Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 439141, 15 pages
doi:10.1155/2012/439141

Research Article

Exploring Many-Core Design Templates for FPGAs and ASICs

Ilia Lebedev,[1] Christopher Fletcher,[1] Shaoyi Cheng,[2] James Martin,[2] Austin Doupnik,[2] Daniel Burke,[2] Mingjie Lin,[2] and John Wawrzynek[2]

[1] *CSAIL, Massachusetts Institute of Technology, Cambridge, MA 02139, USA*
[2] *Department of EECS, University of California at Berkeley, CA 94704, USA*

Correspondence should be addressed to Ilia Lebedev, ilebedev@csail.mit.edu

Received 2 May 2011; Accepted 15 July 2011

Academic Editor: Claudia Feregrino

We present a highly productive approach to hardware design based on a many-core microarchitectural template used to implement compute-bound applications expressed in a high-level data-parallel language such as OpenCL. The template is customized on a per-application basis via a range of high-level parameters such as the interconnect topology or processing element architecture. The key benefits of this approach are that it (i) allows programmers to express parallelism through an API defined in a high-level programming language, (ii) supports coarse-grained multithreading and fine-grained threading while permitting bit-level resource control, and (iii) reduces the effort required to repurpose the system for different algorithms or different applications. We compare template-driven design to both full-custom and programmable approaches by studying implementations of a compute-bound data-parallel Bayesian graph inference algorithm across several candidate platforms. Specifically, we examine a range of template-based implementations on both FPGA and ASIC platforms and compare each against full custom designs. Throughout this study, we use a general-purpose graphics processing unit (GPGPU) implementation as a performance and area baseline. We show that our approach, similar in productivity to programmable approaches such as GPGPU applications, yields implementations with performance approaching that of full-custom designs on both FPGA and ASIC platforms.

1. Introduction

Direct hardware implementations, using platforms such as FPGAs and ASICs, possess a huge potential for exploiting application-specific parallelism and performing efficient computation. As a result, the overall performance of custom hardware-based implementations is often higher than that of software-based ones [1, 2]. To attain bare metal performance, however, programmers must employ hardware design principles such as clock management, state machines, pipelining, and device specific memory management—all concepts well outside the expertise of application-oriented software developers.

These observations raise a natural question: *does there exist a more productive abstraction for high-performance hardware design*? Based on modern programming disciplines, one viable approach would (1) allow programmers to express parallelism through some API defined in a high-level programming language, (2) support coarse-grain multithreading and fine-grain threading while permitting bit-level resource control, and (3) reduce the effort required to repurpose the implemented hardware platform for different algorithms or different applications. This paper proposes an abstraction that constrains the design to a *microarchitectural template*, accompanied by an API, that meets these programmer requirements.

Intuitively, constraining the design to a template would likely result in performance degradation compared to fully-customized solutions. Consider the high-level chart plotting designer effort versus performance, shown in Figure 1. We argue that the shaded region in the figure is attainable by template-based designs and warrants a systematic exploration. To that end, this work attempts to quantify the performance/area tradeoff, with respect to designer effort, across template-based, hand-optimized, and programmable approaches on both FPGA and ASIC platforms. From our analysis, we show how a disciplined approach with architectural constraints, without resorting to manual hardware design,

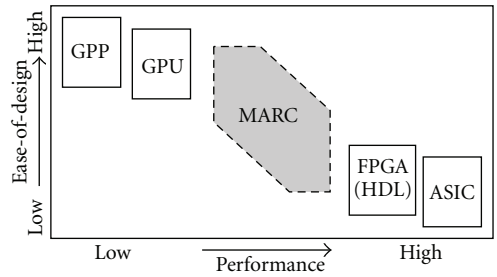

Figure 1: Landscape of modern computing platforms. Ease of application design and implementation versus performance (GPP stands for *general purpose processor*).

may reduce design time and effort while maintaining acceptable performance.

In this paper; we study microarchitectural templates in the context of a compute-intensive data-parallel Bayesian inference application. Our thesis, therefore, is that we can efficiently map our application to hardware while being constrained to a *many-core* template and parallel programming API. We call this project MARC, for *Many-core Approach to Reconfigurable Computing*, although template-based architectures can be applied outside the many-core paradigm.

We think of a *template* as an architectural model with a set of parameters to be chosen based on characteristics of the target application. Our understanding of which aspects of the architecture to parameterize continues to evolve as we investigate different application mappings. However, obvious parameters in the many-core context are the number of processing cores, core arithmetic-width, core pipeline depth, richness and topology of an interconnection network, and customization of cores—from addition of specialized instructions to fixed function datapaths as well as details of the cache and local store hierarchies. In this study we explore a part of this space and compare the performance/area between MARC and hand-optimized designs in the context of a baseline GPGPU implementation.

The rest of the paper is organized as follows: Section 2 introduces the Bayesian network inference application, the case study examined in this paper. Section 3 describes the execution model used by OpenCL, the high-level language used to describe our MARC and GPGPU implementations. The application mapping for the GPGPU platform is detailed in Section 4. We discuss the hand-optimized and MARC implementations in Section 5. Section 6 covers hardware mappings for both the hand-optimized and MARC designs as well as a comparison between FPGA and ASIC technology. Finally, in Section 7, we compare the performance/area between the MARC hand-optimized and GPGPU implementations.

1.1. Related Work. Numerous projects and products have offered ways to ease FPGA programming by giving developers a familiar C-style language in place of HDLs [3]. Early research efforts including [4–6] formed the basis for recent commercial offerings: Catapult C from Mentor Graphics, ImpulseC, Synfora Pico from Synopsys, and AutoESL from Xilinx, among others. Each of these solutions requires developers to understand hardware-specific concepts and to program using a model that differs greatly from standard C—in a sense, using an HDL with a C syntax. Unlike these approaches, the goal of MARC is to expose software programming models applicable to design of efficient hardware.

There has been a long history of mapping conventional CPU architectures to FPGAs. Traditionally, soft processors have been used either as a controller for a dedicated computing engine, or as an emulation or prototyping plat-form for design verification. These efforts have primarily employed a single or small number of processor cores. A few FPGA systems with a large number of cores have been implemented, such as the RAMP project [7]. However, the primary design and use of these machines have been as emulation platforms for custom silicon designs.

There have been many efforts both commercially and academically on customization of parameterized processor cores on an application-specific basis. The most widely used is Xtensa from Tensilica, where custom instructions are added to a conventional processor architecture. We take processor customization a step further by allowing the instruction processors to be replaced with an application-specific datapath that can be generated automatically via our C-to-gates tool for added efficiency and performance. We leverage standard techniques from C-to-gates compilation to accomplish the generation of these custom datapaths.

More recently, there have been several other efforts in integrating the C-to-gates flow with parallel programming models, [8, 9]. These projects share with MARC the goal of exploiting progress in parallel programming languages and automatically mapping to hardware.

2. Application: Bayesian Network Inference

This work's target application is a system that learns Bayesian network structure from observation data. Bayesian networks (BNs) and graphical models have numerous applications in bioinformatics, finance, signal processing, and computer vision. Recently they have been applied to problems in systems biology and personalized medicine, providing tools for processing everincreasing amounts of data provided by high-throughput biological experiments. BNs' probabilistic nature allows them to model uncertainty in real life systems as well as the noise that is inherent in many sources of data. Unlike Markov Random Fields and undirected graphs, BNs can easily learn sparse and causal structures that are interpretable by scientists [10–12].

We chose to compare MARC in the context of Bayesian inference for two primary reasons. First, Bayesian inference is a computationally intensive application believed to be particularly well suited for FPGA acceleration as illustrated by [13]. Second, our group, in collaboration with Stanford University, has expended significant effort over the previous two years developing several generations of a hand-optimized FPGA implementation tailored for Bayesian inference [13, 14]. Therefore, we have not only a concrete reference design but also well-corroborated performance results for fair comparisons with hand-optimized FPGA implementations.

2.1. Statistics Perspective. BNs are statistical models that capture conditional independence between variables via the local Markov property: *that a node is conditionally independent of its nondescendants, given its parents*. Bayesian inference is the process by which a BN's graph structure is learned from the quantitative observation, 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

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

Despite significant recent progress in algorithm development, computational inference of a BN's graph structure from evidence is still NP-hard [15] and remains infeasible except for cases with only a small number of variables [16].

