BLAST Implementation on BEE2

By

Chen Chang

Electrical Engineering and Computer Science
University of California at Berkeley

TABLE OF CONTENTS

ABSTRACT **4**

1 INTRODUCTION **4**

2 BLAST ALGORITHM AND EXISTING SOLUTIONS **5**

3 BEE2 SYSTEM **6**

3.1 BASIC COMPUTING ELEMENTS 7
3.2 COMPUTE NODE 8
3.3 GLOBAL COMMUNICATION NETWORKS 10

4 BLAST PERFORMANCE ON BEE2 .... **11**

4.1 BLAST IMPLEMENTATION ON BEE2 11
4.1.1 Algorithmic description of BLAST implementation on BEE2. .... *12*
4.1.2 Analytical model for memory usage and accesses *15*
4.1.3 Circuit implementation *15*
4.1.4 Database partitioning *17*
4.2 MEMORY PERFORMANCE TUNING 17
4.3 OVERALL PERFORMANCE 23
4.3.1 Simulation environment and results *23*
4.3.2 Performance Comparisons *24*

5 CONCLUSION **26**

6 APPENDIX A: FPGA COMPUTING TECHNOLOGY **27**

6.1 THE OBSOLESCENCE OF VON NEUMANN ERA STRATEGIES 27
6.2 THE FPGA SOLUTION 28
6.3 BERKELEY EMULATION ENGINE PLATFORM 30

7 REFERENCE **31**

TABLE OF FIGURES

FIGURE 1: BASIC COMPUTING ELEMENT ... 7
FIGURE 2: DDR2 ADDITIVE LATENCY ... 8
FIGURE 3: COMPUTE NODE CONNECTIVITY ... 9
FIGURE 4: GLOBAL COMMUNICATION NETWORK OVERVIEW ... 10
FIGURE 5: 4-ARY COMMUNICATION TREE NETWORK ... 10
FIGURE 6: SEQUENCE ALIGNMENT EDIT GRAPH ... 11
FIGURE 7: K-MER INDEXING ... 12
FIGURE 8: HITS INDEXING ... 13
FIGURE 9: CIRCUIT IMPLEMENTATION DIAGRAM ... 16
FIGURE 10: MEMORY SUBSYSTEM OVERVIEW FOR A SINGLE COMPUTE ELEMENT ... 18
FIGURE 11: PHYSICAL MEMORY ADDRESS FIELDS ... 18
FIGURE 12: CONSTANT STRIDE MEMORY ACCESS PERFORMANCE ... 20
FIGURE 13: RANDOM MEMORY ACCESS LATENCY HISTOGRAM ... 20
FIGURE 14: RANDOM MEMORY ACCESS PERFORMANCE ... 21
FIGURE 15: ATOMIC MEMORY ACCESS BANDWIDTH UTILIZATION ... 22
FIGURE 16: VIRTUAL PORT WIDTH EFFECT ON MEMORY THROUGHPUT ... 22
FIGURE 17: EXCUTION TIME (300KB QUERY, 1.2GB DATABASE) ... 24
FIGURE 18: PARALLEL SPEEDUP (300KB QUERY, 1.2GB DATABASE) ... 24
FIGURE 19: PERFORMANCE COMPARISON TO MPIBLAST ON GREEN DESTINY CLUSTER ... 25
FIGURE 20: PERFORMANCE COMPARISON TO TIMELOGIC DECYPHER ... 25
FIGURE 21: COMPUTATIONAL DENSITY AND POWER EFFICIENCY COMPARISION ... 27
FIGURE 22: COMPUTATIONAL DENSITY COMPARISON BETWEEN FPGA AND INTEL PROCESSORS ... 29
FIGURE 23: BEE FPGA ARRAY & TOPOLOGY DIAGRAM ... 30

TABLE OF TABLES

TABLE 1: BLAST SEARCH TYPES ... 5
TABLE 2: XILINX VIRTEX 2 PRO FAMILY FPGA MAJOR FEATURE COMPARISON ... 7
TABLE 3: PSEUDO-CODE FOR STEP 1 ... 13
TABLE 4: PSEUDO-CODE FOR STEP 2 ... 14
TABLE 5: PSEUDO-CODE FOR STEP 3 ... 14
TABLE 6: MEMORY BANDWIDTH EFFICIENCY ... 23

TABLE OF EQUATIONS

EQUATION 1: MEMORY CAPACITY REQUIREMENTS ... 15
EQUATION 2: NUMBER OF MEMORY ACCESSES ... 15

Abstract

Most existing biosequence alignment software tools, such as BLAST (Basic Local Alignment Search Tool) [1], are designed to run on commodity PC clusters. With the exponential growth of the biosequence databases, such the GenBank [2], the computational need for sequence comparisons have far exceeded the computing capacity provided by the semiconductor industry, as stated in Moore's Law. Therefore the current dominating solution is simply adding more and more computer nodes in the cluster to keep the execution time tolerable. However this approach also significantly increases the cost of computer cluster equipment, maintenance, and administration. This paper explores a novel approach to the biosequence alignment problem by using a FPGA (Field Programmable Gate Array) based computing system—BEE2, which provides 100~1000 times faster execution time running the BLAST algorithm, and over 1000 times lower price-performance ratio than existing PC cluster solutions.

1 Introduction

Since most biosequence, such as DNA or protein, have a short sequence alphabet set (4 for DNA, 20 for protein), which can be represented with a short integer of only several bits (2 for DNA, and 5 for protein), the computational precision required by the alignment algorithm are exclusively low precision integer operations. This typically under utilize the CPU processing power by a large margin, because the commercial CPUs are designed to best handle 32 or 64 bit integer and floating point operations. Microprocessors typically provides very limited data-level parallelism (typical less than half a dozen ALUs per processor), but high clock rate to ensure a fast sequential execution time. However, sequence comparison algorithms exhibit a much higher degree of data parallelism, typically hundreds if not thousands of operations can be performed in parallel, which cannot be exploited by existing microprocessor based computer systems. In addition, each sequence symbol comparison only require a few operational cycles by the ALU, but the sequence data need to be moved on and off the processor chip at the matching rate to keep up with the multi-gigahertz processor, hence stressing the memory subsystem bandwidth and latency.

In contrast, custom computing machines made with FPGA chips can be configured to match exactly the computation precision requirement. Internally each FPGA chips consists of up to 100,000 logic elements (look-up tables, a.k.a, LUTs), each can be programmed to performance any 4-bit input with1-bit output logic operations, and arbitrarily interconnected to form larger circuits, such as ALUs, controllers, etc. State-of-the-art Xilinx Virtex 2 Pro platform FPGA [3] also includes two on-die PowerPC 400MHz processor cores for elaborate control mechanisms, over 400 dedicated 18-bit multipliers, and up to 1 MB of on-chip SRAM. By combining platform FPGA with large number of independently addressable external DRAM modules and ultra-fast I/O interface to hard disks, such high-performance reconfigurable computer system can performance up to 5 trillion DNA base pair comparisons per second per FPGA chip.

Existing commercial companies, such as Timelogic [4], only provide fixed algorithms on proprietary FPGA platform at a very high price. The lack of programmability and high initial equipment cost limited the adoption of these systems to only a few isolated cases. At UC Berkeley, the BEE2 project is currently building a standardized FPGA reconfigurable computer platform, which offers a flexible and convenient programming interface through Matlab. We believe that biosequence alignment problem can be very efficient solved on the BEE2 platform, and providing the end users the flexibility of customized search algorithm as needed.

This paper attempts to demonstrate the effectiveness of the BEE2 computer system for sequence alignment problem using the BLAST algorithm. Since the physical hardware of BEE2 platform is still under development at the time of writing, simulated performance are used for analysis with a cycle accurate simulator implemented under Matlab, which include the accurate cycle behavior of the FPGA computing fabric, external DDR2 DRAM, and communication network. Synthetics data sets will be used as inputs to the BLAST application and performance for comparison between the BEE2 and typical microprocessor clusters to calculate speedup as well as identify potential performance bottlenecks.

In the remaining of this paper, first a brief overview of the BLAST algorithm is provided in section 2 along with a short survey of existing solutions for parallelizing the algorithm on a variety of hardware computing platforms. Section 3 explains in detail the BEE2 hardware architecture. Then in section 4, the particular BLAST implementation on BEE2 is explain and analyzed, as well as specific performance tunings achieved with the BEE2 system, followed by performance comparison to existing PC cluster solutions. Finally section 5 concludes the paper with future direction of the research. For readers unfamiliar with the FPGA computing technology, a short overview is provided in Appendix A.

2 BLAST algorithm and existing solutions

The BLAST family of sequence similarity search algorithms serves as the foundation for many computational biology researches. The BLAST algorithms search for local similarities between a short query sequence and a large database of either DNA or amino acid sequences. Newly discovered sequences are commonly searched against a database of known DNA or amino-acid sequences, reporting any statistically significant similarities, which facilitates identifying the function of the new sequence. Other uses of BLAST searches include phylogenetic profiling and pair wise genome alignment.

BLAST provides functionality for comparing all possible combinations of query and database sequence types by translating the sequences on the fly. Table 1 lists the types of searches on each possible combination of query versus database type.

Table 1: BLAST search types

BLAST type	*Query type*	*Database type*	*Translates*
Blastn	Nucleotide	Nucleotide	None
Blastp	Peptide	Peptide	None
Blastx	Nucleotide	Peptide	Query
Tblastn	Peptide	Nucleotide	Database
Tblastx	Nucleotide	Nucleotide	both

Nevertheless, the algorithms for each type of search operate almost identically. The BLAST search heuristic indexes both the query and database sequence into k-mers of a chosen size (11 nucleotides or 3 residues by default). It then searches for matching k-mer pairs (hits) with a score of at least T and extends the match along the diagonal, until the current score drops more than D value below the maximum score achieved along the extension. The threshold (D value) is 20 by default. Standard BLAST output consists of a set of local alignments found within each query sequence, the alignment's score, an alignment of the query and database sequences, and a measure of the likelihood that the alignment is a random match between the query and database (e-value).

Existing approaches to speed up the BLAST execution time can be classified into two categories: the software divide-and-concur and the hardware acceleration.

