Appendix: Accelerating Financial Applications with Specialized Hardware – FPGA

A Major Qualifying Project Report
Submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements
for the Degree of Bachelor of Science

By:

_____

Tri Dang

_____

John Rothermel

Date: December 14, 2007

Report Submitted To:

Prof. Xinming Huang, WPI

Prof. Arthur Gerstenfeld, WPI

Prof. Michael Ciaraldi, WPI

Mr. Jason Choy and JP Morgan Chase Inc.

Appendix Contents

Appendix A: Opteron Benchmark Accuracy Test

A.1: Arithmetic

```
Addition results:
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
Subtraction results:
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
Multiplication results:
```

```
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
Division results:
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
Square Root results:
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
```

```
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0.00000000000000000000000000000000e+00
```

A.2: Transcendental

```
START COMPARING (compares all files to x87)
      Order: sse, x87fm, ssefm, x87UL, sseUL, x87fmUL, ssefmUL
Sine results:
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 2.44065031409263610839843750000000e-01
The Average Ulp was: 267470304.000000
The Max error was: 1.00000000000000000000000000000000e+00
The Max Ulp was: 1665184224
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 2.44065031409263610839843750000000e-01
The Average Ulp was: 267470304.000000
The Max error was: 1.00000000000000000000000000000000e+00
The Max Ulp was: 1665184224
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
Cosine results:
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 2.43995383381843566894531250000000e-01
The Average Ulp was: 266946672.000000
The Max error was: 1.00000000000000000000000000000000e+00
The Max Ulp was: 1655989932
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
```

```
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 2.43995383381843566894531250000000e-01
The Average Ulp was: 266946672.000000
The Max error was: 1.00000000000000000000000000000000e+00
The Max Ulp was: 1655989932
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
Tangent results:
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 4.60023875348269939422607421875000e-04
The Average Ulp was: 264036256.000000
The Max error was: 3.94126325845718383789062500000000e-01
The Max Ulp was: 1203127951
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 4.60023875348269939422607421875000e-04
The Average Ulp was: 264036256.000000
The Max error was: 3.94126325845718383789062500000000e-01
The Max Ulp was: 1203127951
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
Exponent E results:
```

```
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
LogE results:
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 2.60864718715225762935006059706211e-10
The Average Ulp was: 0.003200
The Max error was: 1.17985521796981629449874162673950e-07
The Max Ulp was: 1
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
```

```
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 2.60864718715225762935006059706211e-10
The Average Ulp was: 0.003200
The Max error was: 1.17985521796981629449874162673950e-07
The Max Ulp was: 1
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
Log10 results:
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 5.63924129615145375282736495137215e-10
The Average Ulp was: 0.006800
The Max error was: 1.19060288739092356991022825241089e-07
The Max Ulp was: 1
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
The Ave error was: 5.63924129615145375282736495137215e-10
The Average Ulp was: 0.006800
The Max error was: 1.19060288739092356991022825241089e-07
The Max Ulp was: 1
-------------------------
The Ave error was: 0.00000000000000000000000000000000e+00
The Average Ulp was: 0.000000
The Max error was: 0.00000000000000000000000000000000e+00
The Max Ulp was: 0
-------------------------
```

Appendix B: Impulse C CoDeveloper Projects – C code

B.1: Matrix Multiplication (Provided by Impulse Accelerated Technologies)

1. mmult.h

```
// Matrix multiply requires A_COLS == B_ROWS
// Unrolling requires A_ROWS == multiple of N_UNROLLED
// Address generation requires A_COLS, N_UNROLLED == power of 2
// Memory partitioning requires A_ROWS, B_COLS == multiple of PARTITION_SIZE
// Memory partitioning/unrolling requires N_UNROLLED <= PARTITION_SIZE

#define A_ROWS 512
#define A_COLS 128
#define B_ROWS 128
#define B_COLS 512
#define N_UNROLLED 16
#define NUMBER double
```