The algorithm surveyed in this paper is the union of two BN inference kernels, the order and graph samplers (jointly called the "order-graph sampler"). An *order* is given by a topological sort of a graph's nodes, where each node is placed after its parents. Each order can include or be *compatible* with many different graphs. The order sampler takes a BN's observation data 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. The observation data is generated in a preprocessing steps and consists of $\mathcal{N}$ (*for each node*) sets of $\mathcal{P}$ local-score/parent-set pairs (which we will refer to as "data" for short). A local score describes the likelihood that a given parent set is a node's true parent set, given an order. A postprocessing step is performed after the order-graph sampler to normalize scores and otherwise clean up the result before it is presented to the user.

In this work, we only study the order and graph sampler steps for two reasons. First, the order-graph sampler is responsible for most of the algorithm's computational complexity. Second, the pre- and postprocessing phases are currently performed on a GPP (general purpose processor) platform regardless of the platform chosen to implement the order-graph sampler.

Following [14, 16, 17], the order-graph sampler uses Markov chain Monte Carlo (MCMC) sampling to perform an iterative random walk in the space of BN orders. First, the algorithm picks a random initial order. The application then iterates as follows (1) the current order is modified by swapping two nodes to form a "proposed order," which is (2) scored, and (3) either accepted or rejected according to the Metropolis-Hastings rule. The scoring process, itself, (1) breaks the proposed 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. For a network of $\mathcal{N}$ nodes, the proposed order's score can be efficiently calculated by [18]

$$\text{Score}\left(\mathcal{O}_p \mid \mathcal{D}\right) = \prod_{i=1}^{\mathcal{N}} \sum_{\Pi_i \in \Pi_{\mathcal{P}}} \text{LocalScore}\left(V_i, \Pi_i; \mathcal{D}, \mathcal{O}_p\right), \tag{2}$$

where $\mathcal{D}$ is the set of raw observations that are used by the preprocessor to generate local scores, and $\mathcal{O}_p$ is the proposed order. This iterative operation continues until the score has converged.

To decrease time to convergence, $\mathcal{C}$ orders (together called a *chain*) can be dispatched over a temperature ladder and exchanged per iteration in a technique known as parallel tempering. Additionally, $\mathcal{R}$ independent chains (called multiple *restarts*) can be dispatched to increase confidence that the optimum score is a global optimum.

2.2. Compute Perspective. The order-graph sampler is a compute intensive set of nested loops, shown in Algorithm 1, while the *score*() function arithmetic is shown in Algorithm 2. To put the number of loop iterations into perspec-tive, typical parameter values for $\{\mathcal{I}, \mathcal{R} * \mathcal{C}, \mathcal{N}\}$ are $\{10000, 512, 32 \text{ or } 37\}$, where $\mathcal{I}$ is the number of MCMC iterations. Furthermore, $\mathcal{P} = \sum_{i=0}^{4} \binom{\mathcal{N}-1}{i}$, which equates to 36457 and 66712 ($\mathcal{N} = 32$ and $\mathcal{N} = 37$, resp.) for the design points that we study (see Section 7).

We classify the reformulated loop nest as compute intensive for two reasons. First, a relatively small amount of input (a local order) is needed for the *score*() function to compute per-node results over the $\mathcal{R} * \mathcal{C}$ orders. Second, $\mathcal{D}[n]$ (shown in Algorithm 1) depends on $\mathcal{N}$ and not $\mathcal{R} * \mathcal{C}$. Since $\mathcal{N} \approx 37$ and $\mathcal{R} * \mathcal{C} = 512$ (i.e., $\mathcal{R} * \mathcal{C} \gg \mathcal{N}$), in practice there is a large amount of compute time between when n in $\mathcal{D}[n]$ changes.

3. The OpenCL Execution Model

To enable highly productive hardware design, we employ a high-level language and execution model well suited for the paradigm of applications we are interested in studying: data-parallel, compute-bound algorithms. Due to its popularity, flexibility, and good match with our goals, we employ OpenCL (Open Computing Language) as the programming model used to describe applications.

OpenCL [19] is a programming and execution model for heterogeneous systems containing GPPs, GPGPUs, FPGAs [20], 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, device). The user specifies how the kernels map to an n-dimensional dataset, given a set of arguments (such as constants or pointers to device or 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.

A key property of OpenCL is its memory model. Each kernel has access to three disjoint memory regions: private, local, and global. Global memory is shared by all kernel threads, local memory is shared by threads belonging to the same group, while private memory is owned by one kernel thread. This alleviates the need for a compiler to perform

```
    for {r, c} in ℛ × 𝒞 do
      initialize(𝒪[r][c])
    end for
    for i in ℐ do
5:    for {r, c} in ℛ × 𝒞 do
        𝒪_p[r][c] ← swap(𝒪[r][c])
        Variable initialization:
        𝒪_p[r][c] · 𝒮_o, 𝒪_p[r][c] · 𝒮_g ← 0
        𝒪_p[r][c] · 𝒢 ← []
10:   end for
      for n in 𝒩 do
        for {r, c} in ℛ × 𝒞 do
          (s_o, s_g, g) = score(𝒟[n], 𝒪_p[r][c][n])
          𝒪_p[r][c] · 𝒮_o ← 𝒪_p[r][c] · 𝒮_o + s_o
15:       𝒪_p[r][c] · 𝒮_g ← 𝒪_p[r, c] · 𝒮_g + s_g
          𝒪_p[r][c] · 𝒢 · append(g)
        end for
      end for
      for r in ℛ do
        Metropolis-hastings:
20:     for c in 𝒞 do
          if 1/T_c × (𝒪_p[r][c] · 𝒮_o − 𝒪[r][c] · 𝒮_o) >
          log(rand(0, 1)) then
            𝒪[r, c] · 𝒮_o ← 𝒪_p[r][c] · 𝒮_o
            save({𝒪_p[r][c] · 𝒮_g, 𝒪_p[r, c] · 𝒢})
          end if
25:     end for
        Parallel tempering:
        for c in 𝒞 do
          d ← 𝒪[r][c] · 𝒮_o − 𝒪[r][c + 1] · 𝒮_o
          if log(rand(0, 1)) < d × (1/T_c − 1/T_{c+1}) then
30:         exchange(𝒪[r][c], 𝒪[r][c + 1])
          end if
        end for
      end for
    end for
```

Algorithm 1: The reformulated order-graph sampler loop nest. $\{\mathcal{S}_o, \mathcal{S}_g\}$ and $\mathcal{G}$ are the current {order, graph} scores and graph associated with an order. *initialize*() generates a random order, *swap*() exchanges nodes in an order, and *save*() saves a result for the postprocessing step.

costly memory access analysis to recognize dependencies before the application can be parallelized. Instead, the user specifies how the application is partitioned into data-parallel kernels. With underlying SIMD principles, OpenCL is well suited for data-parallel problems and maps well to the parallel thread dispatch architecture found in GPGPUs.

4. Baseline GPGPU Implementation

To implement the order-graph sampler on the GPGPU, the application is first divided to different parts according to their characteristics. The scoring portion of the algorithm, which exhibits abundant data parallelism, is partitioned into a kernel and executed on the GPGPU, while the less parallelizable score accumulation is executed on a GPP. This ensures that the kernel executed on the GPGPU is maximally parallel and exhibits no interthread communication—an approach we experimentally determined to be optimal. Under this scheme, the latency of the control thread and score accumulation phases of the application, running on a GPP, are dominated by the latency of the scoring function running on the GPGPU. Moreover, the *score*() kernel (detailed in the following section) has a relatively low bandwidth requirement, allowing us to offload accumulation to the GPP, lowering total latency. The GPP-GPGPU implementation is algorithmically identical to the hardware implementations, aside from minor differences in the precision of the $log1p(\exp(d))$ operation, and yields identical results up to the random sequence used for Monte Carlo integration.

4.1. Optimization of Scoring Kernel. We followed four main strategies in optimizing the scoring unit kernel: (1) minimizing data transfer overhead between the control thread and the scoring function, (2) aligning data in device memory, (3)

```
s_o, s_g ← −∞
g ← NULL
for p in 𝒫 do
  if compatible(𝒟[p] · ps, 𝒪_l) then
5:    Order sampler:
      d ← 𝒟[p] · ls − s_o
      If d ≥ HIGH_THRESHOLD then
        s_o ← 𝒟[p] · ls
      else if d > LOW_THRESHOLD then
10:       s_o ← s_o + log(1 + exp(d))
      end if
      Graph sampler:
      if 𝒟[p] · ls > s_g then
        s_g ← 𝒟[p] · ls
15:     g ← 𝒟[p] · ps
      end if
  end if
end for
Return:(s_o, s_g, g)
```

Algorithm 2: The $score(\mathcal{D}, \mathcal{O}_l)$ function takes the data $\mathcal{D}$ (made of parent set (ps) and local score (ls) pairs) and a local order ($\mathcal{O}_l$) as input. The scoring function produces an order score (s_o), graph score (s_g), and graph fragment (g).

allocating kernel threads to compute units on the GPGPU, and (4) minimizing latency of a single kernel thread.

