Grid Computing using DE2 and NIOS II

Calculating Prime Numbers

Presented by:
Jason Stoddart (3808589)
Jean-Francois Bibeau (3716303)

Presented to Daniel Shapiro and Zahia Aidoud
To partially satisfy the requirements of CEG4131

University of Ottawa

November 24th 2008

Contents

Introduction

Grid computing is the way of the future for solving various complex problems that would be impossible for a single organization without access to a super computer. Grid computing allows a large problem to be divided into different sections of work that can be performed independently. A central server divides the problem and assigns a portion to each client that connects to it. When the client has finished its task, it will return the result to the server and receive another portion of the problem. This ideology allows many "low-budget" computers to be used the same way as a super computer. Another advantage is that this approach can make good use of unused CPU cycles that would otherwise be wasted.

We have decided to perform prime number calculations as our grid computing task. This problem was chosen as there is no need to have a result of the previous calculation to do the next calculation. This lends itself very well to dividing a problem up into multiple sections and allowing different processors to perform each section independently.

We have decided to compare grid computing both to a multi-core computer and to a single-core computer. These platforms will be using the Altera DE2 board and running the NIOSII SOPC. The multi-core FPGA will represent a "super computer" and will consist of three cores, one for synchronization and the other two for prime number calculations. The single-core represents a "regular PC", which will be used both for the sequential and grid computing approaches. The hope of this project is to illustrate the idea of grid computing: having access to many regular PCs will give more powerful processing ability than having access to one super computer.

Explanation

Overview

The first step in any project design is planning. As such, the first few days of this project were spent on planning the architectures that would be used, as well as the algorithm used. Since the goal of this project was not to evaluate the efficiency of the various prime-finding algorithms, we decided to go with the simplest algorithm that would have the least sources of error, while keeping the complexity to a minimum. The algorithm we chose is a simple "brute-force" algorithm. An overview of the algorithm is as follows:

1. For every number x, from <lower bound> to <upper bound>, do:
    a. For every number y, from 2 to the square root of x, do:
        i. Divide x by y, if the remainder is 0, the number is <u>not</u> a prime, exit loop.
        ii. If at the end of the loop, no divisions have given a remainder of 0, the number <u>is</u> a prime.

While this algorithm is not the most efficient (we could easily remove any even number, or any factors of 5), it would be efficient enough for this project. A quick implementation in C and verified on a computer confirmed that this algorithm worked. To confirm that this algorithm worked, we cross-referenced it with The Prime Pages website [1], which allowed us to verify we had calculated the right number of primes. Additionally, we verified that our algorithm was correct by researching various algorithms available dedicated to finding prime numbers [2].

Once we had our algorithm confirmed, the next step was to find our upper bound, that is, the maximum number to which we would like to find all prime numbers. The number in question had to allow us to run our multiple implementations within a reasonable time frame, yet be large enough to provide us enough room to compare and calculate speedups. To find this number, we ran our prime number algorithm on a single-core NIOS II CPU running at 100MHz on the DE2 board. By trying various numbers, we reached the conclusion that one million (1 000 000) would be a good number. Calculating every prime number up to a million took roughly 11 minutes, which fit both of our previously established criteria. Now that we had our algorithm and upper bound, we were ready to start doing the specific implementations.

While we would be using three distinct implementations for calculating prime numbers, we only needed two separate NIOS II designs. The sequential method would use the same design as the triple-core method, to ensure consistency. Additionally, to provide the best parallelism and avoid idle processor time, we divided the triple-core workload into "uneven" slices. Since our algorithm gets slower as numbers increase in size, the last half of the calculation will take significantly more time than the first half. As such, instead of dividing the problem into two equal halves, we assigned the first 2/3^{rds} of the calculation to CPU 1, and the last 1/3rd of the calculation to CPU 2. This ensured that both CPUs were always working, and provided a much more accurate result than splitting the work evenly.

Building the Triple-Core System

The triple-core system we designed was built from the ground-up for this project. The goal was to provide the appropriate memory components and size to reduce contention between the three CPUs. We wanted to eliminate the need to "busy-wait" for synchronization, as this proved to be the main reason for memory contention, as experienced in laboratory 2. Additionally, we wanted to introduce the least amount of overhead while the prime numbers were calculating. To achieve all these goals, we decided to use both on-chip memory and SDRAM memory. The exact implementation we decided to go with is included below:

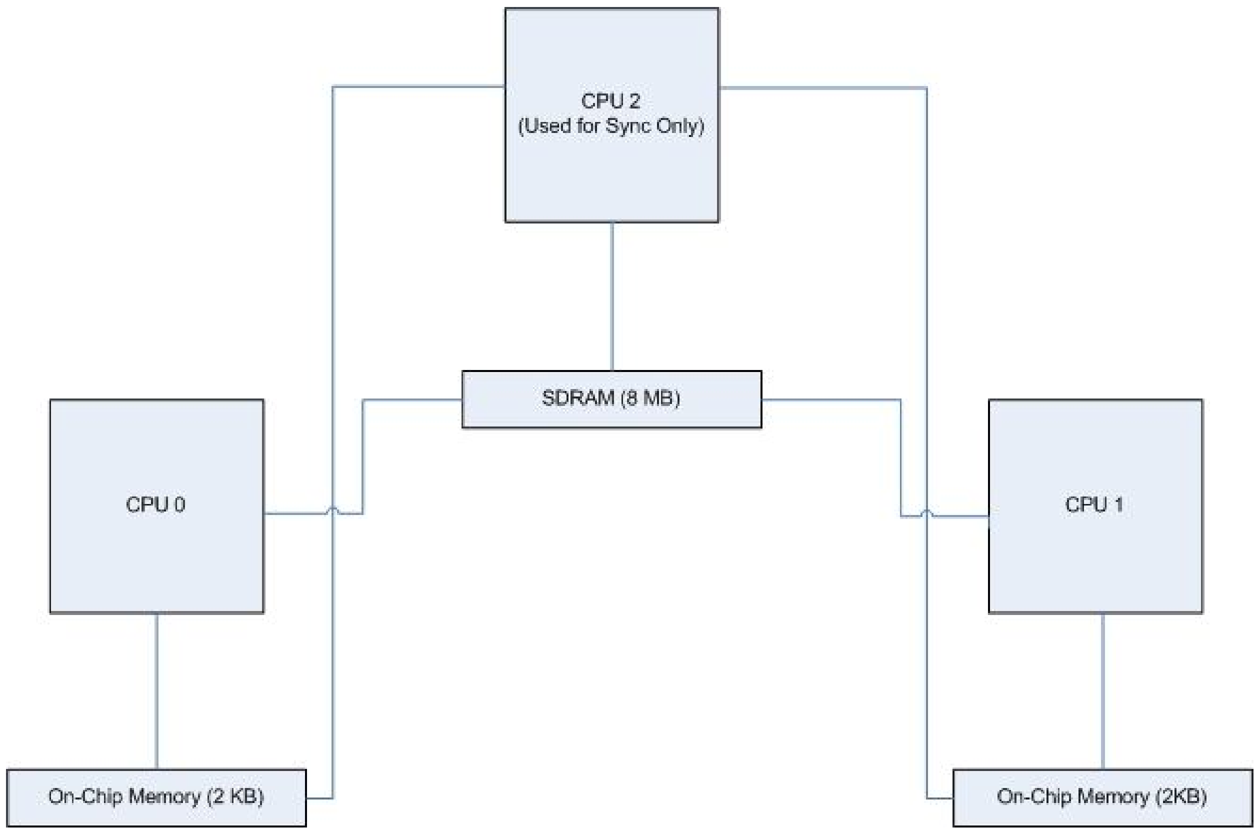


Figure 1: Diagram of CPU and memory integration

Each CPU doing the calculations is connected to its own dual-port memory, with the other port being connected to the master CPU. This memory is only used for synchronization; as such we eliminate any contention between the master and the slaves. Once the CPUs have started doing their calculation, they store all the results on the SDRAM memory. Each CPU has their own partition inside the SDRAM, to avoid the use of time costly mutex locks and unlocks. During the calculation process, CPU2 is sleeping and waiting for status flags to be lifted in the on-chip memory of each CPU. Only when both CPUs have completed their calculation does CPU 2 attempt to read the SDRAM. That is, when both CPUs are done using the SDRAM to do their calculation, CPU 2 will go and read each result sequentially, and output it via the JTAG UART. The design also incorporates a performance counter, to accurately measure the performance of our algorithm, as well as standard components such as individual timers for each CPU. Each CPU runs at 100MHz, while the SDRAM runs at 50MHz. While storing the results in on-chip memory would have eliminated contention from the system completely, the size restriction on the on-chip memory prevented us from implementing such a solution.

The high level block diagram of Quartus is included below:

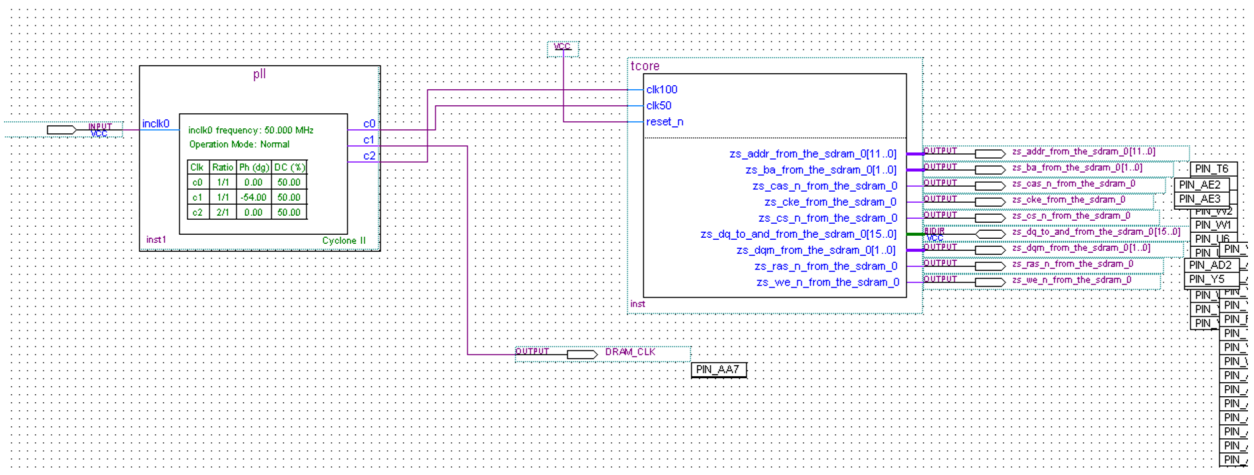
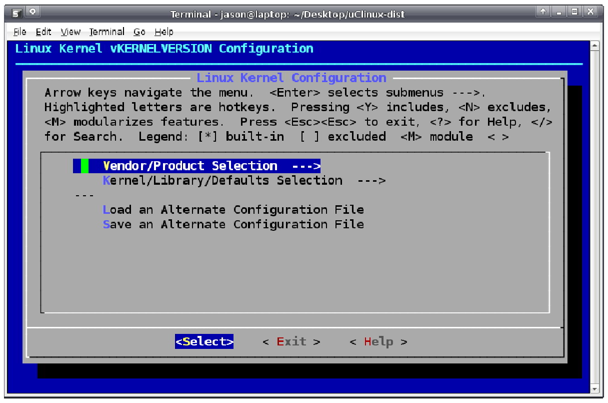


Figure 2: Block diagram of triple-core system

Running uCLinux on the NIOS II

The second implementation we did was a simpler version of the triple-core system, retaining the SDRAM memory but keeping only one core. However, the design had to add the DMA9000A Ethernet component. This interface will be used during the grid computing process to access the network and communicate with the server. The specifications for the NIOS II were kept the same to ensure consistency in the results. Once the SOPC system was generated, we needed to find all the correct drivers to allow uCLinux to work on top of the NIOS II; this included drivers for the Ethernet component, the serial port, and the SDRAM controller. All these and the SOPC system were combined and used to compile the uCLinux kernel, using a compiled tool-chain made specifically for uCLinux. This image was loaded into the SDRAM and the CPU was pointed to this address, allowing it to boot into uCLinux when restarted.