The software divide-and-concur approaches recognize the embarrassingly parallel nature of the BLAST searches, where typically users submit a set of queries over a large database. The queries can be segmented and distributed to each compute nodes in a cluster, and processed in parallel with the database duplicated on each node. However, for databases too large to fit in the physical memory of each compute node, each query still runs slowly due to virtual memory swapping to hard disks. The alternative approach is to segment the database instead, then distribute all queries to all the compute nodes, and finally merge the partial results from each compute nodes for the particular query into the final result. One of the most commonly used implementation of this approach is the mpiBLAST implementation [5], which provides a MPI wrapper around the standard NCBI BLAST software package [6], and automatically handles the database segmentation, query distribution, and results merging tasks. As reported in [7], the mpiBLAST implementation can achieve linear and even super-linear speedup for queries over large databases. However, all of the software approaches are fundamentally limited by the sequential processing nature of the microprocessor and its limited memory capability to transfer data between the processor and external DRAM. Furthermore, the divide-and-concur approaches does not work efficiently for pair-wise comparison on long sequences, especially in the case of direct genome comparison between two or more species

On the contrary, the hardware acceleration approaches focus on the low-level massive data parallelism in the BLAST algorithm, where a large number of systolic-type sequence comparisons can be performance concurrently on specialized VLSI chips or reconfigurable logic arrays. One of the first massively parallel VLSI implementation of the BLAST algorithm is the BioScan system [8][9]. Each system consists of 16 chips; each with 812 PEs running at 32 MHz fabricated in 1.2 um CMOS technology. The system is preloaded with a weight table scaled according to sequence length before performing the comparison. Each chip can process up to 1.88 million characters per second. Another similar VLSI implementation is the BISP system, which implements full hardware dynamic programming [10]. More recently, TimeLogic has commercialized an FPGA-based accelerator called the DeCypher BLAST hardware accelerator. As stated on the TimeLogic web site [11], a single DeCypher accelerator module consists of 4 FPGAs can provide 200 times faster execution time than an 8 CPU Sun Ultra Sparc-III (750MHz) server for all known bacterial proteins (4,289 proteins sequences totaling 1,358,990 symbols from NCBI's E. coliK12 U00096.faa dataset) were searched against 192 bacterial genomes (775,191,168 symbols in 6- frames).

Despite the impressive speed up provided by the specialized hardware accelerators, they generally suffer from high system cost, mainly due to the high non-recurring engineering cost associated with the design of specialized VLSI chips, hardware and software system. In addition, these systems are designed to only handle a very narrow set of algorithms and applications, which lack the flexibility provided by general purpose computer clusters. When factoring the long term administration and maintenance cost, the overall price-performance ratio of these specialized systems is no longer very competitive comparing to the commodity computer cluster solution.

3 BEE2 System

Contrary to specialized hardware accelerators, the BEE2 system is design from ground up to be general purpose computing system that minimizes the overall price-performance ratio. All

aspects of the system are design to leverage as much commodity hardware and software components as possible; meanwhile providing full programmability and flexibility as general purpose computing cluster.

BEE2 hardware system design, just like any other large parallel computer systems, consists of three primary components: processing elements, memory elements, and interconnects. On the system level, processing elements are the FPGA chips; memory elements are the external DRAM modules locally attached to each of the FPGA; interconnects consists of local connections, which links local FPGAs on the same PCB board, as well as global connections that link multiple boards into a unified system. The main difference of the BEE2 design from traditional parallel computer system design is that the processing elements are FPGA chips rather than microprocessors. In addition to the primary components, BEE2 also incorporate a range of secondary system components, including bootstrap, clock distribution, power regulation, and thermal regulation. They support and monitor the primary components to ensure proper operation of the overall system.

The primary system refers to the processing elements, memory elements, and interconnects. The BEE2 hardware organize these components on three levels of hierarchy: basic computing elements, compute nodes, and global communication networks.

3.1 Basic Computing Elements

As shown in Figure 1: basic computing element, each basic computing element (BCE) consists of one Xilinx Virtex 2 Pro FPGA chip and four DDR2 240-pin DRAM DIMMs, each has a capacity from 0.5 to 2GB.

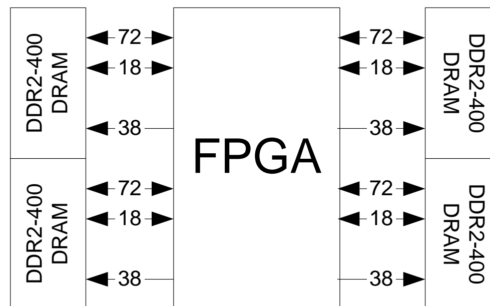


Figure 1: basic computing element

Due to the large number of I/O pins required to simultaneous accommodate both four independent banks of DRAMs and the high bandwidth inter-FPGA connections; we use the largest FPGA package available for the Virtex 2 Pro family, the FF1704 Flip-Chip Fine-Pitch BGA Package. Currently two chips can use this package, the XC2VP70 and XC2VP100. Table 2 compares the feature difference between the two chips.

Table 2: Xilinx Virtex 2 Pro Family FPGA Major Feature Comparison

Features	*Virtex 2 Pro 70*	*Virtex 2 Pro 100*
Logic cells	74,448	99,216
PowerPC cores	2	2
MGTs	20	20
Block RAM (Kbits)	5,904	7,992
18bit Multipliers	328	444
DCMs	8	12

Max User I/Os	996	1,040

In comparison to the original BEE design, where each FPGA only has 1 extern 1 MB ZBT-SRAM chip running at 100MHz with a 32-bit data interface, on the BEE2 system, each FPGA has four independent DRAM channels, each running at 200MHz (400DDR) with a 72-bit data interface (for non-ECC 64-bit data width). Therefore aggregate to a peak memory bandwidth of 12.8 GBps per FPGA—32 times more than the BEE external memory bandwidth. However, each memory access in general has to incur both RAS and CAS latency of about 6 cycles, which is twice the time of SRAM latency. Although DRAM, with its large memory capacity, is preferred in most of the computing applications, QDR SRAM modules are the better choice for latency sensitive applications that do not require DRAM capacity, such as network processing applications. To accommodate these applications, special QDR SRAM DIMMs with the same DDR2 DRAM 240-pin DIMM form factor can be used to connect to the FPGA.

The choice of using DDR2 memory instead of regular DDR is mainly due to the additional features offered by DDR2 memory, such as 50% lower power consumption, on-die-termination, and additive latency. With supply and I/O voltage at 1.8V instead of 2.5V, DDR2 memory consumes only half of the power of DDR. In addition to lower signal swing, with both on-die termination (ODT) and output calibration (OCD), signal integrity has been improved to support data rate up to 800MHz. Currently Xilinx Virtex 2 Pro FPGA officially support 400MHz DDR memory interface. The data rate is largely limited by the FPGA fabric speed rather than the external signal integrity. Another new feature of DDR2 is the additive latency. As shown in Figure 2, with additive latency feature, CAS command can be moved to the cycle right after RAS, therefore fully pipeline the data access when sequentially read/write through the memory banks.

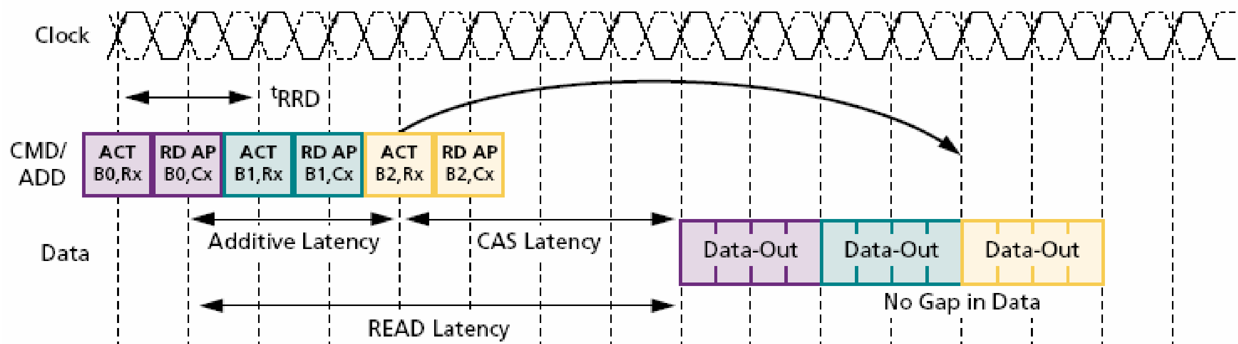


Figure 2: DDR2 additive latency

3.2 Compute Node

Each compute node consists of four basic compute elements and one control element. The control element is similar to compute elements, but with additional global interconnect interfaces and controls signals to the secondary system components. As shown in Figure 3, the connectivity on the compute node can be classified into two classes: local 2D mesh and global 4-ary tree.

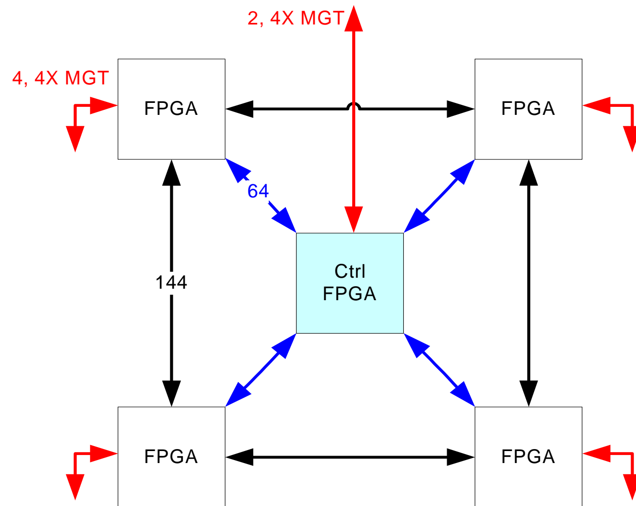


Figure 3: Compute node connectivity

The local mesh connects the four compute FPGAs on a two-by-two 2D grid. Each link between the adjacent FPGA on the grid has 144 physical single-ended circuits designed to achieve up to 200MHz DDR using LVCMOS signal standard, which aggregates to 57.6 Gbps data bandwidth per link. These direct FPGA-to-FPGA mesh connections provide a high-bandwidth low latency connection for FPGAs on the same compute node (e.g. on the same PCB board), such that up to four compute FPGAs can be aggregated together to form a virtual FPGA four times the capacity. With each FPGA capable of emulating 2 million ASIC gates, each compute node has an equivalent ASIC capacity of over 8 million gates (>40 million transistors).

The local mesh is design to maximize on-board inter-FPGA connection bandwidth with minimal latency; therefore a large number of parallel I/O connections are used to directly connect FPGAs. However, a more complex scheme has to be used when constructing global communication networks, which will be explained in detail in the next section. Here, we only discuss about the global 4-ary tree connection on the compute node. From the control FPGA, there are four down links, one for each of the computing FPGA. Each downlink has 64 physical single-ended circuits running up to 200MHz DDR using LVCOMS signal standard, which aggregates to 25.6 Gbps bandwidth.

Unlike the down links, the two uplinks from the control FPGA use the Multi-Gigabit Transceivers (MGTs) on the FPGA. Each of the uplink is a 4X MGT channel bonded to form a 10Gbps full duplex (20Gbps total) connection. The individual MGT channels are configured to run at 3.125Gbps with 8B/10B encoding. The two uplinks are connected to two external Infiniband 4X connectors on the edge of the PCB board. In conjunction with the four downlinks, the control FPGA serves as a tree-node on the 4-ary tree.