First, we minimize data transfers between the GPP and GPGPU by only communicating changing portions of the data set throughout the computation. At application startup, we statically allocate memory for all arrays used on the GPGPU, statically set these arrays' pointers as kernel arguments, and copy all parent sets and local scores into off-chip GPGPU memory to avoid copying static data each iteration. Each iteration, the GPP copies $\mathcal{R} * \mathcal{C}$ proposed orders to the GPGPU and collects $\mathcal{R} * \mathcal{C} * \mathcal{N}$ proposed order/graph scores, as well as $\mathcal{R} * \mathcal{C}$ graphs from the GPGPU. Each order and graph is an $\mathcal{N} \times \mathcal{N}$ matrix, represented as $\mathcal{N}$ 64 bit integers, while partial order and graph scores are each 32 bit integers (additional range is introduced when the partial scores are accumulated). The resulting bandwidth requirement per iteration is $8 * \mathcal{R} * \mathcal{C} * \mathcal{N}$ bytes from the GPP to the GPGPU and $16 * \mathcal{R} * \mathcal{C} * \mathcal{N}$ bytes from the GPGPU back to the GPP. In the BNs surveyed in this paper, this bandwidth requirement ranges from 128 to 256 KB (GPP to GPGPU) and from 256 to 512 KB (GPGPU to GPP). Given these relatively small quantities and the GPGPU platform's relatively high transfer bandwidth over PCIe, the transfer latency approaches a minimal value. We use this to our advantage and offload score accumulation to the GPP, trading significant accumulation latency for a small increase in GPP-GPGPU transfer latency. This modification gives us an added advantage via avoiding intra-kernel communication altogether (which is costly on the GPGPU because it does not offer hardware support for producer-consumer parallelism).

Second, we align and organize data in memory to maximize access locality for each kernel thread. GPGPU memories are seldom cached, while DRAM accesses are several words wide—comparable to GPP cache lines. We therefore coalesce memory accesses to reduce the memory access range of a single kernel and of multiple kernels executing on a given compute unit. No thread accesses (local scores and parent sets) are shared across multiple nodes, so we organize local scores and parent sets by $[\mathcal{N}][\mathcal{P}]$. When organizing data related to the $\mathcal{R} * \mathcal{C}$ orders (the proposed orders, graph/order scores, and graphs), we choose to maximally compact data for restarts, then chains, and finally nodes ($[\mathcal{N}][\mathcal{C}][\mathcal{R}]$). This order is based on the observation that a typical application instance will work with a large number of restarts relative to chains. When possible, we align data in memory—rounding both $\mathcal{R}$, $\mathcal{C}$ and $\mathcal{P}$ to next powers of two to avoid false sharing in wide word memory operations and to improve alignment of data in memory.

Third, allocating kernel threads to device memory is straightforward given the way we organize data in device memory; we allocate multiple threads with localized memory access patterns. Given our memory layout, we first try dispatching multiple restarts onto the same compute unit. If more threads are needed than restarts available, we dispatch multiple chains as well. We continue increasing the number of threads per compute unit in this way until we reach an optimum—the point where overhead due to multithreading overtakes the benefit of additional threads. Many of the strategies guiding our optimization effort are outlined in [21].

Finally, we minimize the scoring operation latency over a single kernel instance. We allow the compiler to predicate conditional to avoid thread divergence. Outside the inner loop, we explicitly precompute offsets to access the $[\mathcal{N}][\mathcal{P}]$ and $[\mathcal{N}][\mathcal{C}][\mathcal{R}]$ arrays to avoid redundant computation. We experimentally determined that loop unrolling the $score()$ loop has minimal impact on kernel performance, so we allow the compiler to unroll freely. We also evaluated a direct implementation of the $log1p(\exp(d))$ operation versus the use of a lookup table in shared memory (which mirrors

the hand-optimized design's approach). Due to the low utilization of the floating point units by this algorithm, the direct implementation tends to perform better than a lookup table given the precision required by the algorithm.

4.2. Benchmarking the GPGPU Implementation. To obtain GPGPU measurements, We mapped the data parallel component to the GPGPU via OpenCL, and optimized the resulting kernel as detailed in Section 4.1. We measured the relative latency of each phase of the algorithm by introducing a number of GPP and GPGPU timers throughout the iteration loop. We then computed the latency of each phase of computation (scoring, accumulation, MCMC, etc.) and normalized to the measured latency of a single iteration with no profiling syscalls. To measure the iteration time, we ran the application for 1000 iterations with no profiling code in the loop and then measured the total time elapsed using the system clock. We then computed the aggregate iteration latency.

5. Architecture on Hardware Platforms

As with the GPGPU implementation, when the Bayesian inference algorithm is mapped to hardware platforms (FPGA/ASIC), it is partitioned into two communicating entities: a data-parallel scoring unit (a collection of Algorithmic-cores or A-cores) and a control unit (the Control-core, or C-core). The A-cores are responsible for all iterations of the *score*() function from Algorithm 1 while the C-core implements the serial control logic around the *score*() calls. This scheme is applied to both the hand-optimized design and the automatically generated MARC design, though each of them has different interconnect networks, memory subsystems, and methodologies for creating the cores.

5.1. Hand-Optimized Design. The hand-optimized design mapping integrates the jobs of the C-core and A-cores on the same die and uses a front-end GPP for system initialization and result collection. At the start of a run, network data and a set of $\mathcal{R} * \mathcal{C}$ initial orders are copied to a DRAM accessible by the A-cores, and the C-core is given a "Start" signal. At the start of each iteration, the C-core forms $\mathcal{R} * \mathcal{C}$ proposed orders, partitions each by node, and dispatches the resulting $\mathcal{N} * \mathcal{R} * \mathcal{C}$ local orders as threads to the A-cores. As each iteration completes, the C-core streams results back to the front-end GPP while proceeding with the next MCMC iteration.

The hand-optimized design is partitioned into four clock domains. First, we clock A-cores at the highest frequency possible (between 250 and 300 MHz) as these have a direct impact on system performance. Second, we clock the logic and interconnect around each A-core at a relatively low frequency (25–50 MHz) as the application is compute bound in the cores. Third, we set the memory subsystem to the frequency specified by the memory (~200 MHz, using a DRAM DIMM in our case). Finally, the C-core logic is clocked at 100 MHz, which we found to be ideal for timing closure and tool run time given a performance requirement (the latency of the C-core is negligible compared to the A-cores).

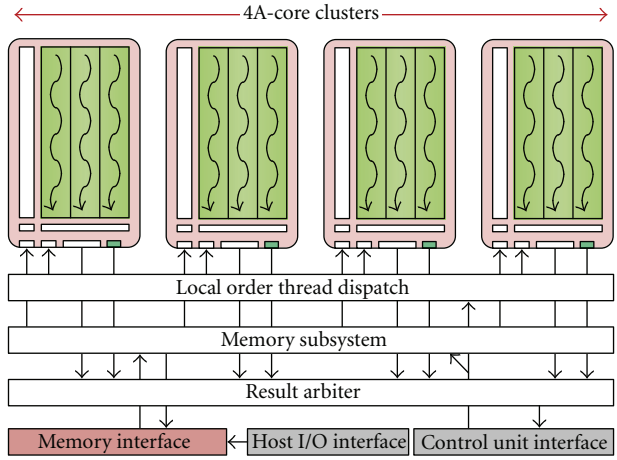

Figure 2: The hand-optimized scoring unit (with 4 A-core clusters).

5.1.1. Scoring Unit. The scoring unit (shown in Figure 2) consists of a collection of clustered A-cores, a point-to-point interface with the control unit, and an interface to off-chip memory.

A scoring unit cluster caches some or all of a node's data, taking a stream of local orders as input and outputs order/graph scores as well as partial graphs. A cluster is made up of {A-cores, RAM} pairs, where each RAM *streams* data to its A-core. When a cluster receives a local order, it (a) pages in data from DRAM as needed by the local orders, strip-mines that data evenly across the RAMs and (b) dispatches the local order to each core. Following Algorithm 1, $\mathcal{R} * \mathcal{C}$ local orders can be assigned to a cluster per DRAM request. Once data is paged in, each A-core runs $\mathcal{P}_f/U_c$ iterations of the *score*() inner loop (from Algorithm 2), where $\mathcal{P}_f$ is the subset of $\mathcal{P}$ that was paged into the cluster, and U_c is the number of A-cores in the cluster.

