Bridging the GPGPU-FPGA Efficiency Gap

Christopher W. Fletcher[†‡], Ilia Lebedev[‡], Narges B. Asadi[≀], Daniel R. Burke[‡], John Wawrzynek[‡]
† CSAIL, Massachusetts Institute of Technology; Cambridge, MA, USA
‡ EECS Dept., University of California, Berkeley; Berkeley, CA, USA
≀ EE Dept., Stanford University; Stanford, CA, USA
cwfletcher@csail.mit.edu, {ilial,drburke,johnw}@berkeley.edu, nargesb@stanford.edu

ABSTRACT

This paper compares an implementation of a Bayesian inference algorithm across several FPGAs and GPGPUs, while embracing both the execution model and high-level architecture of a GPGPU. Our study is motivated by recent work in template-based programming and architectural models for FPGA computing. The comparison we present is meant to demonstrate the FPGA's potential, while constraining the design to follow the microarchitectural template of more programmable devices such as GPGPUs.

The FPGA implementation proves capable of matching the performance of a high-end Nvidia Fermi-based GPU—the most advanced GPGPU available to us at the time of this study. Further investigation shows that each FPGA core outperforms workstation GPGPU cores by a factor of $\sim 3.14\times$, and mobile GPGPU cores by $\sim 4.25\times$ despite a $\sim 4\times$ reduction in core clock frequency. Using these observations, we discuss the efficiency gap between these two platforms, and the challenges associated with template-based programming models.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems

General Terms

Design, Performance, Algorithms

Keywords

FPGA, GPGPU, OpenCL, Reconfigurable Computing, Bayesian Networks

1. INTRODUCTION

FPGA designs are often highly specialized for their application, employing custom architectures and a variety of execution models [6, 8, 9, 12]. Conversely, GPGPU (general-purpose graphics processing unit) approaches follow well-defined programming models such as CUDA and OpenCL [13, 14]. The GPGPU's execution model is rigid, causing applications that do not map well to the GPGPU architecture to perform relatively poorly. FPGAs, on the other hand, can adapt to the natural, optimal, and sometimes arbitrary computation and architectures required by specific applications. For this reason, FPGAs make a compelling platform for targeting classes of applications that do not easily map to standard GPGPU methods of execution.

It is less clear whether FPGAs have value as targets for applications that *do* match a GPGPU in execution model and map well to a GPGPU architecture. This paper considers one such application, Bayesian inference, and characterizes it on an FPGA using computation patterns akin to OpenCL. To this end, we present a custom FPGA implementation that naturally resembles an application-specific GPGPU at a high level, but differs in its implementation of the arithmetic modules ("cores") and memory access scheduling. Specifically, this paper first contributes a new approach to Bayesian inference on FPGAs and GPGPUs which allows us to characterize both implementations using an OpenCL-like execution model. Secondly, we perform a study examining algorithm kernel performance on FPGA and GPGPU platforms, and normalize core performance across devices to help explain differences in their observed efficiency.

We have conducted this research alongside related work in developing microarchitectural template-based programming models for FPGA computing [11]. Both the GPGPU and FPGA designs used in this work follow the same execution model and feature similar many core-based architectures. The FPGA implementation, however, was designed manually at the RTL level, while the GPGPU implementation was described at a high level with OpenCL. Related work proposes automatic mappings from CUDA and OpenCL flows to the FPGA [7, 10]. We intentionally avoided automatic hardware generation in this work to (a) evaluate the FPGA as an implementation target for microarchitectural template-based programming models, and (b) establish and upper-bound on template-based FPGA performance.