Each computing FPGA also has four 4X MGT (10Gbps full duplex) downlinks, which makes the computing FPGA also a tree-node. Therefore each compute node hosts two levels of the 4-ary tree. The bandwidth ratio between up/downlinks is 1:4, except for the uplink of the compute node (same as the uplink of the control FPGA), where two uplinks can be used, which increase the bandwidth ratio to 1:2. When a compute node is used as a leaf node, all the downlinks on the compute FPGA becomes the direct I/O channels to external high-bandwidth real-time devices, such as radio frontends.

Since each FPGA has 20 MGTs, which exceeds the number needed for the global interconnects, the remaining MGTs are configured to connect to serial ATA hard drives. Each

control FPGA can have up to 10 SATA disks. Both type I and II of SATA standards are supported on the hardware.

In addition to the high bandwidth interconnections, each compute node also has one 10/100 Base-T Ethernet connection to the control FPGA through an external Intel PHY controller chip.

3.3 Global Communication Networks

As shown in Figure 4, BEE2 system provides three types of global communication networks between compute nodes: a 4-ary low latency global communication tree, a high-bandwidth non-blocking Infiniband switch, and Ethernet switch.

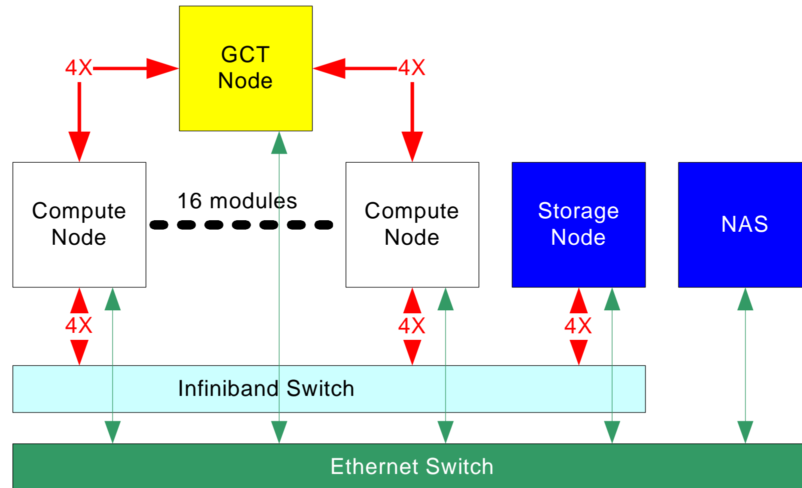


Figure 4: Global communication network overview

The 100 Base-T Ethernet connections provide basic administration services, such as interactive login, system monitoring, and relatively low bandwidth NFS from network attached storage (NAS) servers. The high-bandwidth Infiniband network provides 10 Gbps duplex throughput to each of the computer nodes, forming a non-blocking packet-switched crossbar using commercial Infiniband switch from Mellanox [12]. Each Infiniband switch can connect up to 144 computer nodes, and aggregating to 2.88 Tbps bi-section bandwidth. The Infiniband network is mainly used for high-throughput point-to-point communication services, such as MPI. In addition, the Infiniband network can direct connect to storage area networks (SAN) for high-bandwidth file services. When a compute node is configured as a storage node by connection up to 10 SATA hard disks to the control FPGA, the storage node can be directly attached to the Infiniband network and serve as shared storage for other compute nodes in the system.

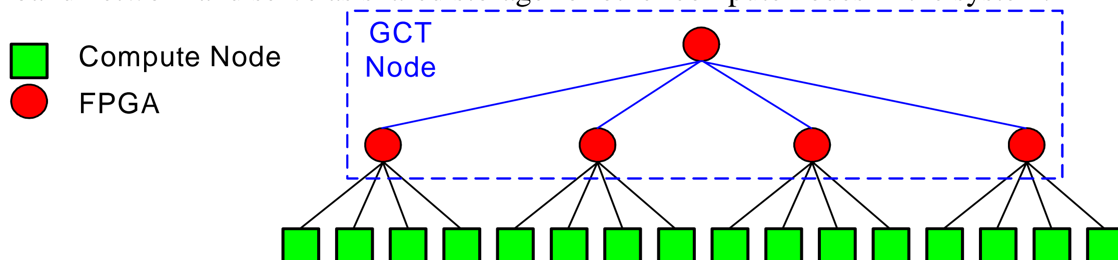


Figure 5: 4-ary communication tree network

To further reduce the node-to-node communication latency, the 4-ary tree global communication network is constructed directly using BEE2 compute nodes as global

communication tree nodes. As shown in Figure 5, when used as a global communication tree (GCT) node, each GCT node can connect to 16 compute nodes as leaves, thus forming a two-level 4-ary tree topology. Larger network can be formed with additional GCT nodes. Each parent level of the tree reduces the aggregate bandwidth to ¼ of the children level. However the node-to-node latency is greatly reduced to logarithmic of the number of nodes in the system. Furthermore, unlike the packet switched Infiniband network, the communication tree provides direct links between nodes without packetization; hence reducing the controller overhead. For a 256 node system (1024 compute FPGAs), the GCT node-to-node worse case latency of 880ns is less than half of the Infiniband latency. The tree topology and the low latency nature of the GCT network provides the necessary components to construct a distributed shared memory system, where a single global address space is partitioned to exclusive local memory region and global shared memory region. The local memory is the physical DRAM attached to each FPGA on the leaf compute nodes. Only the local FPGA can access its own local memory. DRAM attached to the FPGAs on the GCT nodes is in the global shared memory space and any FPGA anywhere in the system can access it contents.

4 BLAST Performance on BEE2

4.1 BLAST implementation on BEE2

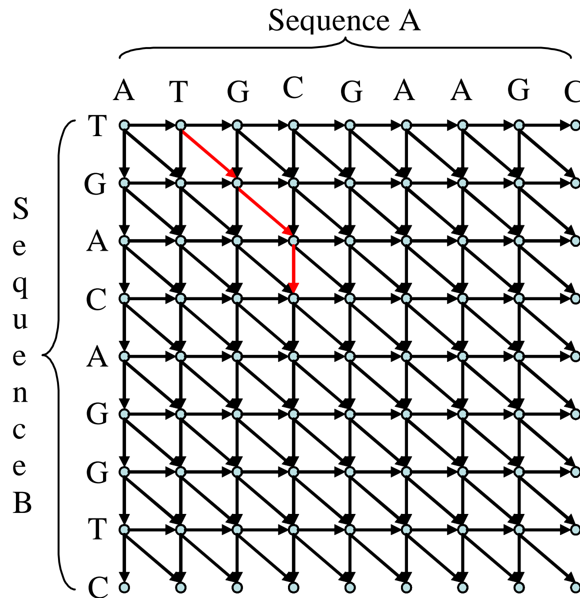


Figure 6: Sequence alignment edit graph

The basic construction for any pair wise sequence alignment is a 2D dynamic programming graph. Vertexes of the graph are located on a 2D grid where each axis represents one of the sequences being compared. A given vertex at location (x,y) represents a alignment between the x_{th} symbol of one sequence to the y_{th} symbol of the other sequence. There are always 3 edges leaving any vertex (except the boundaries), and traversing the edges represents extending a sequence alignment. Any path taken in this 2D dynamic programming graph represents a possible alignment between the two input sequences. Figure 6 is an illustration of a dynamic programming graph. Because the size of the graph grows quadratically with input sequence

length, an exhaustive search algorithm becomes computationally infeasible for large input sizes. The BLAST algorithm was created to reduce the computational requirements for large sequence alignment problems by using a heuristics to restrict the search space while still maintaining an acceptable level of sensitivity. BLAST accomplishes this by limiting search space to only the neighborhood of the dynamic programming graph that contains k consecutive exact matches between the two input sequences. Through this reduction in search space BLAST is able to runs significantly faster than exhaustive searches. However for really large problems even the time required to run BLAST becomes unacceptably long on conventional software based solutions. By mapping the BLAST algorithm on to a FPGA based reconfigurable platform, we hope to show significant speed up over conventional cluster based, multiprocessor implementations.

4.1.1 Algorithmic description of BLAST implementation on BEE2.

4.1.1.1 Step 1 – index k-mers

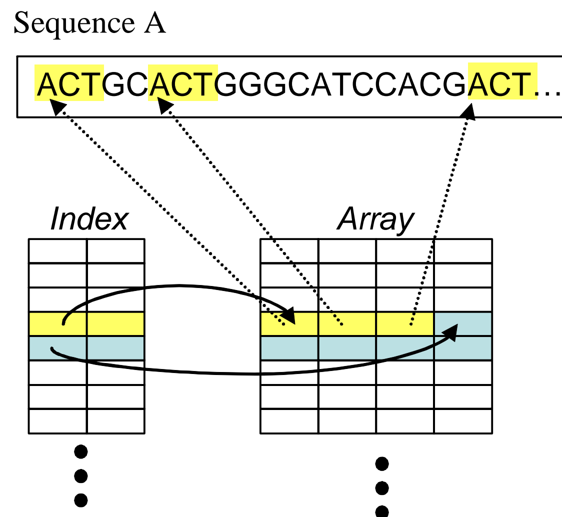


Figure 7: K-mer indexing

The first step in our implementation of the BLAST algorithm is constructing an index of all K-mers for one of the input sequences, as shown in Figure 7. The index is a 1D array in which the address of each entry corresponds to a unique K-mer. The content of the index points to a packed array containing the locations that a particular K-mer appears in the input sequence. Table 3 shows the pseudo code for the generation of the indexed array. There are two reasons for choosing this data structure. The first and most important one is memory usage. Without packing the array we would have to pick a very conservative upper bound on the number of times each k-mer appears in the input sequence, and much of the space would be wasted. The second reason is memory access efficiency, basically we would like to minimize to number of memory access needed to access the array. Although an unpacked array would require only one access, it is way too inefficient in terms of space usage. The next best thing is to add an index, this is similar to a sparse matrix representation, and since we access one row at a time we only incur one extra index access every time.

Choosing this packed array format also means that we need to make an extra preprocessing pass of the input sequence to create a histogram of k-mer distribution, only this way will we know how much memory space to allocate to each k-mer entry

Table 3: Pseudo-code for step 1

```
Phase 1 (reset index_a)
for i = index_base to index_max
      mem_addr(i) = 0;
Phase 2 (construct histagram)
for k-mer = all k-mers in sequence_a
      index(2*k-mer) += 1;
      index(2*k-mer+1) += 1;
Phase 3 (transform histagram to pointer)
      index(0) = array_base;
      Index(1) = array_base;
for i = index_base to index_max step 2
      index(i) = index(i) + index(i-1);
      index(i+1) = index(i);
Phase 4 (construct packed array)
for i = 1 to length(sequence_a)-k
      k-mer = sequence_a(i to i+k);
      mem_addr(index(2*k-mer+1)) = I;
      index(2*k-mer+1) += 1;
```

4.1.1.2 Step 2 – index hits

The second step in our implementation is similar to the first step. This step take for input the indexed array from step 1 and the second input sequence. The output of this step is also an index and packed array. The resulting array for step 2 contains the locations of the hits (or complete K-mer match) from the two input sequences. The key used to index and sort the array is the diagonal of 2D dynamic programming matrix on which a hit is located, as shown in Figure 8. Sorting by diagonal makes construction of the array less efficient since a hit of a particular k-mer in the second sequence appear in different diagonals. However the diagonal sorting enables parallel hit expansion in the last stage.