The main advantage and reason for using uCLinux, and thus going to a higher level than simply programming in C, is the inclusion of a full featured TCP/IP stack. Without at least a minimal TCP/IP stack, we would not be able to do grid computing over an Ethernet network. The overhead introduced by an operating system as small as uCLinux was an acceptable trade off to allow us to run our grid computing algorithm. Some screenshots showing the compiling process and successful boot of uCLinux are included below:

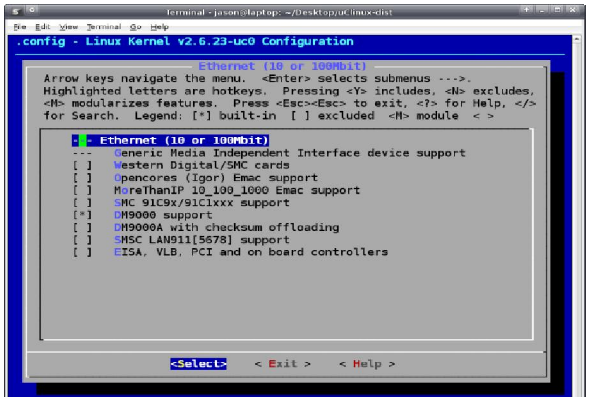


```
Linux Kernel vKERNELVERSION Configuration
                 Linux Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

        Vendor/Product Selection  --->
        Kernel/Library/Defaults Selection  --->
    ---
        Load an Alternate Configuration File
        Save an Alternate Configuration File

              <Select>    < Exit >    < Help >
```

Figure 3: Linux kernel compilation



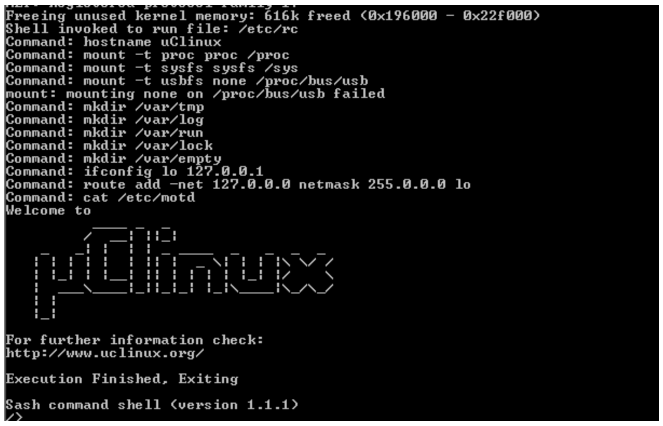
```
.config - Linux Kernel v2.6.23-uc0 Configuration
                 Ethernet (10 or 100Mbit)
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

      --- Ethernet (10 or 100Mbit)
      ---    Generic Media Independent Interface device support
      [ ]    Western Digital/SMC cards
      [ ]    Opencores (Igor) Emac support
      [ ]    MoreThanIP 10_100_1000 Emac support
      [ ]    SMC 91C9x/91C1xxx support
      [*]    DM9000 support
      [ ]    DM9000A with checksum offloading
      [ ]    SMSC LAN911[5678] support
      [ ]    EISA, VLB, PCI and on board controllers

              <Select>    < Exit >    < Help >
```

Terminal - jason@laptop: ~/Desktop/uClinux-dist
File Edit View Terminal Go Help

Figure 4: Integrating Ethernet drivers



```
Freeing unused kernel memory: 616k freed (0x196000 - 0x22f000)
Shell invoked to run file: /etc/rc
Command: hostname uClinux
Command: mount -t proc proc /proc
Command: mount -t sysfs sysfs /sys
Command: mount -t usbfs none /proc/bus/usb
mount: mounting none on /proc/bus/usb failed
Command: mkdir /var/tmp
Command: mkdir /var/log
Command: mkdir /var/run
Command: mkdir /var/lock
Command: mkdir /var/empty
Command: ifconfig lo 127.0.0.1
Command: route add -net 127.0.0.0 netmask 255.0.0.0 lo
Command: cat /etc/motd
Welcome to

For further information check:
http://www.uclinux.org/

Execution Finished, Exiting

Sash command shell (version 1.1.1)
/>
```

Figure 5: uCLinux booting successfully

Programming

All programming for the three implementations was done in C. The sequential and triple-core implementations were programmed directly in the NIOS IDE and loaded with this environment. The grid-computing software was loaded on the board along with the generated uCLinux image, while the server was running on a separate Linux laptop. C structures were used to allow easy manipulation of the data stored in memory from all processors. All of the code can be found in the appendices.

Results

In this section, the denotation “Triple-core” references the design where two cores are working on the calculation, while the third is only used for synchronization. In effect, this makes it equivalent to a “dual-core” design.

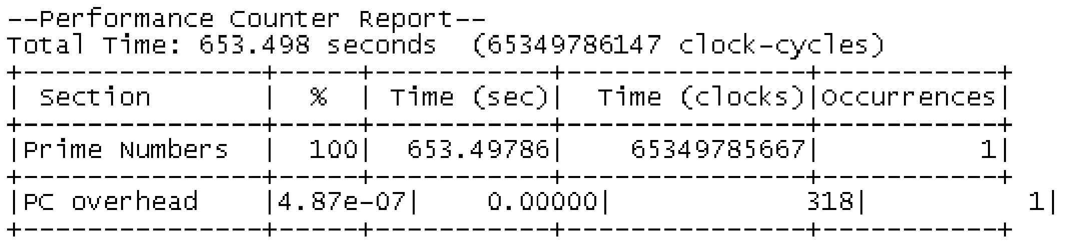
Overall Completion Times

			Grid Computing		
	Single-core	**Triple-core**	**Single DE2**	**Dual DE2**	**Triple DE2**
Time to Completion (s)	653.5	403.6	747	461	290

Table 1: Completion Time

The above table illustrates the time that each implementation took to complete the calculation of all primes between 2 and 1,000,000. At a glance, we can observe that the grid computing approach with three DE2 boards offers a considerable speedup. The difference between dual DE2 boards and the triple-core design can be attributed to the overhead involved in grid-computing.