1.1 Bayesian Inference

Bayesian networks (BNs) are graph-based models that have numerous applications in bioinformatics, finance, signal processing, and computer vision. Bayesian inference is the process by which a BN's graph structure is learned from a set of quantitative data, or "evidence," that the BN seeks to model. Once a BN's structure (a set of nodes $\{V_1, \ldots, V_{\mathcal{N}}\}$) is determined, and the conditional dependence of each node V_i on its parent set Π_i is tabulated, the joint distribution over the nodes can be expressed as:

$$P(V_1, \ldots, V_{\mathcal{N}}) = \prod_{i=1}^{\mathcal{N}} P(V_i|\Pi_i)$$

Despite significant recent algorithmic advances, BN inference from evidence is an NP-hard problem and remains infeasible except for cases with only a small number of variables [3, 5].

The algorithm surveyed in this paper is the union of two BN inference kernels, the order[1] and graph samplers (jointly called the *order-graph sampler*). The order sampler takes a BN's prior evidence ($\mathcal{D}$ or *data*), along with an initial order to use as a starting point, and produces a set of "high-scoring" BN orders (orders that best explain the evidence). The graph sampler takes this set of high-scoring orders and produces a single "highest-scoring" graph for each order.

We generate the prior evidence $\mathcal{D}$, which consists of $\mathcal{N}$ sets of $\mathcal{P}$ local score/parent set pairs, prior to invoking the order-graph sampler. Following [4, 5, 12], the order-graph sampler itself uses Markov chain Monte Carlo (MCMC) sampling to perform an iterative random walk in the space of BN orders—until the order score has converged. Each step in the random walk (1) breaks the current order into $\mathcal{N}$ disjoint "local orders," and (2) iterates over each node's parent sets, accumulating each local score whose parent set is *compatible* with the node's local order. To decrease time to convergence and to increase confidence in the results, $\mathcal{O}$ distinct orders can be dispatched though a technique known as parallel tempering, coupled with random restarts.

Computationally, the order-graph sampler is a compute intensive set of nested loops (Algorithm 1) with an innermost kernel called the *score* function, given between lines 8 and 19. To put the number of loop iterations into perspective, typical parameter values for $\{\mathcal{I}, \mathcal{O}, \mathcal{N}\}$ are $\{10000, 512, 37\}$, where $\mathcal{I}$ is the number of MCMC iterations. Furthermore, $\mathcal{P} = \sum_{i=0}^{4} \binom{\mathcal{N}-1}{i}$ and $|\mathcal{D}| = \mathcal{N} * \mathcal{P}$.

Different degrees of parallelism and data locality can be exploited in the loops within Algorithm 1. Traditionally, the loop over $\mathcal{O}$ was placed outside the loop over $\mathcal{I}$ as $\mathcal{O}$ contributed course-grained and dependency-free parallelism [12, 14]. In this work, we have re-ordered the loops, moving the $\mathcal{O}$ dispatches to within the loop over $\mathcal{N}$, reducing the communication requirement of the algorithm, and relaxing loop dependencies. We classify the reformulated loop nest as compute-intensive because of the relatively small amount of input (a local order) needed for the inner-loop arithmetic to compute per-node results.

1.2 The OpenCL Execution Model

OpenCL [1] is a programming and execution model for heterogeneous systems (containing GPPs (general-purpose processors), GPGPUs, FPGAs [10] and other accelerators) designed to explicitly capture data and task parallelism in an application. The OpenCL model distinguishes control thread(s) (to be executed on a GPP host) from kernel threads (data parallel loops to be executed on a GPGPU, or similar accelerator). The user specifies how the kernels map to an n-dimensional dataset, given a set of arguments (such as constants or pointers to device/host memory). The runtime then distributes the resulting workload across available compute resources on the device. Communication between control and kernel threads is provided by shared memory and OpenCL system calls such as barriers and bulk memory copy operations. With underlying SIMD principles, OpenCL is well-suited for data-parallel problems, and maps well to the parallel thread dispatch architecture found in GPGPUs.

[1]An order is a graph whose nodes have been topologically sorted (each node is placed after its parents).