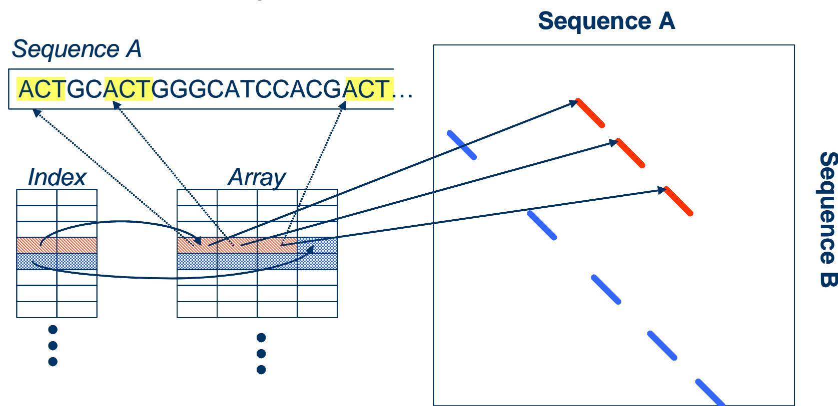


Figure 8: Hits indexing

Like step 1 using a packed array requires a preprocess pass to construct a histogram. Also another feature of this second step is that consecutive hits are recorded only once. This means that if there is an exact sub-sequence match longer than k, only the first k-mer hit will be recorded into the array. This is done by looking at two rows in the dynamic programming matrix at a time and only recording a hit at location (x,y) if (x-1,y-1) is not a hit.

Table 4: Pseudo-code for step 2

```
Phase 1 (reset index_b)
for i = index_base to index_max
        mem_addr(i) = 0;
Phase 2 (construct histagram)
for i = 1 to length(sequence_a)-k
        k-mer = sequence_b(i to i+k);
        for j = all locations in array_a(k-mer)
                index(2*(i-j)) += 1;
                index(2*(i-j) + 1) += 1;
Phase 3 (transform histagram to pointer)
index(0) = array_base;
index(1) = array_base;
for i = index_base to index_max step 2
        index(i) = index(i) + index(i-1);
        index(i+1) = index(i);
Phase 4 (construct packed array)
for i = 1 to length(sequence_a)-k
        k-mer = sequence_b(i to i+k);
        for j = all locations in array_a(k-mer)
                mem_addr(index(2*(i-j) + 1)) = i;
                mem_addr(index(2*(i-j) + 1)+1) = j
                index(2*(i-j) + 1) += 1;
```

4.1.1.3 Step 3 – expand hits

The last step in our BLAST implementation is the actual expansion of hits. In this step the input is indexed array from the previous step, and the two input sequences. In this step the hits from step 2 are processed and expanded. Results with e-values above a certain threshold are returned. Notice that since BLAST only does hit expansion along diagonals, each index entry representing a separate diagonal can be processed in parallel with no dependency from any of the other diagonals.

Table 5: Pseudo-code for step 3

```
for i = index_b_bass to index_b_max step 2
        for j = index_b(i) to index_b(i+1) step 2
                loc_x = array_b(j);
                loc_y = array_b(j+1);
                sub_seq_a = seq_a(loc_x);
                sub_seq_b = seq_b(loc_y);
                score_max = new_max(sub_seq_a, sub_seq_b, score_max, score_cur);
                score_cur = new_score(sub_seq_a, sub_seq_b, score_cur);
                while (score_max - score_cur) < thresh_hold
                        <extend sub_seq_a,b upstream>;
                        <recalculate score_max, score_cur>;
                        <extend sub_seq_a,b downstream>;
                        <recalculate score_max, score_cur>;
```

4.1.2 Analytical model for memory usage and accesses

In this section we present an analytical model for the memory usage and performance of our BLAST implementation. This model will be combined with simulation results to produce performance estimates for benchmark comparisons with other BLAST implementations.

Seq_a = Symbol length of input sequence A

Seq_b = Symbol length of input sequence B

S = bits per symbol (2 for DNA, 5 for protein)

K = k-mer length in Symbols

w = wordsize in Bytes (8)

Equation 1: memory capacity requirements	**Equation 2: number of memory accesses**
$Index_a = 2w \cdot 2^{S \cdot K}$ $Array_a = w \cdot Seq_a$ $Index_b = 2w \cdot (Seq_a + Seq_b)$ $Array_b = 2w \cdot \frac{Seq_a \cdot Seq_b}{2^{S \cdot K}}$	$Step1_{\Phi 1} = \frac{2w}{32} \cdot 2^{S \cdot K}$ $Step1_{\Phi 2} = \left(\frac{s}{8 \cdot 32} + 2\right) seq_a$ $Step1_{\Phi 3} = 2\frac{2w}{32} 2^{S \cdot K}$ $Step1_{\Phi 4} = \left(\frac{s}{8 \cdot 32} + 3\right) seq_a$ $Step2_{\Phi 1} = \frac{2w}{32}(Seq_a + Seq_b)$ $Step2_{\Phi 2} = \left(\frac{s}{8 \cdot 32} + 1 + 3\frac{Seq_a}{2^{S \cdot K}}\right) \cdot Seq_b$ $Step2_{\Phi 3} = 2\frac{2w}{32}(Seq_a + Seq_b)$ $Step2_{\Phi 4} = \left(\frac{s}{8 \cdot 32} + 1 + 4\frac{Seq_a}{2^{S \cdot K}}\right) \cdot Seq_b$ $Step3 = (Seq_a + Seq_b) + 3\frac{seq_a \cdot seq_b}{2^{S \cdot K}}$

From the analytical model we can see that the memory usage for *index_a* is actually exponentially dependent on the k-mer length *k* while *array_b* in inverse exponentially dependent on *k*. This means that depending on input parameters *index_a* could dominate memory usage. However in practical cases *k* will not grow too large since a large *k* will degrade the sensitivity of the search. In practical cases it is usually the quadratic term $Seq_a \cdot Seq_b$ that dominates this computation. However in cases of extreme imbalance between the input sizes, the $Seq_a + Seq_b$ term will dominate. This happens when the expected number of hits per diagonal in the dynamic programming matrix drops below one. In that case many of the index entry in step 2 will be empty and wasting space, and accessing them in phase 1 and 3 will be wasting memory accesses. Our implementation operates inefficiently under this regime, and a different algorithmic approach should be considered. However we want to consider cases of large BLAST searches that even large cluster based computers have difficulty solving quickly. In those cases the query is typically large and operates with good efficiency using our implementation.

4.1.3 Circuit implementation

As mentioned before, implementing the algorithm as hardware circuits allow us to exploit parallelism that would otherwise be sequentialized in a software based processor. For the BLAST algorithm the amount of actual computation performed is relatively small. Most of the work in our implementation of the algorithm revolves around constructing and reading the two

indexed arrays, and the input sequences. This means that the most crucial resource is the memory ports. Thus most of the algorithm implementation effort was spent on ways to efficiently use the memory.

The circuit for steps1 and 2 share certain features with each other. Phase 1, in both steps involves zeroing out the index. The circuit for performing this task is simply an address counter with boundary check and is trivial to implement. Phase 3 of transforming histogram to packed array pointer for both steps is also identical. The circuit implementation for this phase involves on address counter to stream in the index, an adder to generate the new index value and a register to hold the accumulated intermediate result, and finally another address counter to write the index value back.

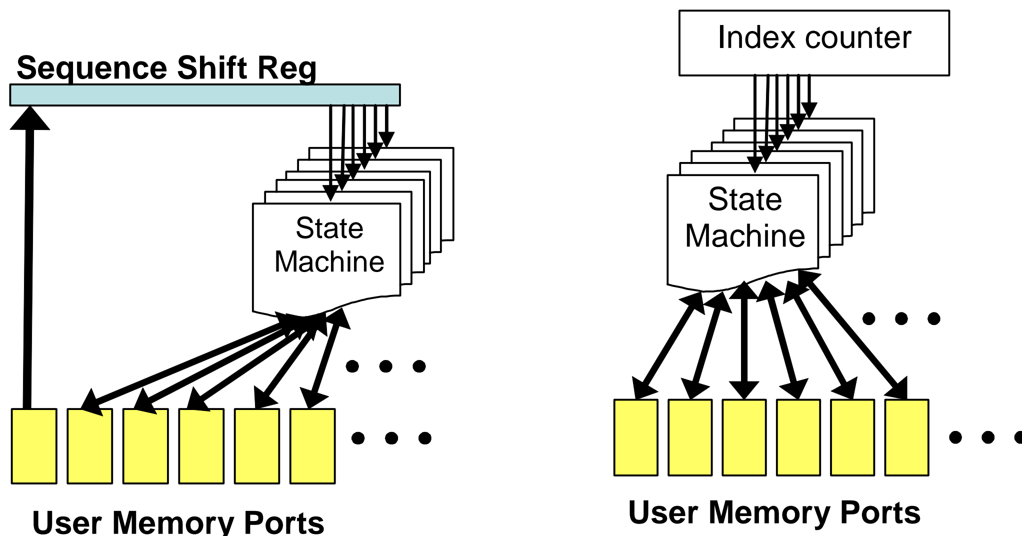


Figure 9: Circuit implementation diagram

The circuit for phases 2 and 4 are different for steps 1 and 2. However some parts of these circuits can still be shared. Both steps involve streaming input of a sequence and using the k-mer at the head of that stream to build a histogram index, and then a packed array. For the Hardware implementation, they both use common circuit that pulls an input sequence into a shift register. K-mers from the head of the shift register are issued into parallel identical state-machines. Depending on the particular Step and phase of the computation, a different set of state-machines is used. Each state-machine in an active set has exclusive access to a memory port.

The state-machines for step 1 are relatively simple, for the histogram in phase 2 it simply issues a read_increment_write of the k-mer address in the index to its memory port. For phase 4 it issues a read_increment_write in addition to a write of the k-mer position into the array.

The state-machines for step 2 are more complex. They used to index k-mer to index into the result array from step 1, loop over the returned locations, the returned locations and the k-mer location are combined to form the hit locations and the diagonal index, and these are then used to update the index and array in the same fashion as step 1.

The atomic read_increment_write function provided by the memory controller is a key feature in the design. This operation allows histogram and array update to be issued in parallel without introducing inconsistencies into the index. Without this operation some kind of index collision detection and arbitration mechanism would have to be used to guarantee index

coherency. This would not only introduce more complexity to the design be also introduce extra dependencies and memory accesses.