Single-core Performance

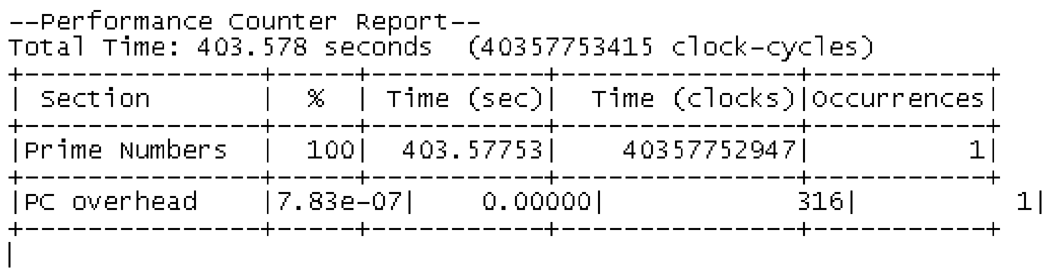


```
--Performance Counter Report--
Total Time: 653.498 seconds  (65349786147 clock-cycles)
+---------------+-----+-----------+---------------+-----------+
| Section       |  %  | Time (sec)|  Time (clocks)|Occurrences|
+---------------+-----+-----------+---------------+-----------+
|Prime Numbers  |  100|  653.49786|    65349785667|          1|
+---------------+-----+-----------+---------------+-----------+
|PC overhead    |4.87e-07|    0.00000|            318|          1|
+---------------+-----+-----------+---------------+-----------+
```

Figure 6: Performance Counter (single-core)

The above figure shows the performance counter of the single-core implementation after running the prime numbers calculation. This is the benchmark on which every calculation will be based as it represents the simplest way of solving the problem.

Triple-Core Performance



```
--Performance Counter Report--
Total Time: 403.578 seconds  (40357753415 clock-cycles)
+---------------+-----+-----------+---------------+-----------+
| Section       |  %  | Time (sec)|  Time (clocks)|Occurrences|
+---------------+-----+-----------+---------------+-----------+
|Prime Numbers  |  100|  403.57753|    40357752947|          1|
+---------------+-----+-----------+---------------+-----------+
|PC overhead    |7.83e-07|    0.00000|            316|          1|
+---------------+-----+-----------+---------------+-----------+
|
```

Figure 7: Performance Counter (triple-core)

The above figure shows the performance counter of the triple-core implementation after running the prime numbers calculation. To prove that our grid computing approach worked, we needed to find an implementation that would beat this time. As the results show, the grid computing method with 3 DE2 boards was able to beat this time by a significant margin, as expected.

Speedup

				Grid Computing		
	Speed Up (Y/X)	**Single-core**	**Triple-core**	**Single DE2**	**Dual DE2**	**Triple DE2**
	Single-core	1	1.62	0.87	1.42	2.25
	Triple-core	0.62	1	0.54	0.88	1.39
Grid Computing	**Single DE2**	1.14	1.85	1	1.62	2.58
	Dual DE2	0.71	1.14	0.62	1	1.59
	Triple DE2	0.44	0.72	0.39	0.63	1

Table 2: Speedup

The above table shows the speed up calculations (Y/X) for all of the implementations that we had tried. As can be seen from the table the best speed up was achieved using grid computing going from one board to three boards. These results illustrate what we expected before beginning this project.

Efficiency

	Triple-core	Dual DE2	Triple DE2
Efficiency	$\frac{1.62}{2} = 0.81$	$\frac{1.62}{2} = 0.81$	$\frac{2.58}{3} = 0.86$

Table 3: Efficiency

The table above shows the relevant efficiencies for the grid and multi-core systems. Surprisingly the efficiency for the triple-core and dual DE2 grid computing are the same. This shows that the overhead caused from grid computing has no effect on the efficiency.

Complexity

The following graph plots all three attempts at grid-computing using multiple FPGAs. As can be seen, an exponential complexity factor results. Although overall calculation speeds decrease, overhead time increases as the number of FPGAs increase. This will result in a decreasing improvement as the number of clients increase, although an overall improvement should always be achieved. This could be improved with better code designed to reduce overhead. Generally, grid computing software would allow the client to do as much work as possible before reporting to the server, thus reducing overhead. In our case, since the work slices were relatively small, the overhead was significant.