2. mmult_sw.c

```
/****

ExtremeData Matrix Multiply (sgemmnn11) Testbench

02/09/2007 - Scott Thibault

****/

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <time.h>
#include "co.h"
#include "mmult.h"

#if defined(IMPULSE_C_TARGET)
#define XD1000
#endif

#define N_ITERATIONS 70


extern co_architecture co_initialize(void *);


void clear_matrix (int m, int n, NUMBER *x, int lda)
{
  int i, j;

  for (j = 0; j < n; j++) {
    for (i = 0; i < m; i++) {
      x[j + i * lda] = 0.0;
    }
  }
  return;
```

```
}

void init_matrix (int m, int n, NUMBER *x, int lda)
{
  static double seed;
  int i, j;
  extern double msrand();

  seed = 2.3;
  for (j = 0; j < n; j++) {
    for (i = 0; i < m; i++) {
      x[j + i * lda] = msrand(&seed);
    }
  }
  return;
}

double msrand( double *seed )
{
   static double a = 16807.;
   static double m = 2147483647.;
   double ret_val;
   static double temp;

   temp = a * *seed;
   *seed = temp - m * (float) ((int) (temp / m));
   ret_val = *seed / m;

   return ret_val;
}

void gemm_ref(int *m, int *n, int *k, NUMBER *a, int *lda,
  NUMBER *b, int *ldb, NUMBER *d, int *ldc)
{
  NUMBER sum;
  int i, j, l;
  int a_dim, b_dim, c_dim;
  a_dim = *lda;
  b_dim = *ldb;
  c_dim = *ldc;

/* Form  D = D + A*B */

  for (j = 0; j < *n; j++) {
    for (i = 0; i < *m; i++) {
      sum = 0.0;
      for (l = 0; l < *k; l++) {
        sum += a[l + i * a_dim] * b[j + l * b_dim];
      }
      d[j + i * c_dim] += sum;
    }
  }
  return;
}

int errorcheck(int *m, int *n, NUMBER *c, int *ldc, NUMBER *d, float test_threshold)
```

```
{
  int i, j;
  int c_dim;
  float relerror;

  c_dim = *ldc;
  for (j=0;j<*n;j++) {
    for (i=0;i<*m;i++) {
      relerror = fabs(c[j+i*c_dim] - d[j+i*c_dim]) / fabs(d[j+i*c_dim]);
      if (relerror > test_threshold) {
        printf("error at point %d %d \n",i,j);
        printf("check threshold \n");
        printf("c ref %f    c calc %f \n",d[j+i*c_dim],c[j+i*c_dim]);
        fflush(0);
        return(-1);
      }
    }
  }
  return(0);
}

// NOTE: A,B,C stored by columns to match sgemmnn11 implementation

static NUMBER d[A_ROWS*B_COLS];

#ifdef XD1000
// Link with -lrt for clock_gettime
struct timespec curr_time()
{
            struct timespec ts;
            clock_gettime(CLOCK_REALTIME, &ts);
            return ts;
}
#define TIME_MINUS(a,b) (((a).tv_sec-(b).tv_sec)+(double)((a).tv_nsec-(b).tv_nsec)/1000000000.0)

#endif

void call_fpga(co_memory imgmem, co_signal start, co_signal end)
{
   uint32 i,j,data;
            int m,k,n,lda,ldb,ldc,ierr;
            NUMBER *chunk,*a,*b,*c;
            float test_threshold = 1.0e-5;
#ifdef XD1000
            struct timespec now, then;
#endif

   a = (NUMBER *) malloc((A_ROWS*A_COLS)*sizeof(NUMBER));
   b = (NUMBER *) malloc((B_ROWS*B_COLS)*sizeof(NUMBER));
   c = (NUMBER *) malloc((A_ROWS*B_COLS)*sizeof(NUMBER));

            init_matrix(A_ROWS, A_COLS, a, A_COLS);
            init_matrix(B_ROWS, B_COLS, b, B_COLS);
            init_matrix(A_ROWS, B_COLS, c, B_COLS);
  for (j = 0; j < B_COLS; j++) {
    for (i = 0; i < A_ROWS; i++) {