The circuit for Step 3 is another set of parallel state-machines each with its own memory port. Each state-machine expands all the hits with in one diagonal index entry. As mentioned before since BLAST only expands on diagonals, there is no interaction between any of the state-machines thus no synchronization is required. And since each memory access is 256bits wide, almost all random hits will be filtered out after the first sequence compare.

4.1.4 Database partitioning

Steps 1 and 2 of our algorithm generates large amount of data. In fact the memory usage required by this implementation is proportional to the product of the lengths of the two input sequences. This means that even with 4GB of DRAM per FPGA, the largest problem size that can fit is approximately 14M by 14M amino acids or 26M by 26M nucleotides symbol search. This means that large searches against databases need to be partitioned to smaller problems that require less memory to run. These smaller problems could then run in parallel if enough hardware resources were available or run sequentially if not enough resources are available. The results from each of these runs would then be combined to generate the final result.

For our algorithm, we took the same approach to dividing the problem as MPI BLAST [5]. We chop the database into smaller pieces, and then assign each piece to a different FPGA. We chose this because the database is typically so large that if left intact, the problem will not fit in memory no matter how small the query is. The other advantage of dividing the database is that because the database is composed of non-continuous sequences, when dividing it at sequence boundaries we do not have to worry about result matching sequences spanning across different divisions.

4.2 Memory performance tuning

Among all potential performance bottlenecks, external DRAM memory bandwidth and latency is the one of the most commonly encountered and most severe limiting factor for majority of applications. Unlike other performance bottlenecks, such as global communication bandwidth and latency, in general application algorithm has little effect on reducing the memory subsystem requirement. Therefore external memory system performance is crucial for achieving high performance on the BEE2 system.

As explained before, each basic computing element has four channels of DDR2 DRAM DIMM modules attached, which makes each FPGA a shared memory node. Just like on conventional shared memory multiprocessor computers, or Symmetrical Multi-Processing (SMP), physically sharing multiple memory channels requires careful design of memory arbitration and interleaving. However, the BEE2 memory system also differ from SMP in two aspects: 1) FPGA does not have cache, therefore the memory system does not suffer from cache coherence and consistency problem; 2) with FPGA as the memory controller, application dependent memory access optimizations are available to maximize the memory system performance.

The basic design goal of the memory subsystem is to present a simple SRAM-like streaming memory access interface, which hides all the complexity of DRAM access and interleaving issues from the end user. As depicted in Figure 10, on the user side (left of the diagram) the memory subsystem offers up to 16 virtual ports per FPGA. Each user memory request is issued through the crossbar to one of the memory channels. Memory requests are queued on the user

side virtual port queues, and only issued if the particular memory channel is available and the requested memory bank is idle. By making the virtual queue size to be exactly one memory access, the memory subsystem can simultaneously offer minimal latency at maximum memory bandwidth to the end user.

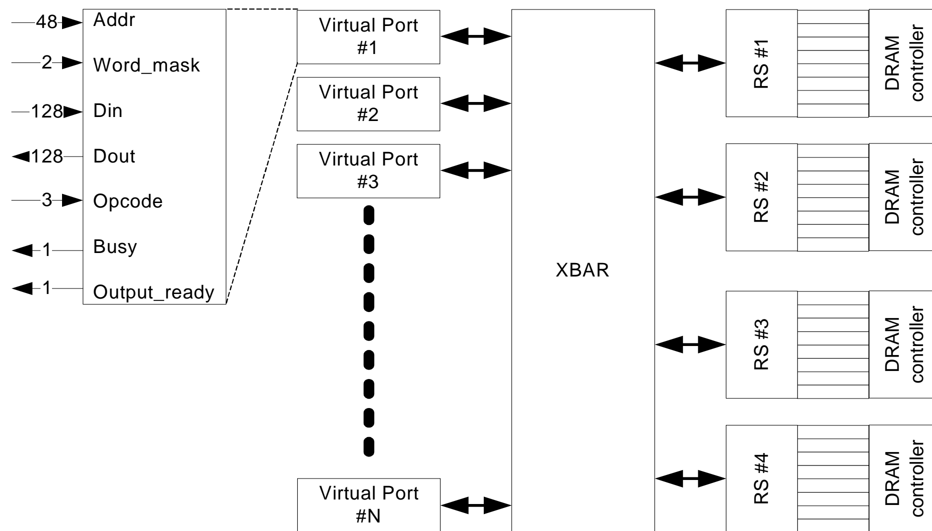


Figure 10: Memory subsystem overview for a single compute element

DDR2 Memory modules are design to provide maximum bandwidth when adjacent memory access does not use the same memory bank, such that the bank precharge time can be hidden from the user. However, this requires the right combination of bank interleaving scheme and memory request issue scheme to achieve the best bandwidth utilization with minimal penalty on access latency. Since most memory access pattern exhibits more volatility on the lower address bits, the BEE2 physical memory address uses the lower bits for interleaving memory channels and banks. As shown in Figure 11, using word address scheme, the least significant two bits are used to identify the individual words in each memory access burst block of 4 consecutive 64-bit words. The next two bits identify which memory channels to use, and the following three bits chooses which of the eight memory banks on the channel. The most significant three bits are used to distinguish different memory access types, such as local memory, shared global memory, etc. Overall with 48 bit resolution, BEE2 system can address up to 2 peta-bytes of data on a single shared address space.

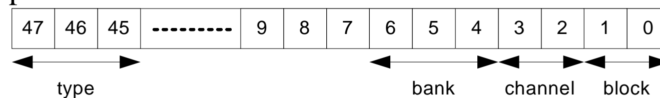


Figure 11: Physical memory address fields

Since the BEE2 system does not have hardware controlled cache, the memory subsystem is free of all cache related coherency and consistency problem and performance degradation, which is a common performance bottleneck found on conventional shared memory computers server. However, concurrent memory access consistency and coherence problem still exist. In order to maintain the correct order of concurrent memory access, the strictly sequential consistency memory access model is used. Each virtual port memory requests are executed in order; while the memory access among all virtual ports on the same cycle can be interleaved in any order. The BEE2 memory controller uses the global round robin scheme to choose the order to serve the virtual port, but chooses which one to server using a greedy algorithm that maximizes the

memory bank utilization by skipping the memory request that conflict with on going memory operations on the same bank. However, the greedy algorithm alone does not guarantee fairness among the virtual ports. In the pessimistic case, some virtual ports can have unbounded access latency. Therefore, to limit the maximum memory access latency, a maximum skip bound is used to ensure that if a particular memory access on the port is skipped consecutively over the limit (typically 3 times), the request will always be server next before any other requests.

To provide coherent memory access, special atomic memory operations are implemented by the memory controller to guarantee the completeness of the operation before other memory request to the same location can be served. One of the commonly used atomic memory operation is “test & set”, which is necessary for locking a portion of the memory access with a flag. It first reads the flag memory location, if the flag value is zero, it writes back the value 1 to memory, and return true; if the flag is non-zero, it returns false. Basically the “test & set” operation links a read and a write to a single memory location.

Other atomic memory operations can be added per application requirements at compilation time. For example, for applications that builds large histogram, such as in BLAST, an “atomic increment & update” operation can greatly improve the histogram performance. Traditionally, to concurrently increment a large number of histogram counters in memory, the program has to first lock the particular counter memory location with a flag by doing a “test & set” operation, if successful, read the counter value, increment then write it back. Overall, even if no memory contention, it requires 2 read and 2 writes. If memory contention occurs, there can be even more reads for multiple failed “test & set” operations. On the other hand, by moving the increment operation into the memory controller, the counter value can be incremented then write back atomically. Therefore no “test & set” operation needed to lock the counter memory access, and exactly 1 read and 1 write is needed to update the histogram counter value. Furthermore, since no memory locking needed, memory contention caused performance loss is minimized.

In addition to specialized atomic memory operations, the BEE2 memory controller also takes advantage of the DRAM access feature to reduce the latency of atomic access even further. Although an atomic operation, such as “test & set”, logically take 1 read and 1 write, the atomic operation can be executed with only 25% increase in latency than a normal read/write access. A normal read/write takes 3 cycles for RAS, 3 cycles for CAS, and 2 cycles for data transfer. When read then write to the same location, the RAS can be omitted and the second CAS for write can be issued right before the data access, therefore only the 2 cycle write data transfer is added to the normal read latency of 8 cycles to achieve the atomic memory operation. The actual logic operation, compare to zero in the case of “test & set” or addition in the case “atomic increment & update” can be pipelined and finished between the read to write data access gap.

The memory controller provides several compile-time parameters to fine tune the memory subsystem for each application or a range of similar applications. The number of virtual ports can variable from 4 to 16, and each port can have data width of 1, 2, or 4 words of 64-bit. A cycle accurate memory simulator was built in Matlab, which can provide accurate memory performance simulation using application memory traces or direct interact with the application. To benchmark the memory subsystem performance in generation, two memory access patterns are used, one is to have all virtual ports sequentially access the memory with a constant stride that is integer multiples of the minimum block size of 4 words, the other pattern is to have all virtual ports randomly access all memory locations.

As shown in Figure 12, for constant stride memory access, if the stride is an odd multiple of the burst length (4 words), then the memory controller can achieve 100% of the total aggregate

memory throughput of 12.8 GBps. This is achieved when all 32 memory banks (8 banks for each of the 4 memory channels) can be perfectly interleaved among the virtual ports. When the stride distance is an even multiple of the burst length, the memory access periodically try to issue to the same memory bank before the previous memory access can be finished, therefore leading to memory throughput degradations. The most pessimistic case is when the stride is 128, which is 32 times the burst length, matching the number of available interleaving memory banks, so every memory access lands on the same bank.

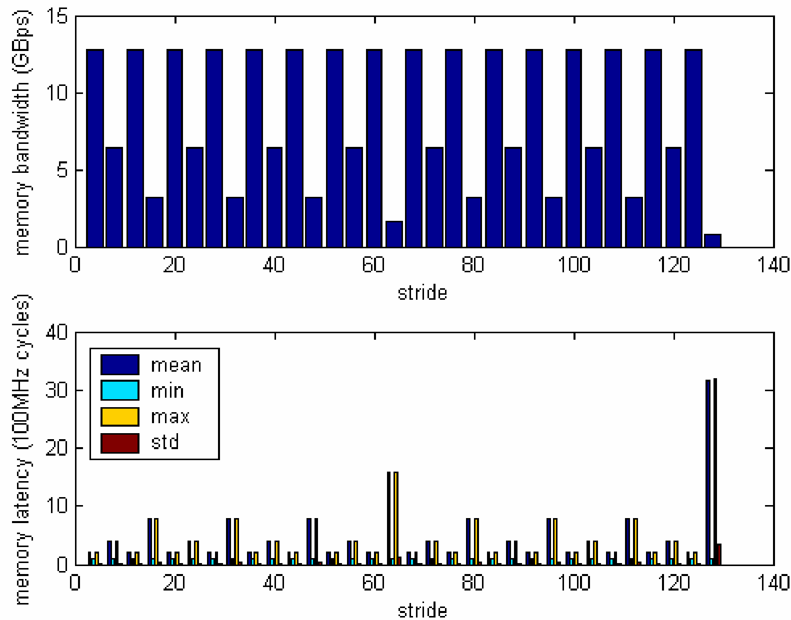


Figure 12: Constant stride memory access performance

Unlike the constant stride memory access, which has a deterministic latency, random memory access latency exhibits the Poisson process distribution, as shown in Figure 13. Majority of the memory access are serviced right away, but an exponentially decreasing number of memory access takes longer time. As explained before, in order to ensure memory access fairness, a maximum skip bound of 3 failed services is enforced by the memory controller. In this benchmark case, the maximum latency is reduced to 30 cycles instead of the unbounded case of over 80 cycles.

