barriers—including counter- and tree-based varieties—directly in hardware. Fine-grained synchronization is also feasible so that execution units need only be synchronized with a select set of threads. Additionally, programmers can apply multiple clock sources with different frequencies; however, this imposes a heavy programming burden on developers.

C. Reduction

Most parallel algorithms involve reduction steps, which condense partial results produced by individual threads. For example, in calculating a running sum serially, a multiply-accumulate operation loops over the data, but a reduction can be done in parallel. In a fast and aggressive FPGA implementation, as Figure 1 illustrates, this operation is built with a cascade of adders of depth of $\log(N)$. Figure 1 shows that the number of working threads reduces in half in each iteration of a GPU implementation of a reduce which requires $\log(N)$ iterations. Unlike the traditional, sequential way of programming, on both of these platforms programmers have to deal with complex parallel reductions to best utilize the available parallel computing resources.

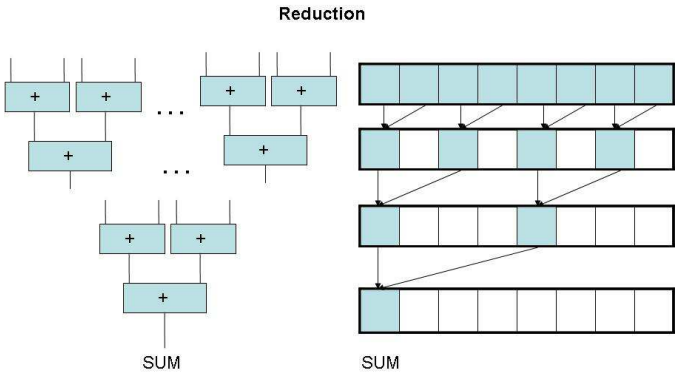

Fig. 1. Reduction on an FPGA and a GPU

V. Hardware Constraints

A. Communication Overhead

Some accelerators work in separate memory spaces from their control processors, with data communication achieved via copy operations. This is different from OpenMP's shared memory programming model, in which programmers are not required to deal with data movement explicitly. For instance, GPUs have their own off-chip device memory, and data must be transferred from main memory to device memory via the PCI-Express bus. The data-copy latency increases linearly with the size of the transferred data, which has the potential to adversely impact overall performance. Our measurements show that an 8 MB data transfer costs about 5ms. However, using CUDA's streaming interface, programmers can batch kernels that run back to back, and increase program execution efficiency by overlapping computations with data communication. FPGAs have similar issues. In addition, the reconfiguration overhead for FPGAs also needs to be taken into account. The reconfiguration (or initialization) process generally takes on the order of seconds for our applications on the FPGA boards. To reduce the impact of this overhead, developers can potentially overlap reconfiguration with other non-conflicting tasks. Furthermore, users may not need to reconfigure after initialization, in which case the configuration is represented by a small, one time cost. GPUs do not have this issue.

B. Control Flow

In GPUs, control flow instructions can significantly impact the performance of a program by causing threads of the same 32-thread SIMD warp to diverge. Since the different execution paths must be serialized, this increases the total number of instructions executed [14]. For example, in a DES sequential program, the programmer can use `if` statements to specify permutations, but directly using the similar implementation on the GPU performs poorly. Figure 2 demonstrates the comparisons of permutation performance on three typical data sizes in DES. The figure compare `if` statements, lookup tables, and no permutations. For 64-bit data, using `if` statements can degrade performance about $5.5\times$ when compared with lookup tables. This is clearly a case where SIMD behavior within warps is important.

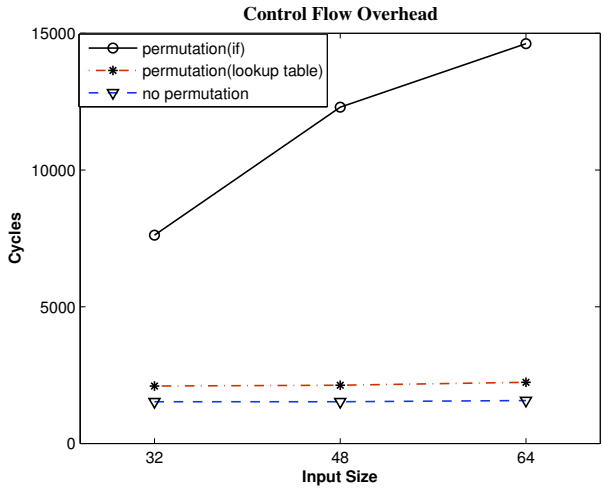

Fig. 2. Overhead of control flow in our CUDA implementation of DES.