```

```
            d[j + i * B_COLS] = c[j + i * B_COLS];
          }
        }

                printf("Size %dx%dx%d\r\n\r\n",A_ROWS,A_COLS,B_COLS);

        printf("Running reference ...\r\n");

    #ifdef XD1000
                now = curr_time();
    #endif
    for (i=0; i<N_ITERATIONS; i++) {
                m=A_ROWS;
                k=B_ROWS;
                n=B_COLS;
                lda=A_COLS;
                ldb=B_COLS;
                ldc=B_COLS;
                gemm_ref(&m,&n,&k,a,&lda,b,&ldb,d,&ldc);
    }
    #ifdef XD1000
                then = curr_time();
                printf("time: %fs\n", (float)TIME_MINUS(then, now));
    #endif
                printf("%f %f %f %f/",(float)d[0],(float)d[1],(float)d[2],(float)d[3]);
                printf("%f%f%f%f\r\n",(float)d[N_UNROLLED],(float)d[N_UNROLLED+1],(float)d[N_UNRO
LLED+2],(float)d[N_UNROLLED+3]);
                printf("%f %f
%f%f/",(float)d[B_COLS],(float)d[B_COLS+1],(float)d[B_COLS+2],(float)d[B_COLS+3]);
                printf("%f%f%f%f\r\n",(float)d[B_COLS+N_UNROLLED],(float)d[B_COLS+N_UNROLLED+
1],(float)d[B_COLS+N_UNROLLED+2],(float)d[B_COLS+N_UNROLLED+3]);

        printf("Running test ...\r\n");

                co_memory_writeblock(imgmem, 0, a, (A_ROWS*A_COLS)*sizeof(NUMBER));
                co_memory_writeblock(imgmem, (A_ROWS*A_COLS)*sizeof(NUMBER), b,
(B_ROWS*B_COLS)*sizeof(NUMBER));
                co_memory_writeblock(imgmem,
((A_ROWS*A_COLS)+(B_ROWS*B_COLS))*sizeof(NUMBER), c,
(A_ROWS*B_COLS)*sizeof(NUMBER));
    #ifdef XD1000
                now = curr_time();
    #endif
    for (i=0; i<N_ITERATIONS; i++) {
                co_signal_post(start, 0);
                co_signal_wait(end, (co_int32*)&data);
                }
    #ifdef XD1000
                then = curr_time();
                printf("time: %fs\n", (float)TIME_MINUS(then, now));
    #endif
    printf("return %d\n",data);
                co_memory_readblock(imgmem,
((A_ROWS*A_COLS)+(B_ROWS*B_COLS))*sizeof(NUMBER),
                        c, A_ROWS*B_COLS*sizeof(NUMBER));
```

```
            printf("%f %f %f %f/",(float)c[0],(float)c[1],(float)c[2],(float)c[3]);
            printf("%f %f %f
%f\r\n",(float)c[N_UNROLLED],(float)c[N_UNROLLED+1],(float)c[N_UNROLLED+2],(float)c[N_UNROL
LED+3]);
            printf("%f %f %f
%f/",(float)c[B_COLS],(float)c[B_COLS+1],(float)c[B_COLS+2],(float)c[B_COLS+3]);
            printf("%f %f %f
%f\r\n",(float)c[B_COLS+N_UNROLLED],(float)c[B_COLS+N_UNROLLED+1],(float)c[B_COLS+N_UNR
OLLED+2],(float)c[B_COLS+N_UNROLLED+3]);

        printf("Checking result ...\r\n");

       ierr = errorcheck(&m, &n, c, &ldc, d, test_threshold);
       if (ierr==0) printf("Passed.\r\n");

    }


    int main(int argc, char *argv[])
    {
       IF_SIM(int c;)
       co_architecture my_arch;

       printf("Matrix Multiply\r\n");
       printf("----------------\n");

       my_arch = co_initialize(NULL);
       co_execute(my_arch);

       IF_SIM(printf("Press Enter key to continue...\n");)
       IF_SIM(c = getc(stdin);)

       return(0);
    }
```