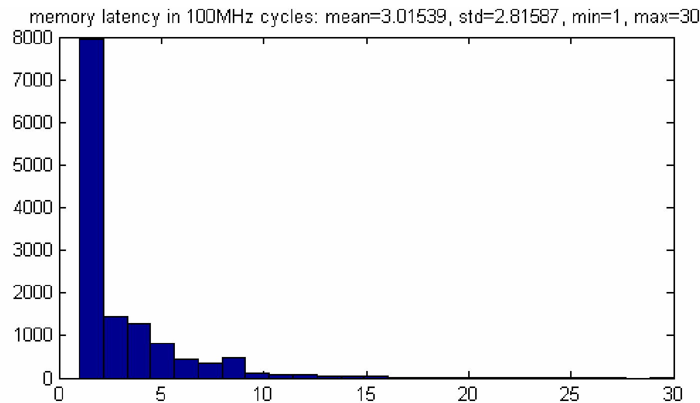


Figure 13: Random memory access latency histogram

Since the memory controller issues the memory request using a greedy algorithm to maximize the concurrent DRAM bank utilization, the memory throughput increases the more virtual ports available (e.g., more concurrent memory accesses to choose from). As shown in Figure 14, the memory throughput saturates for more than 16 virtual ports. However, the implementation cost of the memory controller on FPGA grows roughly as the square of the number of virtual ports. Therefore most efficient utilization of the memory throughput in the random access case is using 16 or less virtual port. In addition, the effect of more virtual port also increases the maximum latency quadraticly, so to achieve the best bandwidth-latency product, 8 virtual ports should be used.

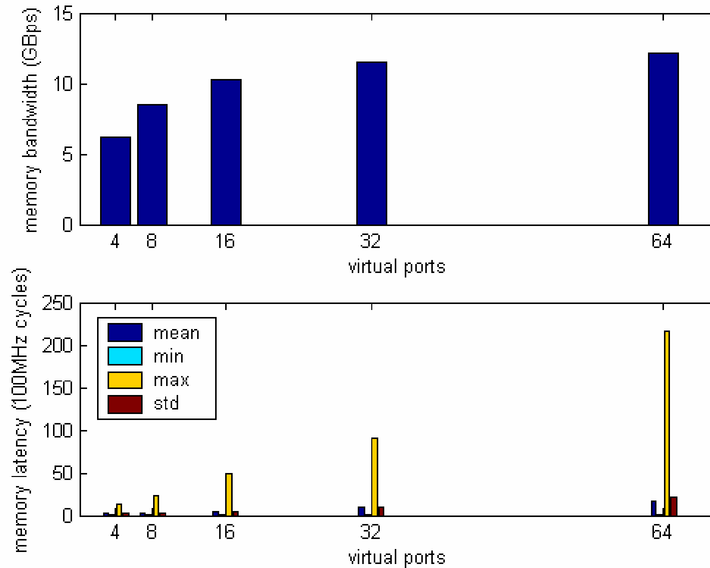


Figure 14: Random memory access performance

When benchmarking atomic memory access over random addresses, the throughput is generally more than 10% higher than just random reads or writes for virtual ports less than 16. This is due to the fact that atomic access put the data access of a read and a write back to back, which fully utilize the memory bandwidth. So when the number of concurrent memory access is relatively low, atomic memory access utilize the bandwidth more efficiently. However, when more concurrent memory requests are present (e.g., for over 16 virtual ports), the normal random reads or writes can utilize the memory bandwidth as well as the atomic accesses. Nevertheless, the higher resource cost of implementing more virtual ports on the memory controller makes it impractical.

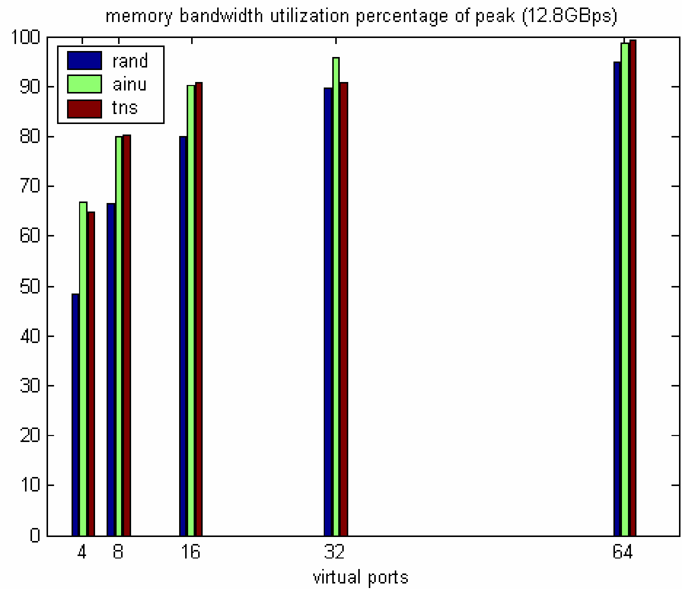


Figure 15: Atomic memory access bandwidth utilization

Another tunable parameter of the memory controller is the virtual port data width. At full DRAM bandwidth, four virtual ports each with 256 bit data width can saturate the bandwidth. If the data width is reduced to either 128 or 64 bits, it takes multiple cycles to transfer the whole 256 bits of data for each DRAM access burst of 4 words, which limits the overall memory throughput. This effect is most pronounced for fewer virtual ports, as shown in Figure 16. Especially for constant stride memory access, in the case of 8 virtual ports, if the port data width is 64-bit, the virtual port bandwidth can only saturate half the DRAM bandwidth, therefore the through put is reduced to 50% of the peak value.

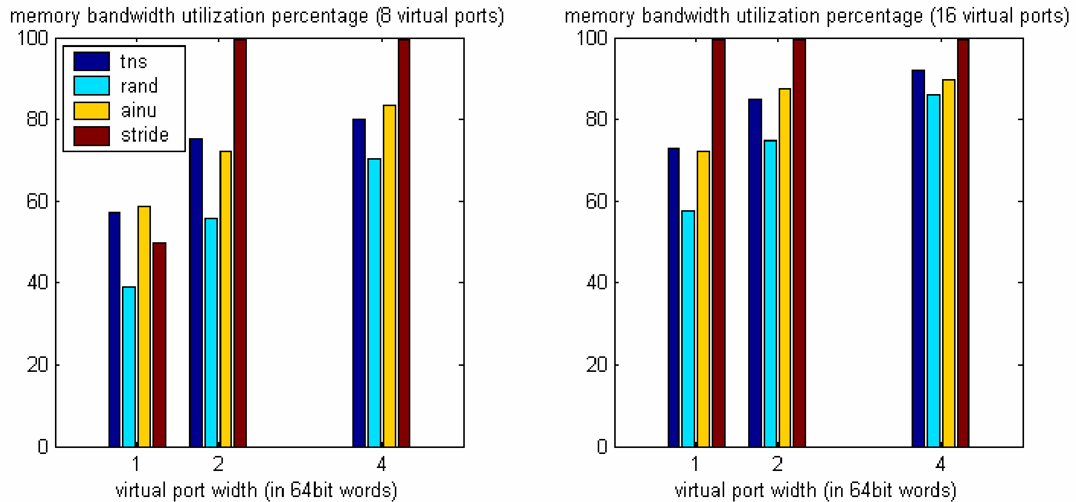


Figure 16: virtual port width effect on memory throughput

4.3 Overall performance

4.3.1 Simulation environment and results

A cycle-based simulator in Matlab was used to test our BLAST implementation. The simulated application is broken down into concurrent modules each describing a different part of the simulated hardware circuit. During each simulated cycle a module is first instructed to grab all its input from the output of the previous cycle, then an `execute()` function is called to updated all the output and internal state of the module. The actual style of coding for the circuits resembled HDL (Hardware Description Languages) codes with the added bonus of Matlab arrays, objects, and debugging interface. The simulation accurately reflects interaction of the BLAST circuit with the DRAM controller circuit and the quad channel DDR2 RAM, all of which are modeled as modules within the simulation.

For all results generated by the simulator, we assumed that the frequency of operation for our circuit is 100 MHz, and that the circuit required for the computation will fit within the FPGA chip. We feel that these are very reasonable assumptions as the FPGAs used in BEE2 have the capacity to implement millions of ASIC gates, and the operating frequency for a typical design in that family is faster than 100 MHz.

Because the cycle-based simulator is relatively slow, large problem sizes are impossible to simulate. On the other hand, even clusters of modern processors running actual BLAST would have trouble at the problem sizes we are interested in. So there is no way that a cycle based simulator can handle that large of an input size. In order to obtain estimated performance for our targeted problem sizes, we benchmarked each section of the program individually. We will then use the order of growth derived from the analytical model to extrapolate the performance our implementation for large problem sizes.

Table 6: Memory bandwidth efficiency

Memory access efficiency				
	Phase 1	Phase 2	Phase 3	Phase 4
Step 1	100.00%	88.83%	50.00%	72.15%
Step 2	100.00%	41.49%	50.00%	29.63%
Step 3	50.59%			

Memory efficiency in Table 6 gives an idea of how well the algorithm is implemented. With quad channel DRAM, the best-case memory performance is 4 memory accesses per cycle. Memory efficiency is obtained by the equation $eff = \frac{TotalMemAccess}{4 \cdot TotalCycle}$. An efficiency of 100% provides an upper bound on the performance of this algorithm. The synthetic memory controller benchmark showed that even random access to the user memory ports could nearly saturate the total memory bandwidth. We found that the sub-optimal efficiency in our implementation is caused by memory latency and dependencies in the state-machines. For example when accessing the content of the array from step 1, the state-machine cannot issue the read to the array until the read from the index has produced a result. This kind of dependency causes a certain fraction of user memory ports to sit idle and in turn reduces the number of operation for the DRAM controller to banking of off. As show in the table, the more complex state-machines of step 2 suffer more from memory dependencies.