A-core clusters are designed to maximize local order throughput. A-cores are replicated to the highest extent possible to maximize read bandwidth achievable by the RAMs. Each A-core is fine-grained multithreaded across multiple iterations of the *score*() function and uses predicated execution to avoid thread divergence in case of non-compatible (!*compatible*()) parent sets. To avoid structural hazards in the scoring pipeline, all scoring arithmetic is built directly into the hardware.

Mapping a single node's scoring operation onto multiple A-cores requires increased complexity in accumulating partial node scores at the end of the *score*() loop. To maximally hide this delay, we first interleave cross-thread accumulation with the next local order's main scoring operation (shown in Figure 3). Next, we chain A-cores together using a dedicated interconnect, allowing cross-core partial results to be interleaved into the next local order in the same way as threads. Per core, this accumulation scheme adds T cycles

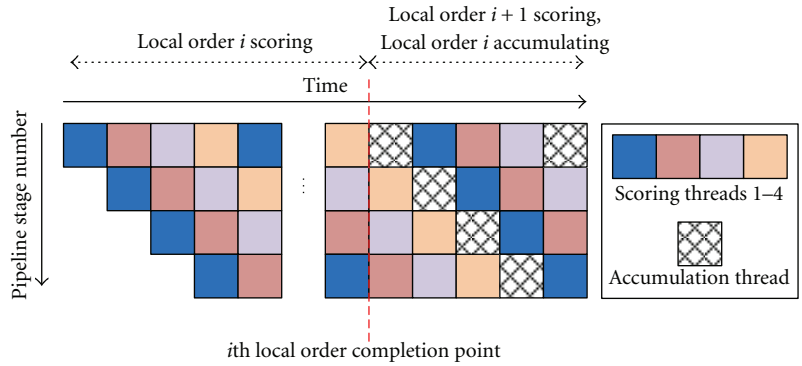

Figure 3: Thread accumulation over a 4-thread/stage core for two adjacent local orders.

of accumulation overhead to the scoring process, for a T-thread datapath and a single additional cycle for cross-core accumulation. To simplify the accumulation logic, we linearly reduce all threads across an A-core and then accumulate linearly across A-cores. The tradeoff here is that the last local order's accumulation is not hidden by another local order being scored and takes $T^2 + T * U_c$ cycles to finish, where U_c is the number of A-cores in the cluster.

Given sufficient hardware resources, more advanced A-core clusters can be built to further increase system throughput. First, the number of A-cores per RAM can be increased to the number of read ports each RAM has. Second, since a given node's data ($\mathcal{D}[n]$) does not change over the $\mathcal{R} * \mathcal{C}$ local orders, A-core chains can be replicated entirely. In this case, we say that the cluster has been split into two or more *lanes*, where A-cores from each lane are responsible for a different local order. In this setup, the cluster's control strip-mines local orders across lanes to initiate scoring. While scoring, corresponding A-cores (the first A-core in each of several lanes, e.g.) across the lanes (called *tiles*) read and process the same data from the same RAM data stream. An example of an advanced A-core cluster is shown in Figure 4.

The following analytic model can be used to estimate the parallel completion time to score $\mathcal{O}_l$ local orders over the $\mathcal{P}_f$ subset of the data (for a single cluster):

$$\text{Cycles}_{\text{DRAM}} + \frac{\mathcal{O}_l}{U_l} * \left(\frac{\mathcal{P}_f}{U_c} + (T+1)\right) + (T^2 + T * U_c), \tag{3}$$

where $\text{Cycles}_{\text{DRAM}}$ is the number of cycles (normalized to the core clock) required to initialize the cluster from DRAM, U_c is the number of A-cores per lane (doubles when two SRAM ports are used, etc.), U_l is the number of lanes per cluster, and T is the number of hardware threads per A-core.

5.1.2. Memory Subsystem. The scoring unit controls DRAM requests when an A-core cluster requires a different subset of the data. Regardless of problem parameters, data is always laid out contiguously in memory. As DRAM data is streamed to a finite number of RAMs, there must be enough RAM write bandwidth to consume the DRAM stream. In cases where the RAM write capability does not align to the DRAM read capacity, dedicated alignment circuitry built into the scoring unit dynamically realigns the data stream.

5.1.3. Control Unit. We implemented the MCMC control unit directly in hardware, according to Figure 5. The MCMC state machine, node swapping logic, parallel tempering logic, and Metropolis-Hasting logic is mapped as hardware state machines. Furthermore, a DSP block is used for multiplicative factors, while $log(rand(0,1))$ is implemented as a table lookup. The random generators for row/column swaps, as well as Metropolis-Hastings and parallel tempering, are built using free-running LFSRs.

At the start of each iteration, the control unit performs node swaps for each of the $\mathcal{R} * \mathcal{C}$ orders and schedules the proposed orders onto available compute units. To minimize control unit time when $\mathcal{R} * \mathcal{C}$ is small, orders are stored in row order in RAM, making the swap operation a single cycle row swap, followed by an $\mathcal{N}$ cycle column swap. Although the control unit theoretically has cycle accurate visibility of the entire system and can therefore derive optimal schedules, we found that using a trivial greedy scheduling policy (first come first serve) negligibly degrades performance with the benefit of significantly reducing hardware complexity. To minimize A-core cluster memory requirements, all $\mathcal{R} * \mathcal{C}$ local orders are scheduled to compute units in bulk over a single node.

When each iteration is underway, partial scores received from the scoring unit are accumulated as soon as they are received, using a dedicated A-core attached to a buffer that stores partial results. In practice, each A-core cluster can only store data for a part of a given node at a time. This means that the A-core, processing partial results, must perform both the slower *score*() operation and the simpler cross-node "+" accumulations. We determined that a single core dedicated to this purpose can rate match the results coming back from the compute-bound compute units.

At the end of each iteration, Metropolis-Hastings checks proceed in $[\mathcal{R}][\mathcal{C}]$ order. This allows the parallel tempering exchange operation for restart r to be interleaved with the Metropolis-Hastings check for restart $r+1$.

5.2. The MARC Architecture

5.2.1. Many-Core Template. The overall architecture of a MARC system, as illustrated in Figure 6, resembles a scalable, many-core-style processor architecture, comprising one

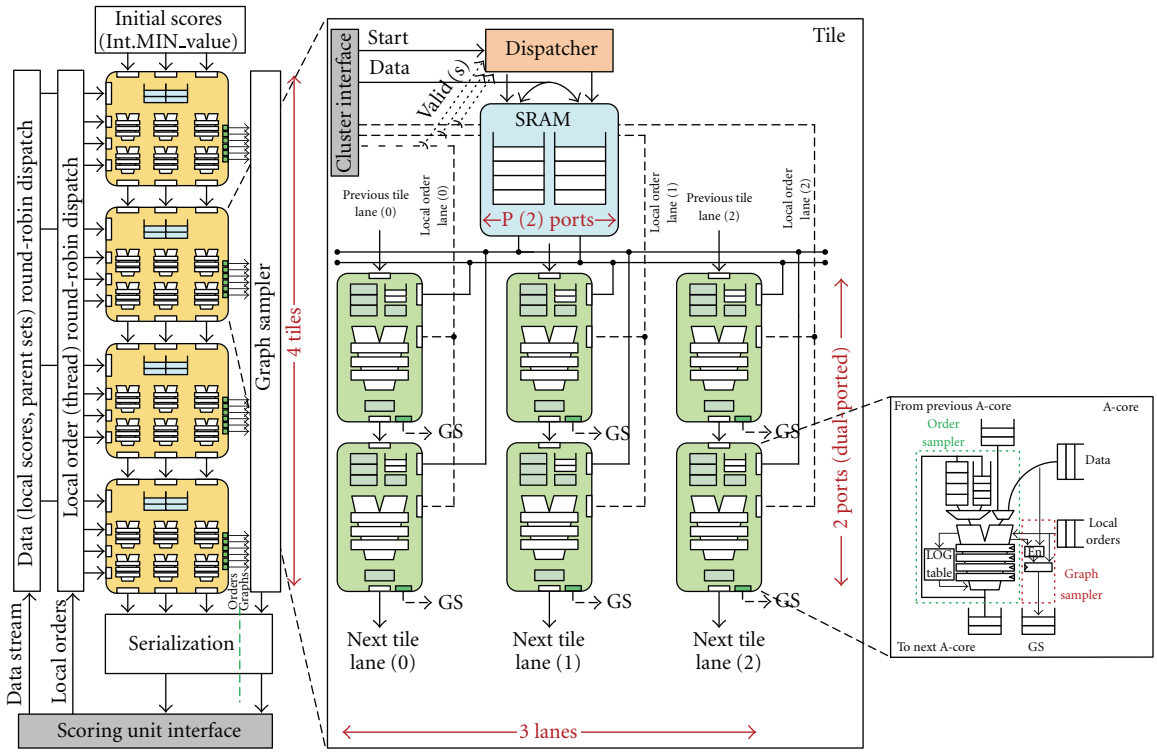

FIGURE 4: The hand-optimized A-core cluster. This example contains four tiles and three lanes and uses two RAM read ports per tile. "GS" stands for graph sampler.