3. mmult_hw.c

```
/****

ExtremeData Matrix Multiply (dgemm) FPGA Implementation

02/09/2007 - Scott Thibault

****/

#include <stdio.h>
#include "co.h"
#include "mmult.h"


// Define a wide element type: bits = N_UNROLLED * bits(NUMBER)
#define WIDE co_uint1024
typedef uint64 WIDE[N_UNROLLED];


#ifdef IMPULSE_C_SYNTHESIS
```

```
#undef co_bit_insert
#define GETELM(x,i) to_double((x)>>(960-64*(i)))
#define PUTELM(x,i,e) x=co_bit_insert(x,960-64*(i),64,double_bits(e))
#define ASSIGN(lhs,rhs) lhs=rhs

#else

#define GETELM(x,i) to_double(x[i])
#define PUTELM(x,i,e) x[i]=double_bits(e)
#define ASSIGN(lhs,rhs) memcpy(lhs,rhs,N_UNROLLED*sizeof(NUMBER))

#endif

extern void call_fpga(co_memory datamem, co_signal start, co_signal end);

#ifdef IMPULSE_C_SYNTHESIS
#define DECLMEM
#else
#define DECLMEM static
#endif

// NOTE: A, B, and C stored by rows

#define C_NUM (B_COLS/N_UNROLLED)

static uint32 nRead,nDone;
static WIDE Cin[2*C_NUM],Cout[2*C_NUM];
static NUMBER Ain[2*A_COLS];

void ioproc(co_memory datamem, co_signal start, co_signal complete)

{
 uint32 j;
 uint32 offAin,offCin,offCout,cHalf,aHalf;
 int32 d;

do {
   offAin=0;
        offCin=(A_ROWS * A_COLS * sizeof(NUMBER))+(B_ROWS * B_COLS *
sizeof(NUMBER));
        offCout=offCin;
        nRead=0;

        co_signal_wait(start,&d);
        co_memory_readblock(datamem,offAin,Ain,2 * A_COLS * sizeof(NUMBER));
        offAin+=2 * A_COLS * sizeof(NUMBER);
        co_memory_readblock(datamem,offCin,Cin,2 * B_COLS * sizeof(NUMBER));
//      printf("in %f\n",GETELM(Cin[0],0));
//      printf("in %f\n",GETELM(Cin[C_NUM],0));
        offCin+=2 * B_COLS * sizeof(NUMBER);
        nRead=2;
        j=0;
        do {
                while (nDone<=j);
                if ((j&1)==0) {
```

```
                        aHalf=cHalf=0;
                    } else {
                        aHalf=A_COLS;
                        cHalf=C_NUM;
                    }
    //              printf("%d: out %f\n",(offCout-
base)/(B_COLS*sizeof(NUMBER)),(float)GETELM(Cout[half],0));
                    co_memory_writeblock(datamem,offCout,&Cout[cHalf],B_COLS * sizeof(NUMBER));
                    offCout+=B_COLS * sizeof(NUMBER);
                    j++;
                    if (j==A_ROWS) break;
                    co_memory_readblock(datamem,offAin,&Ain[aHalf],A_COLS * sizeof(NUMBER));
                    offAin+=A_COLS * sizeof(NUMBER);
                    co_memory_readblock(datamem,offCin,&Cin[cHalf],B_COLS * sizeof(NUMBER));
    //              printf("%d: in %f\n",(offCin-
base)/(B_COLS*sizeof(NUMBER)),(float)GETELM(Cin[half],0));
                    offCin+=B_COLS * sizeof(NUMBER);
                    nRead++;
            } while (1);
        co_signal_post(complete,j);

            IF_SIM(break;)
    } while (1);
    }


    void mmproc(co_memory datamem, co_signal go, co_signal startio)
    {
        uint32 i, j, k, ki, ci, bi, aHalf, cHalf;
        uint32 offA,offB,offCin,offCout,data;