C. Bit-wise Operations

Although it is usually easier to data-parallel algorithms on the GPU, FPGAs can sometimes be very useful, and provide

efficient methods to implement some specific functions, such as bit-wise operations, which are not well supported on GPU hardware. Bit-wise operations dominate the DES algorithm, which includes a lot of bit permutations and shifting. GPU implementations of these operations become very complex, since they must be built using high-level language constructs and fine-grained control flow that tend to cause SIMD divergence. In VHDL, bit-wise operation is very straightforward and the bit permutation for any width data can be done in parallel and finished in one cycle! Note that there is nothing preventing GPUs from extending their instruction sets to support bit-wise operations, assuming that general-purpose applications or new graphics algorithms can justify the datapath costs.

D. Floating Point Operations

One of the most important considerations, especially for scientific computing, is the accelerator's ability to support floating point operations. This is a clear-cut issue for GPUs, as G80 GPUs implement IEEE-754 single-precision [14] (with double-precision coming in the next generation), and AMD's FireStream 9710 GPUs already support double-precision. FPGAs, on the other hand, usually have no native support for floating point arithmetic and many applications use fixed point implementations for ease of development. FPGA developers must use on-chip programmable resources to implement floating point logic for higher precisions [17], but these implementations consume significant die-area and tend to require deep pipelining to get acceptable performance; double precision adders and multipliers typically have 10–20 pipeline stages, and square root requires 30–40 stages [16]. Of course, there is no intrinsic reason that FPGAs cannot support floating-point operations, just as FPGAs already provide some other dedicated arithmetic units.

VI. Performance

Our results for our first application, Gaussian Elimination, are illustrated in Figure 3. For all input sizes, both FPGAs and GPUs show their advantages over CPUs, leveraging their parallel computing capabilities. Specifically, for a 64×64 input size, 2.62×10^5 cycles are needed by the FPGA compared to 7.46×10^5 cycles for the GPU. The single-threaded CPU version required 3.15×10^6 cycles, with the four-threaded OpenMP version requiring 9.45×10^6 cycles, differences of about an order of magnitude. The four-threaded version outperforms single-threaded version when the data size becomes large. The major overhead of GPUs and CPUs comes from executing instructions which rely on memory accesses (300–400 cycles), while the FPGA can take advantage of dataflow streaming, which saves many of the memory accesses. The drawback of the FPGA, of course, is the complexity of programming using VHDL. A controller is implemented in VHDL to take care of the controlling signals for all the computation processes. This involves a much higher level of complexity than the CUDA implementation.

The DES encryption results are even more interesting. To process a single, 64-bit block on our FPGA requires only

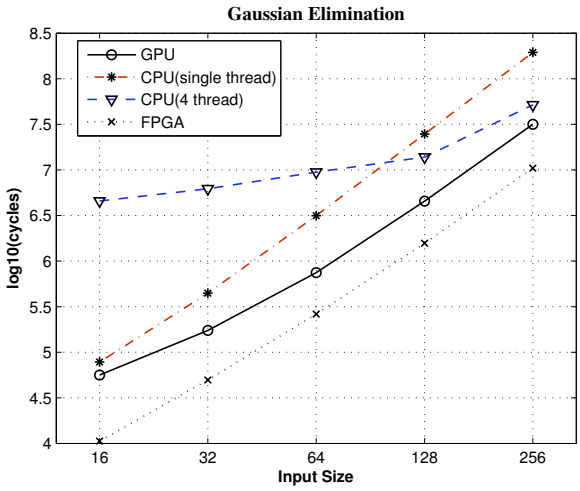

Fig. 3. Execution cycles of the four versions of Gaussian Elimination. The x-axis represents the dimensions of the computation grid. Note that the y-axis is a log scale.

83 cycles, while the same operation executed on the GPU requires 5.80×10^5 cycles. While the GPU does not support some important operations for this application, the main reason for this disparity is that the GPU requires full utilization to take advantage of the hardware's latency hiding design, and this example far underutilized the processor, while the FPGA implementation has no such requirement. As discussed in Section V-C, FPGAs can finish bit-wise calculations in one cycle, and the 64 bit data size is small enough that all intermediate data can be saved in flip-flops, but the GPU involves significant overhead, including memory access times of 300–400 cycles—the permutation lookup tables are all loaded from GPU device memory—and synchronization.