Control Processor (C-core) and multiple Algorithmic Processing Cores (A-cores). Both the C-cores and the A-core can be implemented as conventional pipelined RISC processors. However, unlike embedded processors commonly found in modern SOCs, the processing cores in MARC are completely parameterized with variable bit width, reconfigurable multithreading, and even aggregate/fused instructions. Furthermore, A-cores can alternatively be synthesized as fully customized datapaths. For example, in order to hide global memory access latency, improve processing node utilization, and increase the overall system throughput, a MARC system can perform fine-grained multithreading through shift register insertion and automatic retiming. Finally, while each processing core possesses a dedicated local memory accessible only to itself, a MARC system has a global memory space implemented as distributed memories accessible by all processing cores through the interconnect network. Communication between a MARC system and its host can be realized by reading and writing global memory.

5.2.2. Execution Model and Software Infrastructure. Our MARC system builds upon both LLVM, a production-grade open-source compiler infrastructure [22] and OpenCL.

Figure 7 presents a high-level schematic of a typical MARC machine. A user application runs on a host according to the models native to the host platform—a high-performance PC in our study. Execution of a MARC program occurs in two parts: kernels that run on one or more A-cores of the MARC devices and a control program that runs on the C-core. The control program defines the context for the kernels and manages their execution. During the execution, the MARC application spawns kernel threads to run on the A-cores, each of which runs a single stream of instructions as SPMD units (each processing core maintains its own program counter).

5.2.3. Application-Specific Processing Core. One strength of MARC is its capability to integrate fully customized application-specific processing cores/datapaths so that the kernels in an application can be more efficiently executed. To this end, a high-level synthesis flow depicted by Figure 8 was developed to generate customized datapaths for a target application.

The original kernel source code in C/C++ is first compiled through `llvm-gcc` to generate the intermediate representation (IR) in the form of a single static assignment graph (SSA), which forms a control flow graph where instructions are grouped into basic blocks. Within each basic block, the instruction parallelism can be extracted easily as all false dependencies have been removed in the SSA representation. Between basic blocks, the control dependencies can then be transformed to data dependencies through branch predication. In our implementation, only memory operations are predicated since they are the only instructions that can generate stalls in the pipeline. By converting the control dependencies to data dependencies, the boundaries between basic blocks can be eliminated. This results in a single data flow graph with each node corresponding to a single instruction

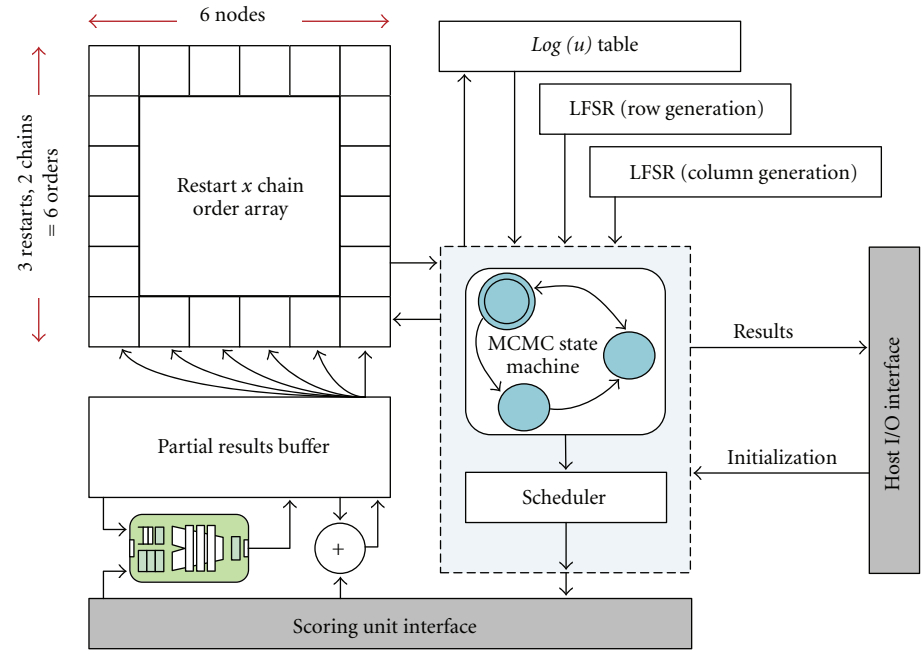

Figure 5: The hand-optimized control unit.

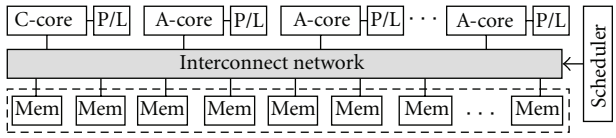

Figure 6: Diagram of key components in a MARC machine.

in the IR. Creating hardware from this graph involves a one-to-one mapping between each instruction and various predetermined hardware primitives. To utilize loop level parallelism, our high-level synthesis tool also computes the minimal interval at which a new iteration of the loop can be initiated and subsequently generates a controller to pipeline loop iterations. Finally, the customized cores have the original function arguments converted into inputs. In addition, a simple set of control signals is created to initialize a C-core and to signal the completion of the execution. For memory accesses within the original code, each nonaliasing memory pointer used by the *C* function is mapped to a memory interface capable of accommodating variable memory access latency. The integration of the customized cores into a MARC machine involves mapping the input of the cores to memory addresses accessible by the control core, as well as the addition of a memory handshake mechanism allowing cores to access global and local memories. For the results reported in this paper, the multithreaded customized cores are created by manually inserting shift registers into the single-threaded, automatically generated core.

5.2.4. Host-MARC Interface. Gigabit Ethernet is used to implement the communication link between the host and the MARC device. We leveraged the GateLib [23] project from Berkeley to implement the host interface, allowing the physical transport to be easily replaced by a faster medium in the future.

5.2.5. Memory Organization. Following OpenCL, A-core threads have access to three distinct memory regions: private, local, and global. Global memory permits read and write access to all threads within any executing kernels on any processing core (ideally, reads and writes to global memory may be cached depending on the capabilities of the device, however in our current MARC machine implementation, caching is not supported). Local memory is a section of the address space shared by the threads within a computing core. This memory region can be used to allocate variables that are shared by all threads spawned from the same computing kernel. Finally, private memory is a memory region that is dedicated to a single thread. Variables defined in one thread's private memory are not visible to another thread, even when they belong to the same executing kernel.

Physically, the private and local memory regions in a MARC system are implemented using on-chip memories. Part of the global memory region also resides on-chip, but we allow external memory (i.e., through the DRAM controller) to extend the global memory region, resulting in a larger memory space.

5.2.6. Kernel Scheduler. To achieve high throughput, kernels must be scheduled to avoid memory access conflicts. The MARC system allows for a globally aware kernel scheduler, which can orchestrate the execution of kernels and control access to shared resources. The global scheduler is controlled via a set of memory-mapped registers, which are implementation specific. This approach allows for a range of

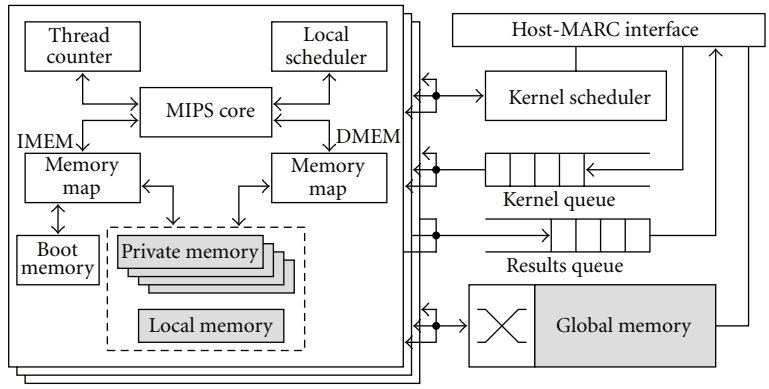

Figure 7: Schematic of a MARC machine's implementation.