            NUMBER m;
        DECLMEM WIDE B[B_ROWS*B_COLS/N_UNROLLED];
            WIDE bval;
            WIDE cval,cres;
            DECLMEM WIDE ctmp[C_NUM];

    //      co_array_config(B,co_kind,"dualsync");
    do {
        offA=0;
            offB=offA+(A_ROWS * A_COLS * sizeof(NUMBER));
    //      offCin=offB+(B_ROWS * B_COLS * sizeof(NUMBER));
    //      offCout=offCin;


        co_signal_wait(go,(int32*)&data);


            // Compute C := C + A*B
            //

    //      co_memory_readblock(datamem, offA, A, A_ROWS * A_COLS * sizeof(NUMBER));
            co_memory_readblock(datamem, offB, B, B_ROWS * B_COLS * sizeof(NUMBER));
            co_signal_post(startio,data);
            nDone=0;

            for (j=0; j < A_ROWS; j++) {
```

```
                while (nRead<=j);
                // Read one row of C
//              co_memory_readblock(datamem, offCin, C, B_COLS * sizeof(NUMBER));
//              offCin+=B_COLS * sizeof(NUMBER);
                if ((j&1)==0) {
                        aHalf=cHalf=0;
                } else {
                        aHalf=A_COLS;
                        cHalf=C_NUM;
                }
                ki=0;
                do {
#pragma co pipeline
#pragma co nonrecursive ctmp
                        k=ki/B_COLS;
                        i=ki%B_COLS;
                        m=Ain[aHalf+k];
                        ci=i/N_UNROLLED;
//                      if ((j&1)==0)
//                              ci=i/N_UNROLLED;
//                      else
//                              ci=(B_COLS+i)/N_UNROLLED;
                        bi=ki/N_UNROLLED;
                        ASSIGN(bval,B[bi]);
                        if (k==0)
                                ASSIGN(cval,Cin[cHalf+ci]);
                        else
                                ASSIGN(cval,ctmp[ci]);
//if ((j==1)&&(ci==0)) printf("in %f\n",(float)*(double *)(cval+0));
                        IF_NSIM(cres=0;)
                        PUTELM(cres,0,GETELM(cval,0)+m*GETELM(bval,0));
                        PUTELM(cres,1,GETELM(cval,1)+m*GETELM(bval,1));
                        PUTELM(cres,2,GETELM(cval,2)+m*GETELM(bval,2));
                        PUTELM(cres,3,GETELM(cval,3)+m*GETELM(bval,3));
                        PUTELM(cres,4,GETELM(cval,4)+m*GETELM(bval,4));
                        PUTELM(cres,5,GETELM(cval,5)+m*GETELM(bval,5));
                        PUTELM(cres,6,GETELM(cval,6)+m*GETELM(bval,6));
                        PUTELM(cres,7,GETELM(cval,7)+m*GETELM(bval,7));
                        PUTELM(cres,8,GETELM(cval,8)+m*GETELM(bval,8));
                        PUTELM(cres,9,GETELM(cval,9)+m*GETELM(bval,9));
                        PUTELM(cres,10,GETELM(cval,10)+m*GETELM(bval,10));
                        PUTELM(cres,11,GETELM(cval,11)+m*GETELM(bval,11));
                        PUTELM(cres,12,GETELM(cval,12)+m*GETELM(bval,12));
                        PUTELM(cres,13,GETELM(cval,13)+m*GETELM(bval,13));