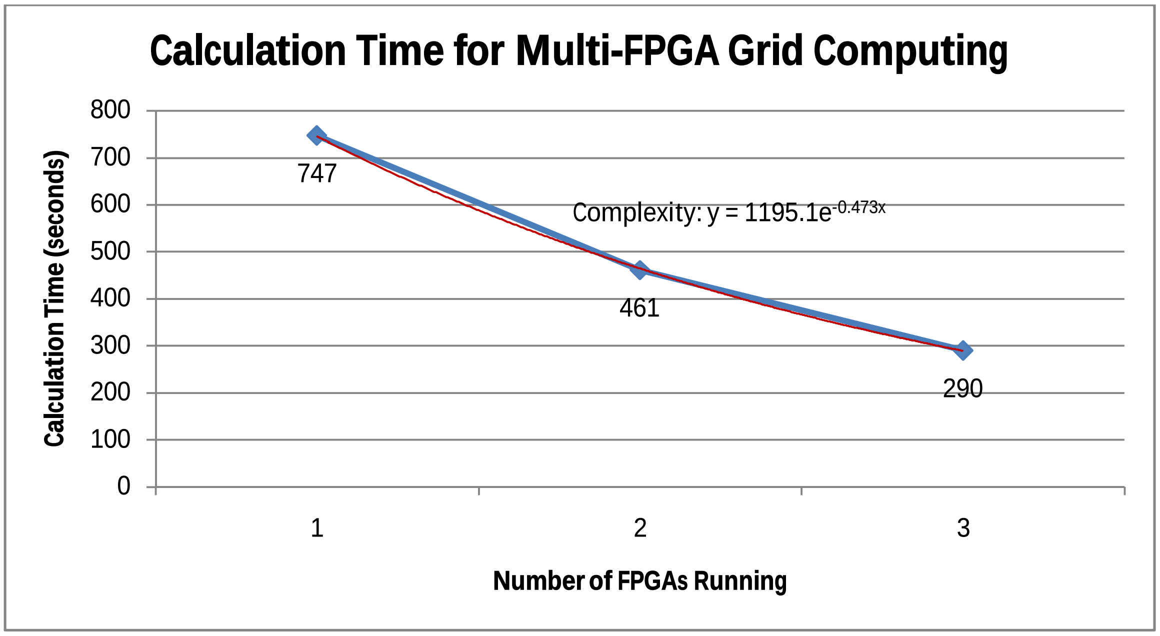


Figure 8: Grid Computing Complexity

Discussion

As can be seen from our results, our project was successful in demonstrating the power of grid computing. By combining three DE2 boards with a single-core each, we were able to exceed the speed of the triple-core NIOS design. The potential scalability of grid computing greatly exceeds the ability for a multi-core system to scale. With our current design, we would be limited to approximately 11 cores on the DE2 board, whereas grid computing is only limited by the number of divisions that a problem can be split into without becoming dependent on another process completing. The overhead created by using a network is very small, as can be seen from the comparison between the grid computing using one DE2 board and the sequential execution in the NIOSII IDE. This overhead is also inflated as each client must return all the prime numbers that it found. If we were to only try to find the number of primes between 2 and 1 million the overhead would be less, as each client would only have to return the number of primes it found in its section.

We encountered a few issues during this project, the main ones were introduced in previous works/labs and mitigation was planned before implementation. The main issue was introduced in the second lab of this class, and was related to our triple-core design. The issue was caused by the sync CPU accessing the shared memory so often that it blocked the worker CPUs from accessing the memory. To avoid this issue, we used two on-chip dual port memories for maintaining the status of each CPU. This allowed the sync CPU to access the memory without preventing another CPU from accessing it. Busy-waiting was also avoided as much as possible.

Additionally, we experienced issues trying to get the NIOSII IDE to download the elf file to the NIOS core, resulting in failures for our application to run. This was a result of not having the correct clock advancement settings in the PLL to the SDRAM chip. Problems that were encountered while implementing the grid computing section had to do with the number of packets being sent back to the server and the server not being able to service them fast enough. This problem was created because the server is only a single thread and the clients sent a new packet every time it found a new prime number. To resolve this issue we changed the clients so that they would send a packet when they had found 200 prime numbers. This alleviated the load on the server and allowed all clients to finish normally.

The final issue we encountered was caused by errors in our C program. Since we had to allocate space for all the prime numbers in the SDRAM, and the SDRAM was also partitioned into three spaces for the three CPUs, we could not just assign a random address for the prime numbers. If we assigned a random value for the address of the prime numbers, they would be overwritten by the stack/heap/program memory. This was very simple to overcome and simply just involved having C allocate the memory needed for the prime numbers and passing the address of this space to each of the working CPUs.

Conclusion

As can be seen from the results and our hypothesis, our experiment in demonstrating the scalability and power of grid computing was successful. The networking added very little overhead and we proved that it was much more scalable and cheaper than a multi-core system.

To be able to perform hardware acceleration, which could increase the computational power of a system, it is very likely that FPGAs could be used. This is because hardware acceleration needs the processor to be augmented to be application-specific. This would be much more feasible on a custom soft-core processor, as opposed to designing a custom ASIC processor [3]. FPGAs also bring ease-of-use and lower power consumption [3]. The table below demonstrates that having an FPGA as a coprocessor brings about a significant speedup to a specific application:

Application	Processor Only	FPGA Co-processing	Acceleration Factor
Hough and Inverse Hough Processing	12 minutes processing time Pentium 4-3 GHz	2 seconds of processing time at 20 MHz	370X

Table 4: FPGA as a Co-Processor

We've shown that FPGAs are flexible and powerful, and that grid computing scales extremely well. There's no reason why these two concepts should not be combined together to create an even more powerful architecture for heavy processing. This is exactly what the RAMP project at Berkeley [4] is trying to achieve. As mentioned in our project, we would be able to include about a dozen CPU cores on a single FPGA. By using these dense FPGAs, and combining several boards, we can utilize the flexibility and processing power of the FPGAs, while incorporating the scalability of grid computing. Ultimately, this is the future of complex computational problem solving.