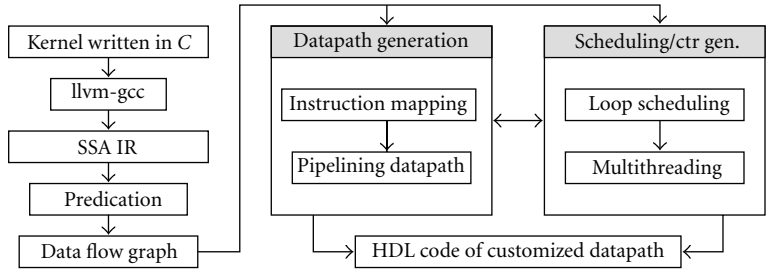

Figure 8: CAD flow of synthesizing application-specific processing cores.

schedulers, from simple round-robin or priority schedules to complex problem-specific scheduling algorithms.

The MARC machine optimized for Bayesian inference uses the global scheduler to dispatch threads at a coarse grain (ganging up thread starts). The use of the global scheduler is therefore rather limited as the problem does not greatly benefit from a globally aware approach to scheduling.

5.2.7. System Interconnect. One of the key advantages of reconfigurable computing is the ability to exploit application-specific communication patterns in the hardware system. MARC allows the network to be selected from a library of various topologies, such as mesh, H-tree, crossbar, or torus. Application-specific communication patterns can thus be exploited by providing low-latency links along common routes.

The MARC machine explores two topologies: a pipelined crossbar and a ring, as shown in Figure 9. The pipelined crossbar contains no assumptions about the communication pattern of the target application—it is a nonblocking network that provides uniform latency to all locations in the global memory address space. Due to the large number of endpoints on the network, the crossbar is limited to 120 MHz with 8 cycles of latency.

The ring interconnect only implements nearestneighbor links, thereby providing very low-latency access to some locations in global memory, while requiring multiple hops for other accesses. Nearest neighbor communication is important in the Bayesian inference accumulation phase and helps reduce overall latency. Moreover, this network topology is significantly more compact and can be clocked at a much higher frequency—approaching 300 MHz in our implementations. The various versions of our MARC machine, therefore, made use of the ring network because of the advantages it has shown for this application.

5.2.8. Mapping Bayesian Inference onto the MARC Machine. The order-graph sampler comprises a C-core for the serial control logic and A-cores to implement the *score*() calls. Per iteration, the C-core performs the node swap operation, broadcasts the proposed order, and applies the Metropolis-Hastings check. These operations consume a negligible amount of time relative to the scoring process.

Scoring is composed of (1) the parent set compatibility check and (2) an accumulation across all compatible parent sets. Step 1 must be made over every parent set; its performance is limited by how many parent sets can be simultaneously accessed. We store parent sets in on-chip RAMs that serve as A-core private memory and are therefore limited by the number of A-cores and attainable A-core throughput. Step 2 must be first carried out independently by each A-core thread, then across A-core threads, and finally across the A-cores themselves. We serialize cross-thread and cross-core accumulations. Each accumulation is implemented with a global memory access.

The larger order-graph sampler benchmark we chose (see Section 7) consists of up to 37 nodes, where each of the nodes has 66712 parent sets. We divide these 66712 elements into 36 chunks and dedicate 36 A-cores to work on this data set. After completion of the data processing for one node, data from the next node is paged in, and we restart the A-cores.

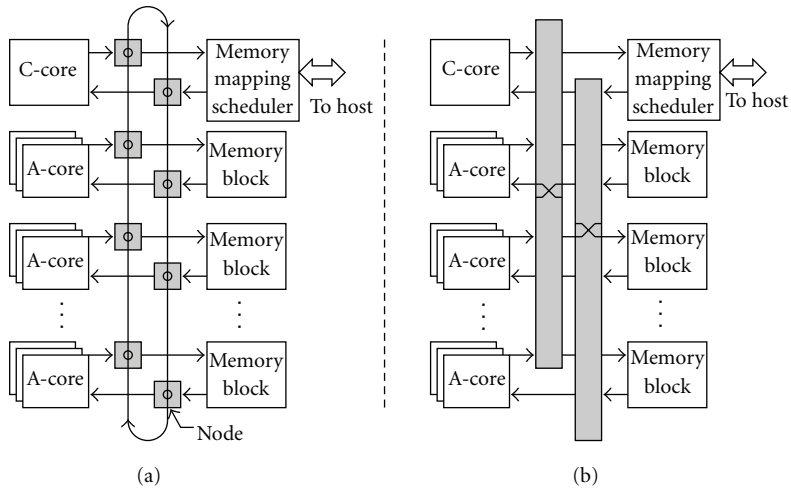

Figure 9: System diagram of a MARC system. (a) Ring network. (b) Pipelined crossbar.

6. Hardware Prototyping

For this research, both the hand-optimized design and MARC machines are implemented targeting a Virtex-5 (XCV5LX155T-2) of a BEEcube BEE3 module for FPGA prototyping. We also evaluate how each design performs when mapped through a standard ASIC design flow targeting a TSMC 65 ns CMOS process. A design point summary, that we will develop over the rest of the paper, is given in Table 1.

The local memory or "RAMs", used in each design point, were implemented using block RAMs (BRAMs) on an FPGA and generated as SRAM (using foundry-specific memory generators) on an ASIC. All of our design points benefit from as much local memory read bandwidth as possible. We increased read bandwidth on the FPGA implementation by using both ports of each BRAM block and exposing each BRAM as two smaller single-port memories. For the ASIC platform, the foundry-specific IP generator gives us the capability to create small single-ported memories suitable for our use.

In addition to simple memories, our designs used FIFOs, arbiters, and similar hardware structures to manage flow control and control state. On an FPGA, most of these blocks were available on the Virtex-5 through Xilinx Coregen while the rest were taken from the GateLib library. On an ASIC, all-of these blocks were synthesized from GateLib Verilog or generated using foundry tools.

To obtain all FPGA measurements, we designed in Verilog RTL and mapped the resulting system using Synplify Pro (Synopsys) and the Xilinx ISE flow for placement and routing (PAR). To obtain ASIC measurements, we used a standard cell-bawed Synopsis CAD flow including Design Compiler and IC Compiler.

No manual placement or hierarchical design was used for our studies. We verified the resulting system post-PAR by verifying (a) timing closure, and (b) functionality of the flattened netlist. The tools were configured to automatically retime the circuit to assist timing closure, at the expense of hard-ware resources. It is worth noting that the automatic retiming did not work as well with the MARC multithreaded cores because of a feedback path in the core datapath. Therefore, manual retiming was required for performance improvement with the MARC multithreaded design points.

6.1. Hand-Optimized Configurations. On the FPGA platform, the best performing configurations were attained when using 48 cores per FPGA running at a 250 MHz core clock and 36 cores at 300 MHz (the former outperforming the latter by a factor of 1.1 on average). Both of these points were used between 65% and 75% of available device LUTs and exactly 95% of device BRAMs. We found that implementing 48 cores at 300 MHz was not practical due to routing limitations and use of the 48 core version at 250 MHz for the rest of the paper.

For the ASIC implementation, because performance is a strong function of the core clock's frequency, we optimize the core clock as much as possible. By supplying the Verilog RTL tp the Synopsys Design Compiler with no prior optimization, the cores can be clocked at 500 MHz. Optimizing the core clock requires shortening the critical path, which is in the datapath. By increasing the number of threads from 4 to 8 and performing manual retiming for the multithreaded datapath, the core clock achieves 1 GHz.

6.2. MARC Configurations. The MARC implementation comprises one C-core and 36 A-cores. While the C-core in all MARC machines is a fully bypassed 4-stage RISC processor, MARC implementations differ in their implementation of the A-cores. For example, fine-grained multithreaded RISC cores, automatically generated application-specific datapaths, and multithreaded versions of the generated cores are all employed to explore different tradeoffs in design effort and performance. To maintain high throughput, the better performing A-cores normally execute multiple concurrent threads to saturate the long cycles in the application dataflow graph.

6.2.1. Memory System. As in other computing platforms, memory accesses significantly impact the overall performance of a MARC system. In the current MARC implemen-

Table 1: A-core counts, for all design points, and a naming convention for all MARC configurations used in the study. If only one core count is listed, it is the same for both 32 and 37 node ($32n$ and $37n$) problems (see Section 7). All A-core counts are given for area normalized designs, as discussed in Section 7.