                        PUTELM(cres,14,GETELM(cval,14)+m*GETELM(bval,14));
                        PUTELM(cres,15,GETELM(cval,15)+m*GETELM(bval,15));
/*
                        PUTELM(cres,16,GETELM(cval,16)+m*GETELM(bval,16));
                        PUTELM(cres,17,GETELM(cval,17)+m*GETELM(bval,17));
                        PUTELM(cres,18,GETELM(cval,18)+m*GETELM(bval,18));
                        PUTELM(cres,19,GETELM(cval,19)+m*GETELM(bval,19));
                        PUTELM(cres,20,GETELM(cval,20)+m*GETELM(bval,20));
                        PUTELM(cres,21,GETELM(cval,21)+m*GETELM(bval,21));
                        PUTELM(cres,22,GETELM(cval,22)+m*GETELM(bval,22));
                        PUTELM(cres,23,GETELM(cval,23)+m*GETELM(bval,23));
                        PUTELM(cres,24,GETELM(cval,24)+m*GETELM(bval,24));
```

```
                                PUTELM(cres,25,GETELM(cval,25)+m*GETELM(bval,25));
                                PUTELM(cres,26,GETELM(cval,26)+m*GETELM(bval,26));
                                PUTELM(cres,27,GETELM(cval,27)+m*GETELM(bval,27));
                                PUTELM(cres,28,GETELM(cval,28)+m*GETELM(bval,28));
                                PUTELM(cres,29,GETELM(cval,29)+m*GETELM(bval,29));
                                PUTELM(cres,30,GETELM(cval,30)+m*GETELM(bval,30));
                                PUTELM(cres,31,GETELM(cval,31)+m*GETELM(bval,31));
        */
                                ASSIGN(ctmp[ci],cres);
                                ASSIGN(Cout[cHalf+ci],cres);
        //if ((j==1)&&(ci==0)) printf("out %f\n",(float)*(double *)(cres+0));
                                ki+=N_UNROLLED;
                        } while (ki < (B_ROWS*B_COLS));
        //                      co_memory_writeblock(datamem, offCout, C, B_COLS * sizeof(NUMBER));
        //                      offCout+=B_COLS * sizeof(NUMBER);
                        nDone++;
                } // end j

                IF_SIM(break;)
        } while (1);

        }

        void config_mmult(void *arg)
        {
           co_signal startsig, donesig, iosig;
           co_memory shrmem;
           co_process cpu_proc, mm, io_proc;

           startsig = co_signal_create("start");
           donesig  = co_signal_create("done");
           iosig  = co_signal_create("iosig");
           shrmem   = co_memory_create("data", "",
    ((A_ROWS*A_COLS)+(B_ROWS*B_COLS)+(A_ROWS*B_COLS))*sizeof(NUMBER));

           cpu_proc = co_process_create("cpu_proc",(co_function)call_fpga,3,shrmem,startsig,donesig);
           mm = co_process_create("mm",(co_function)mmproc,3,shrmem,startsig,iosig);
           io_proc = co_process_create("io_proc",(co_function)ioproc,3,shrmem,iosig,donesig);

           co_process_config(mm, co_loc, "PE0");
           co_process_config(io_proc, co_loc, "PE0");
        }

        co_architecture co_initialize()
        {
           return(co_architecture_create("mmex", "fpga", config_mmult, NULL));
        }
```