Our CUDA DES implementation is comprised of several small kernels. Inside a CUDA kernel, we can use fast on-chip shared memory for data sharing, but there is no consistent state in the shared memory, so between two kernels, state flushes are required, then data must be read back for the next pass. The performance gap between the GPU and FPGA will be much smaller when we try larger data sizes which require the FPGA to accesses external memory, but due to the complexity of the implementation, we have not completed it yet. We plan to extend it in our future work. We do still expect the gap to be significant, however, since the FPGA does have access to fast bit-wise operations.

The third application is the Needleman-Wunsch algorithm. This is a memory intensive application, but not all of the data in the computational domain are involved in the parallel computation. This algorithm processes data in a strip-wise, in-order fashion, and in each iteration, only a single strip of diagonal elements can be processed in parallel. Figure 4 shows that our FPGA implementation again has the lowest overhead. One observation is that when data size is small, both the four-threaded OpenMP version and the GPU version take more cycles than the single-threaded version, but as the data

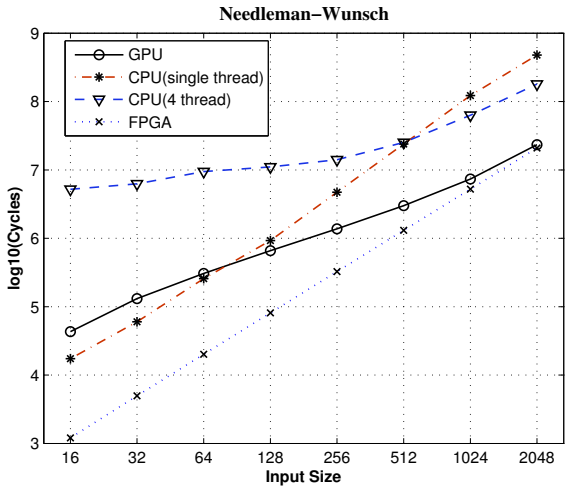

Fig. 4. Execution cycles of the four versions of Needleman-Wunsch. Note that the y-axis is a log scale.

becomes large, both of them outperform the single-threaded CPU version. For a 64×64 input size, the GPU takes 3.0×10^5 cycles while the FPGA takes only 2.0×10^4 cycles, but as the input size increases, the ratio of the GPU execution cycles over FPGA execution cycles becomes smaller, probably attributable to better GPU utilization.

VII. Development Cost

In general, programming in a high level language on a GPU is much easier than dealing with hardware directly with an FPGA. Thus in addition to performance, development time is increasingly recognized as a significant component of overall effort to solution from the software engineering perspective.

We measure programming effort as one aspect in the examination programming models. Because it is difficult to get accurate development-time statistics for coding applications and also to measure the quality of code, we use *Lines-of-Code* (LOC) as our metric to estimate programming effort. Table I shows LOC for the four applications written in CUDA on the GPU platform and written in VHDL on the FPGA platform. CUDA programs consistently require fewer LOC than the VHDL versions (and in the case of DES, the VHDL version is incomplete). This suggests that for a given application it probably requires more control specification to design hardware VHDL than to program equivalently functional software. This implies that GPU application development has better programmability in general, and can be very time-efficient compared with FPGA development. However, high-level languages, like Handel-C, and data-flow languages also exist for programming FPGAs. This is a limitation of our work and an important area of study for future work. Also note that VHDL is more verbose than Verilog and the LOC result should be interpreted in this light.

VIII. Mapping Applications to Accelerators

In this work, we developed three applications, *Gaussian Elimination*, *DES*, and *Needleman-Wunsch* on both an FPGA and a GPU. We compared these two platforms as well as a multicore CPU using several metrics. Based on our results, we present a rough categorization to propose suggestions as to which platform—GPU or FPGA—to which an application best maps. When composing this list (Figure 5), we have tried to consider as many factors as possible, including programming cost, performance, and hardware constraints.

↓Application\Domain→	VHDL	CUDA
Gaussian Elimination	450	160
DES	1400	1100
Needleman-Wunsch	650	320

TABLE I
Development cost measured by length of the code.