Appendix A – Code for the Sequential Approach

```
#include <stdio.h>
#include <system.h>
#include <unistd.h>
#include "altera_avalon_performance_counter.h"

void findPrimes();

int main()
{

  findPrimes(2, 1000000);

  return 0;
}

void findPrimes(int lBound, int uBound) {
    long primes[78495];
    int count = 0;

    /* Start performance counter */
    PERF_RESET (PERFORMANCE_COUNTER_0_BASE);

    PERF_START_MEASURING (PERFORMANCE_COUNTER_0_BASE);

    PERF_BEGIN (PERFORMANCE_COUNTER_0_BASE,2);
        PERF_BEGIN (PERFORMANCE_COUNTER_0_BASE,1);
    PERF_END (PERFORMANCE_COUNTER_0_BASE,2);

    while(lBound <= uBound) {
        long trialDivisor = 2;
        int prime = 1;
        while(trialDivisor * trialDivisor <= lBound) {
            if(lBound % trialDivisor == 0) {
                prime = 0;
                break;
            }
            trialDivisor++;
        }
        if(prime) {
            primes[count] = lBound;
            count++;
            if (count % 10000 == 0)  {
                printf("found 10,000 primes \n");
            }
        }
        lBound++;
    }
    PERF_END (PERFORMANCE_COUNTER_0_BASE,1);            //Stop the Matrix Multiplication Counter
    PERF_STOP_MEASURING (PERFORMANCE_COUNTER_0_BASE);   //Stop all counters

    perf_print_formatted_report((void *)PERFORMANCE_COUNTER_0_BASE, ALT_CPU_FREQ, 2,
    "Prime Numbers","PC overhead");

    printf("primes found: \n");
    for (i = 0; i < sizeof(primes)/sizeof(long); i++ ) {
        printf("%ld ", primes[i]);
    }
}
```

Appendix B – Code for the Triple-Core Approach

CPU 0 Synchronization Code

```
#include <stdio.h>
#include <system.h>
#include <unistd.h>
#include "structs.h"
#include "altera_avalon_performance_counter.h"

/* CPU 0 */
int main()
{
  /* SDRAM */
  cpu1_struct cpu1;
  cpu2_struct cpu2;
  //cpu1 = (cpu1_struct*)(SDRAM_0_BASE + 0xFFFF);
  //cpu2 = (cpu2_struct*)(SDRAM_0_BASE + 0xFFFF + sizeof(cpu1_struct));

  /* ONCHIP (2KB) */
  status_struct *cpu1_status;
  status_struct *cpu2_status;
  cpu1_status = (status_struct*)CPU1_ONCHIP_BASE;
  cpu2_status = (status_struct*)CPU2_ONCHIP_BASE;

  /* Start both CPUs */
  cpu1.lBound = 2;
  cpu1.uBound = 666666;
  cpu2.lBound = 666667;
  cpu2.uBound = 1000000;

  int i;
  printf("Initializing arrays.\n");
  for (i = 0; i < sizeof(cpu1.primes)/sizeof(long); i++ ) {
    cpu1.primes[i] = 0;
  }

  for (i = 0; i < sizeof(cpu2.primes)/sizeof(long); i++ ) {
    cpu2.primes[i] = 0;
  }

  cpu1_status->dataLocation=&cpu1;
  cpu2_status->dataLocation=&cpu2;

  cpu1_status->done = 0x00;
  cpu2_status->done = 0x00;

  printf("CPUs 2 and 3 starting.\n");

  /* Start performance counter */
  PERF_RESET (PERFORMANCE_COUNTER_0_BASE);

  PERF_START_MEASURING (PERFORMANCE_COUNTER_0_BASE);

  PERF_BEGIN (PERFORMANCE_COUNTER_0_BASE,2);
    PERF_BEGIN (PERFORMANCE_COUNTER_0_BASE,1);
  PERF_END (PERFORMANCE_COUNTER_0_BASE,2);

  cpu1_status->ready = 0x01;
  cpu2_status->ready = 0x01;

  int counter = 0;

  while(1) {
    usleep(10000000); /* Poll every 10 seconds */
    counter++;
    if ( counter % 6 == 0 ) {
        /* Update every minute with the progress */
        printf("CPU1 is at %ld ", cpu1.progress);
        printf("CPU2 is at %ld \n", cpu2.progress);
    }

    if ( cpu1_status->done == 0x01 ) {
        if ( cpu2_status->done == 0x01 ) {
            break;
        }
    }
  }

  PERF_END (PERFORMANCE_COUNTER_0_BASE,1);            //Stop the Matrix Multiplication Counter
  PERF_STOP_MEASURING (PERFORMANCE_COUNTER_0_BASE);   //Stop all counters

  perf_print_formatted_report((void *)PERFORMANCE_COUNTER_0_BASE, ALT_CPU_FREQ, 2,
  "Prime Numbers","PC overhead");

  /*
  printf("cpu1 primes: \n");
  for (i = 0; i < sizeof(cpu1.primes)/sizeof(long); i++ ) {
    printf("%ld ", cpu1.primes[i]);
  }

  printf("cpu2 primes: \n");
  for (i = 0; i < sizeof(cpu2.primes)/sizeof(long); i++ ) {
    printf("%ld ", cpu2.primes[i]);
  }
  */

  printf("Done.\n");

  return 0;
}
```

CPU 1 and 2 Calculation Code and Structures Used

```
#include <stdio.h>
#include <system.h>
#include <unistd.h>
#include "structs.h"

void findPrimes();

/* CPU 1 */
int main()
{
  /* ONCHIP (2KB) */
  status_struct *cpu1_status;
  cpu1_status = (status_struct*)CPU1_ONCHIP_BASE;

  while(1) {
    if ( cpu1_status->ready == 0x01 ) {
        break;
    }
    usleep(1000);
  }

  findPrimes(cpu1_status->dataLocation);

  cpu1_status->done = 0x01;

  return 0;
}

void findPrimes(int location) {
  /* SDRAM */
  cpu1_struct *cpu1;
  cpu1 = (cpu1_struct*)location;

    long lBound = cpu1->lBound;
    long uBound = cpu1->uBound;
    long count = 0;
    long progcount = 0;
    cpu1->lBound++;

    while(lBound <= uBound) {
        long trialDivisor = 2;
        int prime = 1;
        while(trialDivisor * trialDivisor <= lBound) {
            if(lBound % trialDivisor == 0) {
                prime = 0;
                break;
            }
            trialDivisor++;
        }
        if(prime) {
            cpu1->primes[count] = lBound;
            count++;
        }
        lBound++;
        progcount++;
        if ( progcount % 1000 == 0 ) {
            cpu1->progress = lBound;
        }
    }
}

typedef struct {
  long lBound;
  long uBound;
  long progress;
  long primes[54069];
} cpu1_struct;

typedef struct {
  long lBound;
  long uBound;
  long progress;
  long primes[24429];
} cpu2_struct;

typedef struct {
  int ready;
  int done;
  int dataLocation;
} status_struct;
```