Algorithm 1 The order-graph sampler loop nest. ps, ls, and $o[n]$ are parent set, local score, and local order, respectively.

```
    for $i$ in $\mathcal{I}$ do
      Setup (partition $\mathcal{O}$ orders into $\mathcal{O} * \mathcal{N}$ local orders)
      for $n$ in $\mathcal{N}$ do
        for $o$ in $\mathcal{O}$ do
 5:         $s_o, s_g \leftarrow -\infty$
            $g \leftarrow$ NULL
            for $p$ in $\mathcal{P}$ do
              if $compatible(\mathcal{D}[n][p].ps, o[n])$ then
                $d \leftarrow \mathcal{D}[n][p].ls - s_o$
10:             if $d \geq$ HIGH_THRESHOLD then
                  $s_o \leftarrow \mathcal{D}[n][p].ls$
                else if $d >$ LOW_THRESHOLD then
                  $s_o \leftarrow s_o + log(1 + exp(d))$
                end if
15:             if $\mathcal{D}[n][p].ls > s_g$ then
                  $s_g \leftarrow \mathcal{D}[n][p].ls$
                  $g \leftarrow \mathcal{D}[n][p].ps$
                end if
              end if
20:         end for
            Teardown (post-process $s_o$, $s_g$, and $g$)
          end for
        end for
    end for
```

2. IMPLEMENTATION

The order-graph samplers on both the FPGA and GPGPU are many core-based systems that map instances of the *score* function over a two-dimensional space (given by $\mathcal{O} \times \mathcal{N}$) according to the OpenCL model. In this work, GPGPU cores correspond to Nvidia CUDA cores while FPGA cores are custom datapaths implementing the BN scoring kernel.

2.1 FPGA

The FPGA implementation is composed of compute cores that are paired with block RAMs (BRAMs). Each core iterates over a disjoint subset of the *score* calls, while the BRAMs are responsible for caching and streaming the data needed by those *score* calls. All *score* function arithmetic is built directly into the FPGA fabric to eliminate resource contention. We implemented the $log(1 + exp(d))$ function (Algorithm 1) as a table lookup, given that it is non-linear over the narrow range of d that it is called. To maximize the read bandwidth and throughput achievable by the BRAMs, cores are replicated to the highest extent possible and fine-grain multi-threaded across multiple iterations of the *score* function.

To balance logic and BRAM utilization, each node's *score* operation is mapped onto multiple cores which run in parallel and accumulate results upon completion. This technique reduces parallel completion time, yet requires increased complexity to accumulate partial node scores at the end of the *score* loop. To maximally hide this overhead, we chain cores together, using a dedicated interconnect, and interleave the cross-core partial result accumulation process with the next local order's scoring period. To simplify the accumulation logic, we linearly reduce all threads across a core, and then accumulate linearly across cores.

To better understand FPGA performance, the following analytic model can be used to estimate the number of FPGA core cycles needed to score $\mathcal{O} * \mathcal{N}$ local orders over subset S of the data (where S is scored for each of the $\mathcal{N}$ nodes):

$$\mathcal{N} * \left[\mathcal{O} * \left(\frac{|\mathrm{S}|}{\mathrm{C}} + (\mathrm{T}+1)\right) + (\mathrm{T}^2 + \mathrm{T} * \mathrm{C}) + \mathrm{Cycles}_{\mathrm{DRAM}}\right]$$

$\mathrm{Cycles}_{\mathrm{DRAM}}$ is the number of cycles (normalized to the core clock) required to page S into the cores' BRAMs, C is the number of cores assigned to subset S, and T is the number of hardware threads per core. The $\mathrm{T}+1$ and $\mathrm{T}^2 + \mathrm{T} * \mathrm{C}$ overheads are due to cross-thread and cross-core accumulations, respectively.