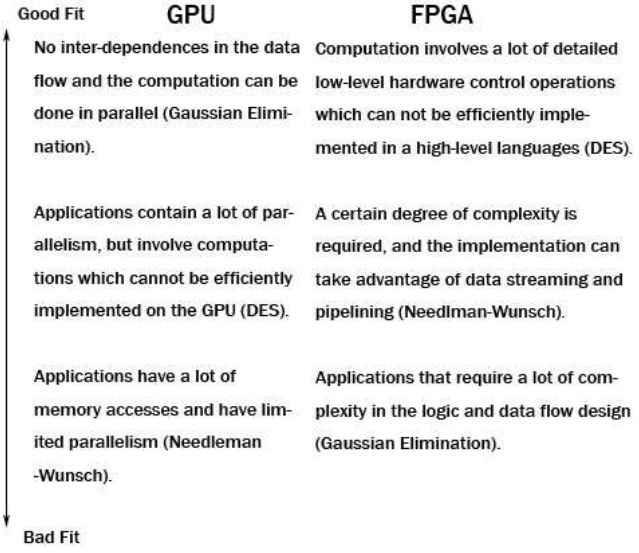

Fig. 5. Application Characteristic Fitness Categorization.

IX. Conclusions and Future Work

In this work, taking FPGAs and GPUs as case studies, we have developed three applications on each platform—as well as single- and multi-core CPU implementations—in an attempt to gain insights into how well applications will map onto various accelerators. We consider many factors, including overall hardware features, application performance, programmability and overhead, and more importantly how to trade off among these factors, and we evaluate the pros and cons of FPGAs and GPUs.

The current comparison is primarily qualitative, due to a limited set of benchmarks for comparison. A methodology for a quantitative comparison, especially one that distinguishes between fundamental organizational limits versus easily changed features, is an important area for further research. A more direct comparison might be between CUDA and a high level language for FPGAs. We plan to extend this work to examine more applications on more accelerators, including DSPs and

network processors. In this paper, we discuss the problem from a performance point of view. We also plan to include new metrics which take into account power consumption and chip area, that potentially determine the best choice of platform. Also, both FPGAs and GPUs are likely to migrate closer to the main CPU in future architectures. Issues related to this migration are another interesting areas for future work.

ACKNOWLEDGEMENTS

This work has been supported in part by NSF grant nos. IIS-0612049 and CNS-0615277, and grants from Intel MTL and NVIDIA Research. We would like to thank David Tarjan for his helpful advice, and the anonymous reviewers for their excellent suggestions on how to improve the paper.

REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] Z. K. Baker and V. K. Prasanna. Efficient hardware data mining with the Apriori algorithm on FPGAs. In *Proceedings of the 13th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 3–12, 2005.

[3] N. Calazans, E. Moreno, F. Hessel, V. Rosa, F. Moraes, and E. Carara. From VHDL register transfer level to SystemC transaction level modeling: A comparative case study. In *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design*, page 355, 2003.

[4] S. Che, J. Meng, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphics processors. In *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[5] B. Cope, P. Y. K. Cheung, W. Luk, and S. Witt. Have GPUs made FPGAs redundant in the field of video processing? In *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology*, pages 111–118, 2005.

[6] B. de Ruijsscher, G. N. Gaydadjiev, J. Lichtenauer, and E. Hendriks. FPGA accelerator for real-time skin segmentation. In *Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pages 93–97, 2006.

[7] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 International Conference on Management of Data*, pages 215–226, 2004.

[8] Handel-C. Web resource: http://www.celoxica.com/.

[9] B. Harris, A. C. Jacob, J. M. Lancaster, J. Buhler, and R. D. Chamberlain. A banded Smith-Waterman FPGA accelerator for Mercury BLASTP. In *Proceedings of the 2007 International Conference on Field Programmable Logic and Applications*, pages 765–769, 2007.

[10] L. W. Howes, P. Price, O. Mencer, O. Beckmann, and O. Pell. Comparing FPGAs to graphics accelerators and the Playstation 2 using a unified source description. In *Proceedings of the 2006 International Conference on Field Programmable Logic and Applications*, pages 1–6, 2006.

[11] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.,49(4/5):589C604*, 2005.

[12] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, 2003.

[13] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[14] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[15] L. Nyland, M. Harris, and J. Prins. Fast N-Body simulation with CUDA. *GPU Gems 3*, 2007.

[16] R. Scrofano, G. Govindu, and V. K. Prasanna. A library of parameterizable floating-point cores for FPGAs and their application to scientific computing. In *Proceedings of the 2005 International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 137–148, 2005.

[17] R. Scrofano and V. K. Prasanna. A hierarchical performance model for reconfigurable computers. *Handbook of Parallel Computing: Models, Algorithms and Applications*, 2008.

[18] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Proceedings of the Graphics Hardware 2007*, pages 97–106, 2007.

[19] T. Yamanouchi. AES encryption and decryption on the GPU. *GPU Gems 3*, July 2007.