B.2: Addition Benchmark Version 1

1. HeadAddition.h

```
// This file provides the number of operations needed to test.
// This benchtest will deal with double floating point numbers.
// A_NUM = B_NUM = 2^17
```

```
    // Variable mem_trans define the amount of memory is transfer for each memory transfer command in
hardware.
    // memory (bits) = mem_trans (# of doble precision number) * 64 (64 bits in 1 double precision number).
    // Better explantion for this variable is in Hardware code.

    #define A_NUM 131072
    #define B_NUM 131072
    // #define A_NUM 65536
    // #define B_NUM 65536
    //#define MEM_TRANS 2048
    #define MEM_TRANS 256    // 2^8 numbers at a time
    #define UN_ROLLED 16
    // This variable determine how many times the arrays will be recalculated
    #define N_ITERATION 1000
```

2. Addition_sw.c

```
/****

XtremeData Addition Testbench
Author: John Rothermel + Tri Dang
Version 1
Date: Nov 26, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C.

****/

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <time.h>
#include "co.h"
#include "cosim_log.h"
#include "HeadAddition.h"

// This variable determine how many times the arrays will be recalculated
//#define N_ITERATION 1

// If compiling for the target platform
#if defined(IMPULSE_C_TARGET)
#define XD1000
#endif

extern co_architecture co_initialize(void *);

//static double D[A_NUM];

// Create Random numbers
double msrand( double *seed )
{
   static double a = 16807.;
   static double m = 2147483647.;
   double ret_val;
   static double temp;

   temp = a * *seed;
   *seed = temp - m * (float) ((int) (temp / m));
```

```
    ret_val = *seed / m;

    return ret_val;
}



// Initalize Array
void init_array(double *A, double seed)
{
  int j;
  int exp;
  //static double seed;
  extern double msrand();
  //seed = 2.3;
  for (j = 0; j < A_NUM; j++) {
       A[j] = msrand(&seed);
       exp = (int) 10 * msrand(&seed);  // use msrand to get a random float (between 0 and 1), multiply by 10,
then truncate to int
       A[j] *= pow(-1,exp);  // exponentiate -1 to random exp
  }
  return;
}



// Clear Array
void clear_array(double *A)
{
  int j;
  for (j = 0; j < A_NUM; j++){
     A[j] = 0.0;
  }
}


/* Reference implementation of xGEMM
 * Runs on the Opteron CPU for comparison with the FPGA implementation
 */
void get_ref(double *A, double *B, double *D){
    double sum;
    int i;
    for (i = 0; i < A_NUM; i++){
      sum = 0;
      sum = A[i] + B[i];
      D[i] = sum;
    }
    //printf("\n the first result is %f\n", D[0]);
}



/* Verify that two results are equal (difference in every element <=
 * test_threshold)
 */
int errorcheck(double *C, double *D, float test_threshold)
```

```
{
  float relerror;
  int j;
  for ( j= A_NUM; j > 0; j--) {
       relerror = fabs(C[j] - D[j]) / fabs(D[j]);
       if (relerror > test_threshold) {
          printf("error at point %d \n", j);
          printf("check threshold \n");
          printf("CPU ref %f    FPGA calc %f \n",D[j] , C[j]);
                        fflush(0);
          return(-1);
       }
  }
  return(0);
}



#ifdef XD1000
/* Return current time from an OS timer.
 * Link with -lrt for clock_gettime (you must modify the generated Makefile)
 */
struct timespec curr_time()
{
      struct timespec ts;
      clock_gettime(CLOCK_REALTIME, &ts);
      return ts;
}
#define TIME_MINUS(a,b) (((a).tv_sec-(b).tv_sec)+(double)((a).tv_nsec-(b).tv_nsec)/1000000000.0)
#endif



/* Software process, runs on Opteron
 *
 * Initializes memory with array values, then runs the reference software and
 * FPGA-accelerated hardware versions.  Compares the output of both