4.3.2 Performance Comparisons

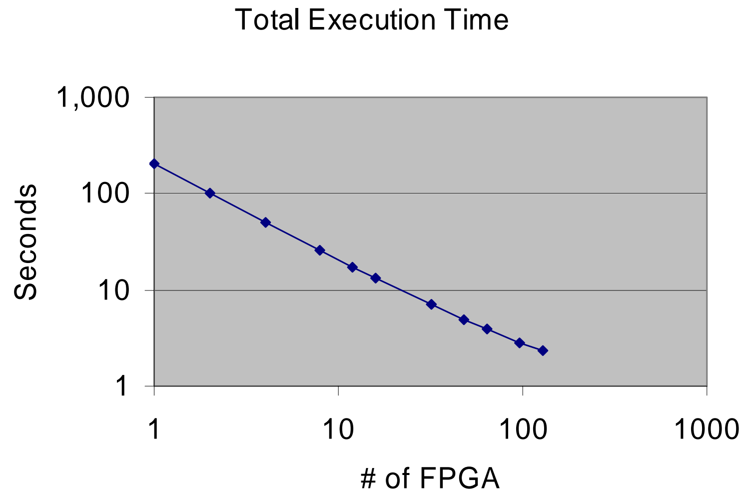


Figure 17: Excution time (300KB query, 1.2GB database)

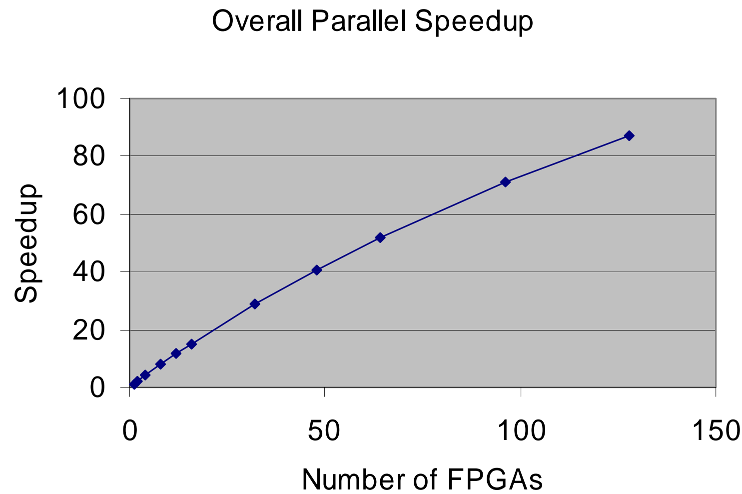


Figure 18: Parallel Speedup (300KB query, 1.2GB database)

Figure 17and Figure 18 shows the scaling of our implementation on BLAST with more resources. The input database was assumed to be fragmented into 120 evenly sized pieces. The sub-linear speedup results from the global communication, sorting, and merging time of the final BLAST output results, which can not be parallelized.

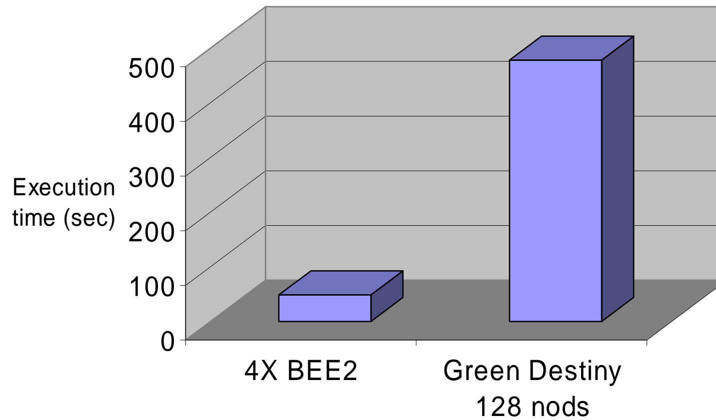


Figure 19: Performance comparison to mpiBLAST on Green Destiny cluster

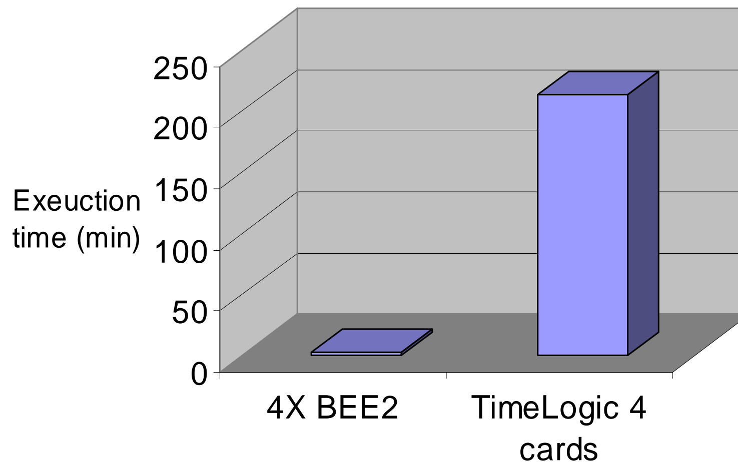


Figure 20: Performance comparison to TimeLogic Decypher

Figure 19 and Figure 20 shows comparison with published runtimes of other BLAST implementations. Figure 19 shows that one BEE2 board with 4 user FPGAs and 16GB of user memory is approximately 10 times faster than a 128 node 667Mhz transmeta TM5600 cluster with 82GB of memory while running a bench march of 300KB query against a 1.2GB database. Figure 20 shows a comparison between 4 TimeLogic Decypher FPGA based accelerator cards attached to an 8 processor SUN server and one BEE2 board with 4 FPGAs running a 2 billion symbol by 1.4 million symbol DNA search. The BEE2 board came out approximately 120 times faster than the TimeLogic implementation. Both TimeLogic and our implementation is FPGA based, we suspect that the difference in performance lies in memory bandwidth and latency. The TimeLogic accelerator cards sits on a PCI bus that severely limits the memory bandwidth and latency. Where as the FPGA chip in BEE2 have direct access 12.8 GBps of dedicated memory bandwidth, with latency only limited by the DRAM chip itself.

5 Conclusion

By combining a cycle-based simulator with an analytical model, we have projected the performance capability of the BEE2 platform to be 1 to 2 magnitude order higher than any of the present day computing systems when running the BLAST algorithm. When the cost of the system is factored in, with a per BEE2 module cost estimation of $8000, the price-performance ratio of the BEE2 system is over 40 times lower than the 128 node Green Destiny cluster, which cost $35000. When compared to the extremely expensive TimeLogic hardware accelerators, the BEE price-performance ratio is over 1000 times better.

The outstanding BLAST performance on the BEE2 system can be attributed to the following three reasons:

- **Higher computation density and efficiency**

  Microprocessors have a fixed operand size of either 32 or 64 bits. In the case of BLAST, where the algorithm only require 8 bit or less computations, this mismatch not only causes computation inefficiency on each operation, but also more instructions to be used for operations such as shifting and masking just to get the data in the right format. On the contrary, circuits implemented on a FPGA are configured specifically for the particular problem at hand, which maximize the computation density given a fix chip resource.
- **Massive spatial parallelism**

  By implementing circuits directly on the FPGA computing fabric allows for parallel tasks to be computed spatially to take advantage of all levels of parallelism in the algorithm; where as in a processor, parallelism is extracted inefficiently from sequential instructions. For the BLAST algorithm, this means that many parallel lightweight state-machines can enable more efficient utilization of the memory bandwidth. Hardwired circuits also remove the need for the loading of instruction streams, leaving the complete memory subsystem for data access only.
- **High-throughput low-latency memory subsystem**

  The main advantage of BEE2's design is the memory access advantage. In terms of bandwidth, each FPGA on the BEE2 board has a full 12.8GBps of memory bandwidth, which is more than twice as much as typical microprocessor based computer systems available today. In terms of latency, by customizing the memory controller to the specific application requirements, the BEE2 design can provide minimal latency without scarifying for memory bandwidth. This advantage is magnified in the BLAST algorithm, where the pointer chasing over a large random dataset renders any microprocessor's data cache virtually useless. In fact having caches hurts the processor because of all the latency involved in cache miss and refill. Finally by implementing the DRAM controller directly on the FPGA, customized memory operations such as read increment write can be used to streamline DRAM access.

6 Appendix A: FPGA computing technology

Ever since general purpose CMOS processors based on Von Neumann architecture (stored program, sequential execution, and time multiplexing hardware) became the underlying fabric of digital computing, the basic strategy to improve performance has been twofold: 1) use CMOS circuits that have the maximum performance on a single chip using the most advanced technology available, and 2) assemble N of these chips along with memory attempting to achieve an N times computational increase.

We feel that a new approach to computing using hardware reconfigurable chips for matching the computation in an application to the hardware resources and a high-level stream programming model can provide two orders of magnitude or more performance per unit cost and power consumption for a wide range of applications.

6.1 The obsolescence of Von Neumann era strategies

It has been over 50 years since Von Neumann first developed the processor architecture in widespread usage today. The experience and software tools that have been developed with this architecture have resulted in an enormous inertia that continues to hinder the consideration of any fundamentally different approach. Until recently the Von Neumann strategy had a number of advantages. First, it provided a means to easily exploit the cost, power and performance improvements made possible by the technology advancements characterized by Moore's law. Second, the software development (at least for the individual processors) was built on a wide experience base as well as a number of legacy programs that required the underlying Von Neumann computational model. The extension to parallel computers with Von Neumann nodes has been an active area for many years and many important applications have been parallelized.

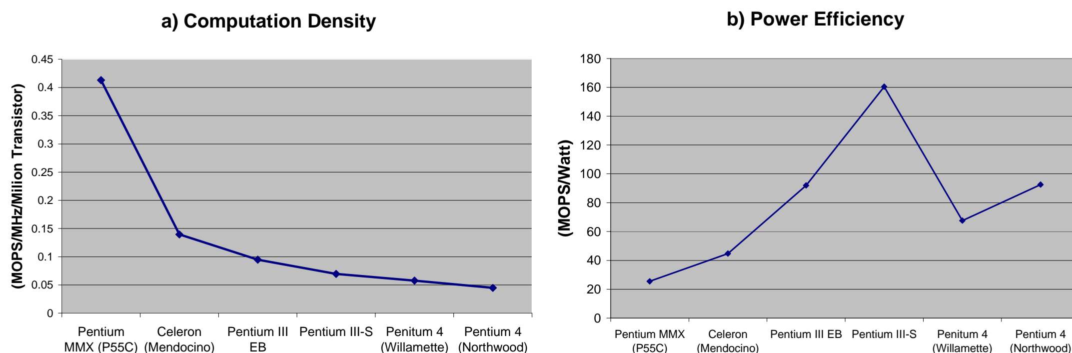


Figure 21: Computational density and power efficiency comparision

However, it is amazing that an approach that was developed over 50 years ago is still so dominant, in particular because the basic technology assumptions which were present when Von Neumann developed his approach are no longer valid. In particular, one of the strategies he proposed was to time multiplex the computational hardware because it was extremely expensive. While it was true 50 years ago, it clearly is no longer valid. In fact, the basic strategy of time multiplexing hardware has resulted in power and area inefficiencies which are causing the basic Von Neumann architecture to be unable to exploit the improvements provided by Moore's law scaling. Figure 21 shows the computation density and power efficiency vs. technology for Intel microprocessors. Figure 21a) shows the computation density in MOPS per MHz per million transistors to be decreasing with technology.