Alias	Description	Number of cores ($32n$, $37n$)
Hand design FPGA	—	48
Hand design ASIC	—	2624, 2923
MARC-Ropt-F	RISC A-core with optimized network on FPGA	36
MARC-C1-F	Customized A-core on FPGA	36
MARC-C2-F	Customized A-core (2-way MT) on FPGA	36
MARC-C4-F	Customized A-core (4-way MT) on FPGA	36
MARC-Ropt-A	RISC A-core with optimized network on ASIC	1269, 1158
MARC-C1-A	Customized A-core on ASIC	1782, 1571
MARC-C2-A	Customized A-core (2-way MT) on ASIC	1768, 1561
MARC-C4-A	Customized A-core (4-way MT) on ASIC	1715, 1519
GPGPU	—	512

tation, private or local memory accesses take exactly one cycle, while global memory accesses typically involve longer latencies that are network dependent. We believe that given different applications, such discrepancies between local and global memory access latencies provide ample opportunities for memory optimization and performance improvements. For example, the MARC machine in this work has been optimized for local memory accesses, reflecting the needs of the Bayesian inference algorithm.

6.2.2. Clock Speed and Area on FPGA and ASIC. For better throughput, we have implemented single-threaded, two-way multithreaded, and four-way multithreaded application-specific A-cores for MARC devices. When individually instantiated on the Virtex-5 FPGA, these cores are clocked at 163 MHz, 234 MHz, and 226 MHz, respectively. There is a decrease in clock frequency when the number of threads is changed from two to four. This is due to the increased routing delay to connect LUT-FF pairs further apart in a larger physical area. When the completely assembled MARC machines traverse the hardware generation flow, the cores' clock frequency decreases further to 144 MHz, 207 MHz, and 206 MHz, respectively due to added FPGA resource utilization. The same A-cores are used for the ASIC implementation, where they operate at 526 MHz, 724 MHz, and 746 MHz, respectively. Due to a higher degree of freedom in ASIC place and route, we do not see the performance dip observed when the two-threaded FPGA implementation is changed to four-threaded. However, it is apparent that despite the decrease in levels of logic in the critical path, it is difficult to squeeze out more performance by simple register insertion and retiming.

With respect to area, the overhead of multithreading is more pronounced on an FPGA relative to an ASIC. For the 37 node benchmark, the MARC machines with single, two-way, and four-way multithreaded customized A-cores utilize 47%, 65%, and 80% of the flip-flops on Virtex-5. Since they operate on the same amount of data, 85% of BRAMs are used for each of the three design points. Meanwhile, on an ASIC we only observe an area increase from 6.2 mm^2 in the single-threaded case to 6.4 mm^2 for the four-way multithreaded design. This is because the ASIC implementation exposes the actual chip area, where the increase in number of registers is dwarfed by the large SRAM area.

7. Performance and Area Comparisons

We compare the performance of the hand-optimized design and the MARC machines on FPGA as well as ASIC platforms. For both the hand-optimized and the MARC implementations on an ASIC, we normalize our area to the FPGA's die area. FPGA die area was obtained by X-ray imaging the packaged dies and estimating the die area from the resulting photographs. For the remainder of the paper, all devices whose die areas and process nodes are relevant are given in Table 2.

For the FPGA designs, we packed the device to its limits without performance degradation. Effectively, the designs are consuming the entire area of the FPGA. We then performed a similar evaluation for the ASIC platform by attempting to occupy the same area as an FPGA. This is achieved by running the design for a small number of cores and then scaling up. This technique is valid as the core clock is not distributed across the network, and the network clock can be slow (50–100 MHz) without adversely affecting performance.

The specific Bayesian network instances we chose consist of 32 and 37 nodes, with dataset of 36457 and 66712 elements, respectively. The run times on each hardware platform are shown in Tables 4 and 5, for the 32 and 37 node-problem, respectively. The execution time for each platform is also normalized to the fastest implementation—hand-optimized design on ASIC—to show the relative performance of every design point.

7.1. Benchmark Comparison. The large gap between the amount of data involved in the two problems gives each distinct characteristics, especially when mapped to an ASIC platform. Because data for the 32 node problem can fit on an ASIC for both MARC and the hand-optimized design, the problem is purely compute bound. The hand-optimized solution benefits from the custom pipelined accumulation

Table 2: Device die areas and process nodes.

Device	Die area (mm^2)	Process (nm)
Virtex-5 LX155T FPGA	270	65
Nvidia GeForce GTX 580	520	40

and smaller and faster cores, resulting in its 2.5x performance advantage over the best MARC implementation. The 37 node problem, on the other hand, could not afford to have the entire dataset in the on-chip SRAMs. The required paging of data into the on-chip RAMs becomes the performance bottleneck. Having exactly the same DRAM controller as the MARC machines, the hand-optimized design only shows a small performance advantage over MARC, which can be attributed to its clever paging scheme. For the FPGA platform, both the 32 and 37 node problems involve paging of data, but as the run time is much longer, data transfer only accounts for a very small fraction of the execution time (i.e., both problems are compute bound).

7.2. MARC versus Hand-Optimized Design. For compute-bound problems, it is clear that MARC using RISC instruction processors to implement A-cores achieves less than 2% of the performance exhibited by the hand-optimized implementation, even with optimized interconnect topology (a ring network versus a pipelined crossbar). Customizing the A-cores, however, yields a significant gain in performance, moving MARC to within a factor of 4 of the performance of the hand-optimized implementation. Further optimizing the A-cores through multithreading pushes the performance even higher. The best performing MARC implementation is within a factor of 2.5 of the hand-optimized design and corresponds to two-way multithreaded A-cores. Like the FPGA platform, further increase to four threads offers diminishing returns and is outweighed by the increase in area, and therefore the area-normalized performance actually decreases.

7.3. Cross-Analysis against GPGPU. We also benchmark the various hardware implementations of the order-graph sampler against the GPGPU reference solution, running on Nvdia's GeForce GTX 580.

As the GTX 580 chip has a much larger area than Virtex-5 FPGA and is also on 40 nm process rather than 65 nm, we scaled its execution time according to the following equations, following Table 2:

$$\text{Scaled Area}_{\text{GPU}} = \text{Area}_{\text{GPU}} * S^2 = 520 * \left(\frac{65}{40}\right)^2 = 1373, \tag{4}$$

$$T_{\text{scaled}} = \frac{\text{Scaled Area}_{\text{GPU}}}{\text{Area}_{\text{FPGA}}} * S * T = 8.264 * T. \tag{5}$$

To make sure the comparison is fair, the technology scaling [24] takes into account the absolute area difference between the GPU and FPGA, as well as the area and delay scaling (i.e., S, the technology scaling factor) due to different processes. Our first assumption is that the performance scales linearly with area, which is a good approximation due to our Bayesian network problem and device sizes. Second, we assume zero wire slack across both process generations for all designs. The original and scaled execution times are displayed in Table 3.

Table 3: Scaled GPGPU design for 65 nm process.

Problem	Per-iteration time 40 nm (μs)	Scaled per-iteration time 65 nm (μs)
32-Node	21.0	174
37-Node	37.8	312

It can be seen from Tables 4 and 5 that MARC on FPGA can achieve the same performance as the GPGPU when application-specific A-cores are used. With multithreading, the best MARC implementation on FPGA can achieve more than a 40% performance advantage over the GPGPU. Hand-optimized designs, with more customization at the core and network level, push this advantage even further to 3.3x. The reason for this speedup is that each iteration of the inner loop of the *score*() function takes 1 cycle for A-cores on MARC and the hand-optimized design, but 15 to 20 cycles on GPGPU cores. It is apparent that the benefit from exploiting loop level parallelism at the A-cores outweighs the clock frequency advantage that the GPGPU has over the FPGA.

When an ASIC is used as the implementation platform, the speedup is affected by the paging of data as explained in Section 7.1. For the 32 node problem where paging of data is not required, the best MARC implementation and the hand-optimized design achieve 156x and 412x performance improvement over the GPGPU, respectively. For the 37 node problem, which requires paging, we observe a 69x and 84x performance advantage from the best MARC variant and hand-optimized implementation, respectively. Using only a single dual channel DRAM controller, we have about 51.2 Gb/sec of memory bandwidth for paging. However the GPGPU's memory bandwidth is 1538.2 Gb/sec—30x that of our ASIC implementations. As a result, the GPGPU solution remains compute bound while our ASIC implementations are getting constrained by the memory bandwidth. Thus, the performance gap between the 32 and 37 node problems is because of the memory-bound nature of our ASIC implementations.