2.2 GPGPU

The primary strategies used to implement the GPGPU's *score* function were to optimize data placement in memory and to minimize the latency of a single kernel thread. Upon dispatching work to the GPGPU, we compacted the input to the *score* function by order and then by node ($[\mathcal{N}][\mathcal{O}]$), based on the relatively large number of orders that are to be processed per node. When possible, we aligned data in memory—rounding both $\mathcal{O}$ and $\mathcal{P}$ to the next power of two, to avoid false sharing in wide word memory operations, and to improve data alignment. In the *score* loop, we found that a direct implementation of the $log(1+exp(d))$ operation performed better than a table lookup, most likely due to the algorithm's limited use of floating point arithmetic. Broadly speaking, many of the strategies guiding our *score* function optimization effort are outlined in [2].

3. RESULTS AND ANALYSIS

We evaluate the FPGA fabric's efficiency relative to the GPGPU by normalizing the *score* function's performance to device core count, for each device shown in Table 1.

Device	BW_i (Gb/s)	BW_o (Gb/s)	M_c (Kb)	Cores	f_{core} (MHz)
gt-330m	n/a	204.8	128,0	48	1265
gtx-480	n/a	1419.2	512,6144	480	1401
v5-155t	3731	51.2	1640,7632		
v6-240t	7322	51.2	3650,14976		

Table 1: FPGA and GPGPU devices employed in each study. BW_i and BW_o describe on-chip and off-chip memory bandwidths, respectively. M_c describes the on-chip memory capacity ({distributed, block} RAM on FPGA and {local, global L2$} on the GPGPU). "Core" denotes an Nvidia CUDA core.

We have chosen to map to both the Virtex-5 LX155T (v5-155t) and Virtex-6 LX240T (v6-240t) to show FPGA core count scalability given additional fabric. The GT 330m (gt-330m—a mobile GPGPU platform) is used because its power envelope is known to be comparable to the FPGAs'. The GTX 480 (gtx-480—a workstation GPGPU based on the Nvidia Fermi architecture) allows us to compare the highest-performing GPGPU available to us with the highest performing FPGA parameterization. All studies assume single-socket systems (one FPGA/GPGPU paired with one GPP for system initialization).

3.1 Methodology

The FPGA implementation (whose best-effort parameterizations are shown in Table 2) was written in Verilog HDL and mapped to the device using Synplify Pro (Synopsys) and the Xilinx ISE 12.1 flow for placement and routing (PAR). We measured performance through cycle-accurate traces taken from post-PAR simulation, after verifying that the post-PAR netlist met timing closure and was functionally correct. To measure the GPGPU total iteration time, we ran the application for 1000 iterations without profiling code in the loop. We then calculated the percent time spent in the *score* function through measuring (a) the *score* time using timers placed around the inner loop, and (b) the total time across 1000 iterations, with the timers from (a) enabled.

Name	$\mathcal{N}$	f_{core} (MHz)	LUT (%)	FF (%)	BRAM (%)	Cores
v5-155t	32	250	66	79	95	48
v5-155t	37	250	70	83	95	48
v6-240t	32	300	80	64	99	120
v6-240t	37	300	83	67	99	120

Table 2: Best-effort FPGA configurations used.

3.2 Core Normalization Study

This study compares the absolute (T_{Sc}) and percent time spent (% Sc) in the *score* function across both GPGPU and FPGA. All sample points assume $\mathcal{O} = 512$. The results for experimental 32-node and synthetic 37-node networks are given in Table 3. We chose these datasets for their significant increase in complexity—previously surveyed networks [6, 12] did not offer enough work per iteration to saturate a GPGPU system. We focus on the *score* function and not the outer-loop control logic (shown as *Setup* and *Teardown* in Algorithm 1) because execution time is dominated by the *score* function (as shown by the % Sc columns in Table 3).