Appendix C – Code for the Grid Computing Approach

Server Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/in_systm.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include "prime.h"
#include <time.h>

#define echoServPort   8888

int main(int argc, char* argv[]) {
        if ( argc != 2 ) {
                printf("Usage: server <max>\n");
                return 0;
        }

        long ubound = atol(argv[1]);
        int servSock;
        long currentID=0;
        long numOfPrimes=0;
        struct sockaddr_in echoServAddr;
        long clientsDone=0;
        time_t start, endTime;
        //long primes[ubound]={};
        printf("Starting up Distributed Prime Number Finder (DPNF) Server\n");
        printf("Will find every prime number up to %d \n\n", ubound);

        /* Create socket for incoming connections */
        if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
            printf("Couldn't create TCP socket \n");
            return 0;
        }

        echoServAddr.sin_family = AF_INET;                          /* Internet address family */
        echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);/* Any incoming interface */
        echoServAddr.sin_port = htons(echoServPort);           /* Local port */

        if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0) {
            printf("Couldn't bind to port %d \n", echoServPort);
            return 0;
        }

        /* Mark the socket so it will listen for incoming connections */
        if (listen(servSock, 50) < 0) {
            printf("Couldn't call listen()\n");
            return 0;
        }

        printf("Waiting for connection on port %d...\n", echoServPort);
        struct packet recvPacket;
        struct packet sendPacket;
        for (;;) /* Run forever */
        {
                int clntLen, clntSock, i;
                struct sockaddr echoClntAddr;
                unsigned char echoBuffer[2];
            clntLen = sizeof(echoClntAddr);
                bzero(&echoBuffer, sizeof(echoBuffer));
            if ((clntSock=accept(servSock,(struct sockaddr *)&echoClntAddr,&clntLen)) < 0)
            {
               printf("Couldn't accept connection");
            }
            else
            {
                   int recvMsgSize = 0;
                              if ((recvMsgSize = recv(clntSock, &recvPacket, sizeof(struct
packet), 0)) < 0)
                              {
                      printf("Couldn't receive message\n");
                   }
                   else
                   {
                              printf("received packet header: %x\n",recvPacket.header);
                              if(recvPacket.header == headerID)
                              {
                                     if (currentID == 0)
                                            start = time(NULL);
                                     if (currentID<division)
                                     {
                                            sendPacket.header=headerBound;
                                                   sendPacket.ID=currentID;
                                                   sendPacket.lBound=ubound/division*currentID;

       sendPacket.uBound=ubound/division*(currentID+1);
                                                   currentID++;
                                            }
                                            else
                                            {
                                                   sendPacket.header=headerEnd;
                                            }
                                            send(clntSock, &sendPacket, sizeof(struct
packet),0);

                              }
                              else if(recvPacket.header == headerPrime)
                              {
                                     //primes[numOfPrimes]=recvPacket.lBound;
                                     //numOfPrimes++;
                                     //printf("%d\n",recvPacket.lBound);
                                     printf("Client: %d found %d
primes\n",recvPacket.ID,recvPacket.lBound);
                                     long myPrimeArray[recvPacket.lBound];
                                     //myPrimeArray = (long
*)malloc(sizeof(long)*recvPacket.lBound);
                                     if ((recvMsgSize = recv(clntSock, &myPrimeArray,
recvPacket.lBound*sizeof(long), 0)) < 0)
                                            {
                                     printf("Couldn't receive message\n");
                                 }
                                 else
                                 {
                                     int q =0;
                                     for(q=0; q<recvPacket.lBound;q++)
                                     {
                                            printf("%d\n",myPrimeArray[q]);
                                            if
((myPrimeArray[q]==myPrimeArray[q+1])||(myPrimeArray[q]>ubound)){
                                                   printf("ERROR!\n");
                                                   exit(-1);
                                            }
                                     }
                                 }
                                 //free(myPrimeArray);


                              }
                              else if(recvPacket.header == headerEnd)
                              {
                                     clientsDone++;
                                            printf("Client: %d has finished\n",recvPacket.ID);
                                            if (clientsDone==division){
                                                    endTime = time(NULL);
                                                    close(clntSock);
                                                    close(servSock);
                                                    printf("done in %f
seconds\n",difftime(endTime,start));

                                                    return 0;
                                            }

                              }
                              else
                              {
                                      printf("Unknown header: %x from ID: %d\n",
recvPacket.header,recvPacket.ID);
                              }



                         }
                        }
                  close(clntSock);

        }
        return 0;