In the past decade, the basic strategy for improving the performance of individual Von Neumann processors has been to increase the clock rate, because it has been found that improving parallelism at the processor level through instruction set parallelism, multiple issues, and multiple threads, quickly reaches a point of diminishing returns. The increase of clock rate directly results in the explosion of the power density on the processor to the point that the heat can not be extracted fast enough to avoid component damage. Therefore, the performance obtained from each individual processor must be limited or very expensive cooling solutions must be deployed. This is a fundamental change in that power now limits performance, rather than circuit speed. The power efficiency graph for Intel microprocessors, shown in Figure 21b), stays relatively flat, despite the exponentially improving underlying silicon technology. This problem is even more pronounced on high end computing platforms which require the highest performance processors. Hence for future HEC platforms, the current solution is to either use a larger number of lower performance and thus lower power consumption processors (such as IBM blue gene), or build very expensive vapor cooling system (such as in NEC Earth Simulator, or Cray X1). Nevertheless, neither of the two solution lead to a cost efficient computing system.

Fast processors require high bandwidth data access to memory. The speed of low-cost large capacity memory (DRAM) has historically lagged behind the processor clock rate because memory technology has been optimized for capacity (chip area) rather than speed. Memory hierarchy has been widely used to reduce the memory data access time. A consequence of the heavy time multiplexing inherent in the Von Neumann model is that a large amount of intermediate states must be stored in fast on-chip cache to maintain performance, which further reduces the area available for computation. Today only a tiny portion of a microprocessor is performing the actual computation.

6.2 The FPGA solution

It is clear that a fundamentally different set of strategies are required if the inherent inefficiencies of the Von Neumann era approach is to be avoided. An approach is needed that exploits the increased density of logic and memory capacity as provided by technology scaling. This increase in logic and memory capacity should be used to provide spatial parallel computation at the chip level to achieve high levels of performance, rather than attempting to do it by time multiplexing a small amount of hardware. Since the application focus for HEC is fundamentally general purpose computing, the particular configuration of the parallel architecture needs to be fully reconfigurable. Furthermore, it is desired that the spatial parallelism be exploited at all levels in the application.

A strategy for the basic unit to be replicated in an HEC platform that meets the above requirements uses chips that have the ability to be configured to implement any arbitrary computational structure. The goal of mapping an algorithm into a spatial computational structure is to minimize the need for memory and maximize the number of parallel computational units. This mapping is analogous to compilation in a Von Neumann processor. The present commercial realization of this technology is Field Programmable Gate Arrays (FPGAs). FPGAs were originally designed to replace small amounts of random logic; however, because they exploit the density increases of the technology so well, they have now increased in capability to the point where they offer the highest computational density of any programmable chip. The basic architecture of an FPGA is composed of 100's of thousands of primitive logical blocks which have a fully reconfigurable network that allows the logical blocks to be arbitrarily

interconnected. For example, to implement a fully IEEE compliant floating point unit requires 1500 blocks, while a 16-bit integer adder only requires 16 blocks.

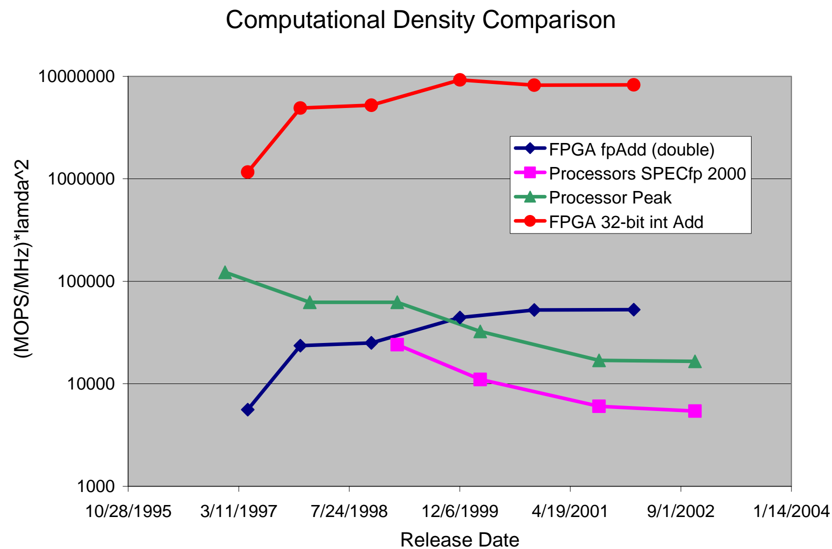


Figure 22: Computational density comparison between FPGA and Intel Processors

The state of the art FPGA in the most advanced technology are able to implement up to 86,000 MOPS (millions of operations per second), where an operation is a 16 bit add equivalent or 9,600 MFLOPS for full IEEE 754 single precision floating point operations. This is achieved by implementing a large number of simple processing elements on the same FPGA (spatial parallelism), with each processing element running at 100 MHz. Due to the fact that the basic FPGA architecture directly exploits the logic density improvements from technology scaling, the performance of an FPGA increases by a factor of 2 with each new process generation. In fact, FPGAs are now used by IC fabrication foundries to drive their newest process development because of the regular structure of an FPGA and ease of designing into a new process generation. Since each new process generation reduces the minimum feature size by a factor of 0.7, this results in a 2.4 times increase in computational throughput (assuming 2 times more units and approximately a 1.2 times increase in clock rate). The relatively low clock rates (< 250 MHz) protects the FPGA solution from the power limited computational limits being seen by modern Von Neumann processors and matches with the large capacity memory speed.

While the performance of a single FPGA is impressive, to achieve HEC computational requirements, arrays of FPGAs will be required. Instead of taking the current approach of clusters of multiprocessors, a large capacity virtual FPGA is constructed out of an array of densely connected physical FPGAs. To program a single FPGA, the algorithms of an application are expressed in Discrete Time based Block Diagrams (DTBD). These diagrams are spatially partitioned and directly mapped to the logic elements on the FPGA using automated hardware synthesis tools.

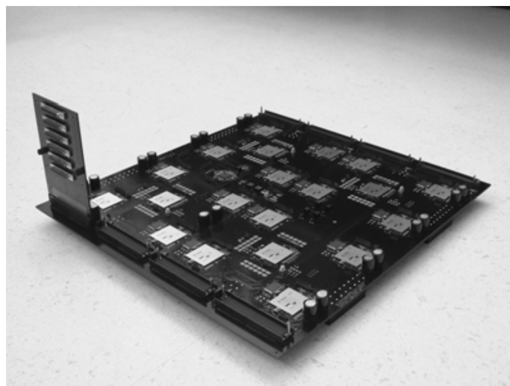
A unique characteristic of fully reconfigurable processors is that the computation architecture can be optimized for each application, and therefore the performance that can be achieved is near the maximum performance limit of the array. Because the reconfigurable interconnect is an integral part of the computing fabric, the machine is very flexible in its communications. This flexibility allows the system to emulate any number of different communication and synchronization strategies, such as message passing, static scheduled

interconnection patterns, and other ad hoc schemes on a task specific basis. Furthermore, the actual arithmetic can be optimized for each application. For legacy applications, it is possible to provide IEEE compatible arithmetic, but for many applications a more optimal arithmetic can be used as well. This optimization can provide another 2 order of magnitude increase in computational density, as shown in Figure 22. To exploit this capability optimally will require "compiler" like optimizations, which determine the accuracy of the arithmetic which is required. There has been some initial work in this area, but it remains relatively unexplored.

Modern FPGA chips typically run at a couple of hundreds of MHz, and offer large amounts of parallel I/O pins (currently up to 1200 for Xilinx Virtex II Pro series). The clock rate matches the current speed of DRAM technology and the large number of I/O pins offer the solution to the high memory bandwidth requirement by supporting up to 4 independent 8-byte-wide DDR memory channels running at 533MHz. In addition, the high throughput multi-giga bit transceivers available on FPGAs offer enough bandwidth to construct an array of FPGAs that are locally interconnected to their neighbors such that the off-chip interconnection behaves similar to the on-chip interconnection. This allows a straightforward extension of the single FPGA programming model to arrays of FPGAs through the addition of FPGA level spatial partitioning.

6.3 Berkeley Emulation Engine Platform

At UC Berkeley we have implemented an array composed of 20 FPGAs (Xilinx Virtex-E 2000), shown in Figure 23, which has the overall capability of 600 GOPS or 20GFLOPS while running at 60 MHz. An automated design flow in routine use compiles and directly "compiles" applications described using MathWorks' Simulink/Stateflow/Matlab tools directly onto the FPGA array. The array is in use by students in classes and researchers at UCB, as well as by users at a number of other Universities over the internet through an Ethernet interface into the array. The programming model while particularly optimal for stream based programming, also can support finite difference and vector based computations. It has been primarily used for real-time emulations of communication systems, signal processing, and video processing applications.



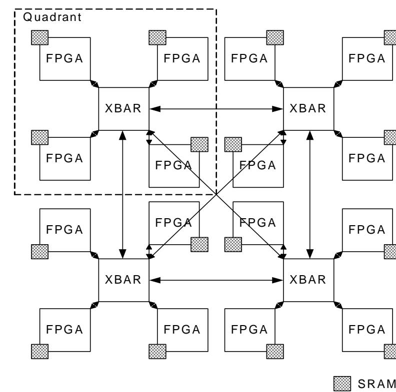


Figure 23: BEE FPGA Array & Topology Diagram

7 Reference

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. "Basic local alignment search tool," *Journal of Molecular Biology*, 215:403~410, 1990

[2] http://www.ncbi.nlm.nih.gov/Genbank/

[3] http://www.xilinx.com/products/platform/

[4] http://www.timelogic.com

[5] http://mpiblast.lanl.gov/

[6] http://www.ncbi.nlm.nih.gov/BLAST/

[7] A. Darling, L. Carey, and W. Feng, "The Design, Implementation, and Evaluation of mpiBLAST," *ClusterWorld Conference & Expo*, San Jose, CA, June 2003.

[8] R. K. Singh, W. D. Dettlo, V. L. Chi, D. L. Homan, S. G. Tell, C. T. White, S. F. Altschul, and B. W. Erickson, "BioSCAN: A dynamically reconfigurable systolic array for biosequence analysis"

[9] R. Singh, et al, "A scalable systolic multiprocessor system for analysis of biological sequences," *Proc. Symp. On Integrated Systems*, April, 1993.

[10] E. Chow, T. Hunkapiller, J. Peterson, and M. S. Waterman, "Biological information signal processor," *Proc. Int. Conf. Application Specific Array Processors*, pp. 144-160, Sep. 1991.

[11] http://www.timelogic.com/benchmark_blast.html

[12] http://www.mellanox.com