Device	T_{Sc} (s)	T_i (s)	% Sc	T_{Sc} (s)	T_i (s)	% Sc
v5-155t	0.051	0.051	99.0	0.110	0.110	99.3
v6-240t	0.018	0.018	97.7	0.036	0.037	98.4
gt-330m	0.180	0.217	82.6	0.394	0.458	86.1
gtx-480	0.014	0.020	71.4	0.027	0.037	73.5

Table 3: Absolute time (T_{Sc}) and percent time spent (% Sc) in the *score* function, relative to the latency of a single iteration (T_i). Results are given for the 32 node network (left) and the 37 node network (right), over $\mathcal{O} = 512$ orders.

		T_{Sc} Speedup		T_i Speedup	
Network	FPGA	gt-330m	gtx-480	gt-330m	gtx-480
32 Nodes	v5-155t	3.55	.275	4.26	.392
	v6-240t	10.0	.777	12.1	1.11
37 Nodes	v5-155t	3.58	.250	4.16	.336
	v6-240t	10.9	.750	12.4	1.00

Table 4: Speedup achieved by FPGA systems relative to GPGPU systems. T_{Sc} and T_i time is detailed in Table 3.

To gain insight into the performance discrepancies between the FPGA and GPGPU, we normalize performance to the number of compute cores available on each device. The resulting metric, shown in Table 5, is given by $\text{S}\textsc{c} * \frac{C_g}{C_f}$ where $\text{S}\textsc{c}$ corresponds to a T_{Sc} speedup from Table 4, and $\{C_g, C_f\}$ refers to the number of {GPGPU, FPGA} cores available on the device. We also scale the Virtex-5 FPGA performance (by $\frac{300}{250}$) to compensate for its lower obtainable frequency (in comparison with the Virtex-6 FPGA), in order to show how normalized performance is consistent across all FPGA sample points.

	gt-330m	gtx-480
v5-155t	$4.26 - 4.30$	$3.00 - 3.30$
v6-240t	$4.00 - 4.36$	$3.00 - 3.11$

Table 5: FPGA relative to GPGPU T_{Sc} speedup, normalized to the number of compute cores available and to the FPGA compute core frequency.

The algorithm is compute bound on both the FPGA and the GPGPU platforms, which is underlined by the consistent speedups in Table 5. We believe that the discrepancy in normalized performance between GPGPUs is a function of GPGPU architecture (g80 for the gt-330m and Fermi for the gtx-480) and GPGPU datapath clock frequency (1265 MHz for the gt-330m and 1401 MHz for the gtx-480).

4. DISCUSSION AND CONCLUSIONS

Notably, the application surveyed in this study does not make significant use of floating point or off-chip memory bandwidth, which are well-known strengths of the GPGPU platform. Nevertheless, it is important to account for the discrepancy in normalized performance (Table 5) between the FPGA and GPGPU platforms, especially given the relatively high GPGPU core clock frequencies. Tables 1 and 2 show that the FPGA core clocks fall between 250 and 300 MHz[2] while GPGPU core clocks reach as high as 1265 and 1401 MHz. In the case of the mobile GPGPU, we must rationalize why normalized performance falls in the FPGA's favor by a factor of $\sim 4.18\times$, while the core clock frequency of the FPGA design is $4.21\times$ less than that of the GPGPU. Similarly in the case of the Nvidia Fermi GTX480, we must account for the FPGA's $\sim 3.05\times$ gain in performance despite a $4.67\times$ loss in clock frequency.

We believe the performance gap predominantly results from (a) the performance delta between programmable cores and custom hardware datapaths, and, to a lesser extent (b) the effect of optimizing the memory access pattern between each FPGA core and its BRAM. Each FPGA core is built with replicated arithmetic units and an application-specific local memory system—which allows each FPGA core to commit one *score* inner loop iteration per cycle in the steady state. To better understand the *score* inner loop on the GPGPU, we compiled the GPGPU OpenCL kernel to a variety of GPP architectures using `GCC`. Using full optimization, the compiler produces kernels with loops averaging 39 RISC-like instructions.[3] This, coupled with the fact that not all inner loop iterations will pass the *compatible*() check in Algorithm 1, helps explain FPGA core performance.