}
```

Client Code and Structures Used

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/in_systm.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include "prime.h"


int main(int argc, char* argv[]) {
        if ( argc != 2 ) {
                        printf("Usage: client <max packet size>\n");
                        return 0;
        }
        long sizeOfBuf = atol(argv[1]);
        while(1)
        {
                struct packet myID;
                getClientID(&myID);
                printf("client ID is: %d lbound is: %d uBound is:
%d\n",myID.ID,myID.lBound,myID.uBound);
                if (myID.header == headerEnd)
                {
                        printf("no more work to do\n");
                        return 0;
                }
                findPrimes(&myID,sizeOfBuf);
                myID.header=headerEnd;
                sendToServer(myID);
        }

        /*printf("Sending...\n");
        unsigned char packet[2] = { 0xAA, 0xBB };
        sendToServer(packet, sizeof(packet));
        printf("Sent.\n");*/
}
```

```
int sendToServer(struct packet sendPacket)
{
        int i;

        struct sockaddr_in serv_addr, cli_addr;       ;
        int sock, newsockfd, clilen;

        // Create the socket
        sock = socket (AF_INET, SOCK_STREAM, 0);

        // Init the struct
        bzero((char*)&serv_addr,sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;

        /* serverIP and serverPort defined in system_definitions.h */
        serv_addr.sin_port = htons (serverPort);
        inet_aton(serverIP, &serv_addr.sin_addr);

        if (connect (sock, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        {
                printf("[ERROR] Could not connect to server.\n");
                exit(-1);
                return 0;
        }

        // Send the command
        send(sock, &sendPacket, sizeof(struct packet), 0);


        // Close the socket
        close(sock);
}

int getClientID(struct packet *myIDHolder)
{
        int i;

        struct sockaddr_in serv_addr, cli_addr;       ;
        int sock, newsockfd, clilen;

        struct packet *myID ;
        myID = (struct packet *)myIDHolder;
        // Create the socket
        sock = socket (AF_INET, SOCK_STREAM, 0);

        // Init the struct
        bzero((char*)&serv_addr,sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;

        /* serverIP and serverPort defined in system_definitions.h */
        serv_addr.sin_port = htons (serverPort);
        inet_aton(serverIP, &serv_addr.sin_addr);

        if (connect (sock, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        {
                printf("[ERROR] Could not connect to server.\n");
                exit(-1);
                return 0;

        }
        myID->header = 0xBB;
        send(sock, myID, sizeof(struct packet), 0);
        if(recv(sock, myID, sizeof(struct packet),0)<0)
        {
                printf("failed to receive ID\n");
                exit(-1);
        }
        else
        {
                printf("header: %x client ID is: %d lbound is: %d uBound is: %d\n",myID-
>header,myID->ID,myID->lBound,myID->uBound);
```

```
        }
        return 0;
}

int findPrimes(struct packet *myIDHolder, long sizeOfBuf)
{

                int i =1;
                struct packet *myID ;
                myID = (struct packet *)myIDHolder;
                struct packet myPrime;
                myPrime.ID = myID->ID;
                myPrime.header = headerPrime;
                printf("From findPrimes client ID is: %d lbound is: %d uBound is: %d\n",myID-
>ID,myID->lBound,myID->uBound);
                long *myPrimeArray;
                myPrimeArray = (long *)malloc(i*sizeof(long));
                while(myID->lBound < myID->uBound)
                {
                        //printf("checking if %d is prime\n", myID.lBound);
                        long trialDivisor = 2;
                        int prime = 1;
                        myPrimeArray = (long *)realloc(myPrimeArray,i*sizeof(long));

                        while(trialDivisor * trialDivisor <= myID->lBound)
                        {
                                if(myID->lBound % trialDivisor == 0)
                                {
                                        prime = 0;
                                        break;
                                }
                                trialDivisor++;
                        }
                        if(prime)
                        {

                                myPrimeArray[i-1]=myID->lBound;
                                //printf("%d\n", myPrime.lBound);

                                /*if we have over 2000 send now*/

                                if(i>sizeOfBuf)
                                {
                                        //printf("%d sending now\n",i);
                                        myPrime.lBound=i;
                                        //sendToServer(myPrime);
                                        int q =0;

                                        sendPrimesToServer(myPrime,myPrimeArray,i);
                                        //free(myPrimeArray);
                                        i=0;
                                }
                                i++;
                        }
                        myID->lBound++;
                }
                if (i-1>0)
                {

                        myPrime.lBound=i-1;
                        //sendToServer(myPrime);
                        int q =0;

                        sendPrimesToServer(myPrime,myPrimeArray,i-1);
                        free(myPrimeArray);
                }

        return 0;
}
```

```
int sendPrimesToServer(struct packet sendPacket, long *myPrimeArray, long size)
{
        int i;
        int q;
        struct sockaddr_in serv_addr, cli_addr;
        int sock, newsockfd, clilen;
        // Create the socket
        sock = socket (AF_INET, SOCK_STREAM, 0);
        for (q=0; q<size;q++)
        {
                printf("%d\n",myPrimeArray[q]);
        }

        // Init the struct
        bzero((char*)&serv_addr,sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;

        /* serverIP and serverPort defined in system_definitions.h */
        serv_addr.sin_port = htons (serverPort);
        inet_aton(serverIP, &serv_addr.sin_addr);

        if (connect (sock, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        {
                printf("[ERROR] Could not connect to server.\n");
                exit(-1);
                return 0;
        }

        // Send the command
        send(sock, &sendPacket, sizeof(struct packet), 0);
        send(sock, myPrimeArray, size*sizeof(long), 0);


        // Close the socket
        close(sock);
        return 0;
}

void bzero (to, count)
  char *to;
  int count;
{
  while (count-- > 0)
    {
      *to++ = 0;
    }
}


#define headerID 0xBB
#define headerBound 0xAA
#define headerPrime 0xFF
#define headerEnd     0xDD
struct packet {
        unsigned char header;
        long ID;
        long lBound;
        long uBound;
};

int getClientID(struct packet *myIDHolder);


/* client use */
#define serverIP                    "192.168.0.230"
#define serverPort                  8888
#define clientID                    0x05

/* server use */
#define echoServPort                8888
#define division                    10
```

References

[1] Prof. Chris Caldwell, “The Prime Pages”, 2008. [Online]. Available: http://primes.utm.edu [Accessed: Nov. 6, 2008].

[2] Steve Litt, “Fun With Prime Numbers”, 2004. [Online]. Available: http://www.troubleshooters.com/codecorn/primenumbers/primenumbers.htm [Accessed: Nov. 5, 2008]

[3] Altera Corporation, “Accelerating High-Performance Computing With FPGAs”, October 2007. [Online]. Available: http://www.altera.com/literature/wp/wp-01029.pdf [Accessed: Nov. 18, 2008]

[4] Greg Gibeling, Andrew Schultz & Krste Asanovi´c, “The RAMP Architecture & Description Language”, January 17, 2006. [Online]. Available: http://ramp.eecs.berkeley.edu/Publications/The%20RAMP%20Architecture%20and%20Description%20Language.pdf [Accessed: Nov. 22, 2008]