It is also interesting that the MARC implementation with RISC A-cores on ASIC is about 6 times faster for the 32 node problem and 8 times faster for 37 node problem, compared to the GPGPU. With both MARC RISC A-cores and GPGPU cores, the kernel is executed as sequence of instructions rather than by a custom datapath. In addition, the clock frequency gap between MARC on ASIC and the GPGPU is small. We claim that the performance gap is due to the application-specific nature of the MARC design—MARC is able to place more cores per unit area (see Table 1) while still satisfying the requirements of local caching. In addition, the network structure in MARC machines is also optimized to the Bayesian inference accumulation step. The combined effect results in a significantly better use of chip area for this application.

Table 4: 32-Node. Performance comparison between MARC, hand-optimized, and GPGPU.

Configuration	Per iteration Time (μs)	Relative Perf.
GPGPU scaled reference	174	0.0024
MARC-Ropt-F	2550	0.0002
MARC-C1-F	172	0.0025
MARC-C2-F	124	0.0034
MARC-C4-F	136	0.0031
Hand design FPGA	51.4	0.0082
MARC-Ropt-A	27.6	0.0152
MARC-C1-A	1.47	0.2863
MARC-C2-A	1.11	0.3808
MARC-C4-A	1.17	0.3608
Hand design ASIC	0.422	1.0000

Table 5: 37-Node. Performance comparison between MARC, Hand-optimized, and GPGPU.

Configuration	Per iteration Time (μs)	Relative Perf.
GPGPU scaled reference	312	0.0119
MARC-Ropt-F	5130	0.0007
MARC-C1-F	310	0.0120
MARC-C2-F	221	0.0169
MARC-C4-F	235	0.0158
Hand design FPGA	110	0.0339
MARC-Ropt-A	38.1	0.0978
MARC-C1-A	5.02	0.7429
MARC-C2-A	4.53	0.8231
MARC-C4-A	4.61	0.8083
Hand design ASIC	3.73	1.0000

8. Conclusion

MARC offers a methodology to design FPGA and ASIC-based high-performance reconfigurable computing systems. It does this by combining a many-core architectural template, high-level imperative programming model [19], and modern compiler technology [22] to efficiently target both ASICs and FPGAs for general-purpose, computationally intensive data-parallel applications.

The primary objective of this paper is to understand whether a many-core architecture is a suitable abstraction layer (or execution model) for designing ASIC and FPGA-based computing machines from an OpenCL specification. We are motivated by recently reemerging interest and efforts in parallel programming for newly engineered and upcoming many-core platforms, and feel that if we can successfully build an efficient many-core abstraction for ASICs and FPGAs, we can apply the advances in parallel programming to high-level automatic synthesis of computing systems. Of-course, constraining an execution template reduces degrees of freedom for customizing an implementation using application-specific detail. However, we work under the hypothesis that much of the potential loss in efficiency can be recovered through customization of a microarchitectural template designed for a class of applications using application-specific information. The study in this paper represents our initial effort to quantify the loss in efficiency incurred for a significant gain in design productivity for one particular application.

We have demonstrated via the use of a many-core microarchitectural template for OpenCL that it is at least sometimes possible to achieve competitive performance relative to a highly optimized solution and to do so with considerable reduction in development effort (days versus months). This approach also achieves significant performance advantage over a GPGPU approach—a natural platform for mapping this class of applications. In this study, the most significant performance benefit came from customization of the processor cores to better fit the application kernel—an operation within reach of modern high-level synthesis flows.

Despite these results, the effectiveness of MARC in the general case remains to be investigated. We are currently limited by our ability to generate many high-quality hand-optimized custom solutions in a variety of domains to validate and benchmark template-based implementations. Nonetheless, we plan to continue this study, exploring more application domains, extending the many-core template tailored for OpenCL and exploring template microarchitectures for other paradigms. We are optimistic that a MARC-like approach will open new frontiers for rapid prototyping of high-performance computing systems.

Acknowledgments

The authors wish to acknowledge the contributions of the students, faculty, and sponsors of the Berkeley Wireless Research Center and the TSMC University Shuttle Program. This work was funded by the NIH, Grant no. 1R01CA130826-01 and the Department of Energy, Award no. DE-SC0003624.

References

[1] M. Lin, I. Lebedev, and J. Wawrzynek, "Highthroughput Bayesian computing machine with reconfigurable hardware," in *Proceedings of the 18th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '10)*, pp. 73–82, ACM, Monterey, California, USA, 2010.

[2] M. Lin, I. Lebedev, and J. Wawrzynek, "OpenRCL: from sea-of-gates to sea-of-cores," in *Proceedings of the 20th IEEE International Conference on Field Programmable Logic and Applications*, Milano, Italy, 2010.

[3] Wikipedia, "C-to-hdl," November 2009, http://en.wikipedia.org/wiki/C_to_HDL/.

[4] M. Gokhale and J. Stone, "Napa c: compiling for a hybrid risc/fpga architecture," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, Napa, Calif, USA, 1998.

[5] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "Garp architecture and C compiler," *Computer*, vol. 33, no. 4, pp. 62–69, 2000.

[6] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, "Spatial computation," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI '04)*, pp. 14–26, New York, NY, USA, October 2004.

[7] J. Wawrzynek, D. Patterson, M. Oskin et al., "RAMP: research accelerator for multiple processors," *IEEE Micro*, vol. 27, no. 2, pp. 46–57, 2007.

[8] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and M. W. Hwu, "Fcuda: enabling efficient compilation of cuda kernels onto fpgas," in *Proceedings of the 7th IEEE Symposium on Application Specific Processors (SASP '09)*, San Francisco, Calif, USA, 2009.

[9] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of platform architectures from opencl programs," in *Proceedings of the 19th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '11)*, Salt Lake City, Utah, USA, 2011.

[10] J. Friedman, T. Hastie, and R. Tibshirani, "Sparse inverse covariance estimation with the graphical lasso," *Biostatistics*, vol. 9, no. 3, pp. 432–441, 2008.

[11] D. Heckerman, D. Geiger, and D. M. Chickering, "Learning Bayesian networks: the combination of knowledge and statistical data," *Machine Learning*, vol. 20, no. 3, pp. 197–243, 1995.

[12] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann, San Francisco, Calif, USA, 1988.

[13] C. Fletcher, I. Lebedev, N. Asadi, D. Burke, and J. Wawrzynek, "Bridging the GPGPU-FPGA efficiency gap," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*, pp. 119–122, New York, NY, USA, 2011.

[14] N. Bani Asadi, C. W. Fletcher, G. Gibeling et al., "Paralearn: a massively parallel, scalable system for learning interaction networks on fpgas," in *Proceedings of the 24th ACM International Conference on Supercomputing*, pp. 83–94, ACM, Ibaraki, Japan, 2010.

[15] D. M. Chickering, "Learning Bayesian Networks is NP-Complete," in *Learning from Data: Artificial Intelligence and Statistics V*, pp. 121–130, Springer, New York, NY, USA, 1996.

[16] B. Ellis and W. H. Wong, "Learning causal Bayesian network structures from experimental data," *Journal of the American Statistical Association*, vol. 103, no. 482, pp. 778–789, 2008.

[17] M. Teyssier and D. Koller, "Ordering-based search: a simple and effective algorithm for learning Bayesian networks," in *Proceedings of the 21st Conference on Uncertainty in AI (UAI '5)*, pp. 584–590, Edinburgh, UK, July 2005.

[18] N. Friedman and D. Koller, "Being Bayesian about network structure," in *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pp. 201–210, Morgan Kaufmann, San Francisco, Calif, USA, 2000.

[19] Khronos OpenCL Working Group, The OpenCL Specification, version 1.0.29, December 2008, http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

[20] M. Lin, I. Lebedev, and J. Wawrzynek, "OpenRCL: low-power high-performance computing with reconfigurable devices," in *Proceedings of the 18th International Symposium on Field Programmable Gate Array*, 2010.

[21] NVIDIA OpenCL Best Practices Guide, 2009, http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf.

[22] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)*, pp. 75–86, Palo Alto, Calif, USA, March 2004.

[23] G. Gibeling et al., "Gatelib: a library for hardware and software research," Tech. Rep., 2010.

[24] J. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits*, chapter 5, Prentice Hall, New York, NY, USA, 2nd edition, 2003.