[2]We normalize to 300 MHz for the rest of this discussion.

[3]These instruction counts are estimates as we are unable to directly examine the output of the OpenCL compiler.

One metric by which FPGA devices continue to lag, in the area of computing, is the time and expertise required to design, verify, and "compile" to an FPGA platform. Related work in high-level tools for productive FPGA design, using MIMD and SIMT many-core templates [11], has begun to address this challenge. Perhaps this approach can be extended to take advantage of a hierarchical tool flow, significantly reducing tool time. Looking forward, this approach could allow rapid prototyping of high-performance FPGA implementations with a design time and expertise requirement similar to the GPGPU programming environment.

5. ACKNOWLEDGMENTS

This project was supported by the NIH, grant no. 130826-02, the DoE*, award no. DE-SC0003624, and the Berkeley Wireless Research Center. We would also like to thank the members of the Berkeley Reconfigurable Computing group, for contributing ideas and discussions surrounding this work, and the Stanford Nolan Lab for providing data for the 32 and 37 node networks. *: Support from the DoE does not constitute the DoE's endorsement of the views expressed in this paper.

6. REFERENCES

[1] Khronos Group - OpenCL. http://www.khronos.org/opencl/

[2] Nvidia OpenCL Best Practices Guide.

[3] D. M. Chickering. Learning bayesian networks is np-complete. *Learning from Data: Artificial Intelligence and Statistics*. V. Springer Verlag, 1996.

[4] M. Teyssier and D. Koller. Ordering-based search: A simple and effective algorithm for learning bayesian networks. *Proc. of the Twenty-first Conference on Uncertainty in AI (UAI)*, pages 584–590, Edinburgh, Scotland, UK, July 2005.

[5] B. Ellis and W. H. Wong. Learning causal bayesian network structures from experimental data. *Journal of the American Statistical Association*, 103(482), 2008.

[6] N. B. Asadi, T. H. Meng, and W. H. Wong. Reconfigurable computing for learning bayesian networks. *Proc. of the 16th Intl. Symposium on FPGAs*, 2008.

[7] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong and W. m. Hwu. FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs *Proc. of the Symposium on Application Specific Processors*, 2009.

[8] K. Server, B. Khaled, and L. Ying. A high performance fpga-based implementation of position specific iterated blast. *Proc. of the 17th Intl. Symposium on FPGAs,* 2009.

[9] M. Lin, I. Lebedev, J. Wawrzynek. High-throughput bayesian computing machine with reconfigurable hardware. *Proc. of the 18th Intl. Symposium on FPGAs*, 2010.

[10] M. Lin, I. Lebedev, J. Wawrzynek. OpenRCL: low-power high-performance computing with reconfigurable devices. *Proc. of the 20th Intl. Conference on Field Programmable Logic and Applications*, 2010.

[11] I. Lebedev, S. Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, M. Lin, J. Wawrzynek. MARC: A Many-Core Approach to Reconfigurable Computing. *Proc. of the 6th international conference on reconfigurable computing and FPGAs*, 2010.

[12] N. B. Asadi, C. W. Fletcher, G. Gibeling, E. N. Glass, K. Sachs, D. Burke, Z. Zhou, J. Wawrzynek, W. H. Wong, and G. P. Nolan. *Paralearn:* a massively parallel, scalable system for learning interaction networks on fpgas. *Proc. of the 24th Intl. Conference on Supercomputing*, 2010.

[13] D. Kumar, M. A. Qadeer. Fast heterogeneous computing with CUDA compatible Tesla GPU computing processor. *Proc. of the International Conference and Workshop on Emerging Trends in Technology*, 2010.

[14] M. D. Linderman, R. Bruggner, V. Athalye, T. H. Meng, N. B. Asadi, and G. P. Nolan. High-throughput bayesian network inference using heterogeneous multicore computers. *Proc. of the 24th Intl. Conference on Supercomputing*, 2010.