 * implementations and prints timing results.
 */
void call_fpga(co_memory imgmem, co_signal start, co_signal end)
{
   uint32 data;
   int j, k;
      int ierr;
      double *A,*B,*C, *D;                  // A, B, C, D are the arrays that will be used.
      float test_threshold = 1.0e-10;
      float time_measured_fpga, time_measured_cpu;

#ifdef XD1000
      struct timespec now, then;
#endif
      IF_SIM(cosim_logwindow log = cosim_logwindow_create("call_fpga");)

   //Create arrays
   A = (double *) malloc((A_NUM)*sizeof(double));   // Operand array A
   if(A==NULL) printf("ERROR!\n");                  // Test if memory is properly allocated
```

```
    B = (double *) malloc((B_NUM)*sizeof(double));   // Operand array B
    if(B==NULL) printf("ERROR!\n");
    C = (double *) malloc((A_NUM)*sizeof(double));   // Result array C from FPGA calculation
    if(C==NULL) printf("ERROR!\n");
    D = (double *) malloc((A_NUM)*sizeof(double));   // Result array D from CPU calculation
    if(D==NULL) printf("ERROR!\n");

    // Initalize operands array
      init_array(A, 2.3);
      init_array(B, 3.2);
    clear_array(C);
    clear_array(D);

    printf("the first few values of array A are %f  %f  %f  %f\n", A[0], A[1], A[A_NUM-2], A[A_NUM-1]);
    printf("\nthe first few values of array B are %f  %f  %f  %f\n", B[0], B[1], B[A_NUM-2], B[A_NUM-1]);

#ifndef XD1000
      printf("Timing statistics only calculated on XD1000 system\n");
#endif

    printf("Running reference (CPU)...\r\n");

#ifdef XD1000
      now = curr_time();
#endif

    // Get reference values from the CPU - do N_ITERATIONS times
      for ( j = 0 ; j < N_ITERATION ; j++)
      get_ref(A, B, D);

#ifdef XD1000
      then = curr_time();
      time_measured_cpu = (float)TIME_MINUS(then, now);
      printf("time: %fs\n", time_measured_cpu);
#endif
    // Print out a few examples
      printf("%f %f %f %f\n", (float)D[0],(float)D[1],(float)D[2],(float)D[3]);
      printf("%f %f %f %f\r\n", (float)D[4],(float)D[5],(float)D[6],(float)D[7]);
      printf("%f %f %f %f\n", (float)D[A_NUM-8], (float)D[A_NUM-7], (float)D[A_NUM-6],
(float)D[A_NUM-5]);
      printf("%f %f %f %f\r\n",(float)D[A_NUM-4], (float)D[A_NUM-3], (float)D[A_NUM-2],
(float)D[A_NUM-1]);

    printf("Running test (FPGA)...\r\n");

      // Write the A, B, and C matrices to the SRAM
      // The FPGA will wait for a signal, then read/write matrices in the SRAM
      // while computing the product.
      co_memory_writeblock(imgmem, 0, A, (A_NUM)*sizeof(double));
      co_memory_writeblock(imgmem, (A_NUM)*sizeof(double), B, (B_NUM)*sizeof(double));
      co_memory_writeblock(imgmem, ((A_NUM)+(B_NUM))*sizeof(double), C, (A_NUM)*sizeof(double));


#ifdef XD1000
      now = curr_time();
#endif
```

```
    for ( k = 0; k < N_ITERATION; k++){
      // Signal the FPGA to start, then wait for the "done" signal
            co_signal_post(start, 0);
            // FPGA computes result...
            co_signal_wait(end, (co_int32*)&data);
    }

#ifdef XD1000
      then = curr_time();
      time_measured_fpga = (float)TIME_MINUS(then, now);
      printf("time: %fs\n", time_measured_fpga);
      printf("speedup: %fx performance of CPU\n\n", time_measured_cpu/time_measured_fpga);
#endif


      // Read the FPGA-computed result out of SRAM into 'c' for verification
      co_memory_readblock(imgmem, ((A_NUM)+(B_NUM))*sizeof(double), C, A_NUM*sizeof(double));

    // Print out a few examples
      printf("%f %f %f %f\n", (float)C[0],(float)C[1],(float)C[2],(float)C[3]);
      printf("%f %f %f %f\r\n", (float)C[4],(float)C[5],(float)C[6],(float)C[7]);
     printf("%f %f %f %f\r\n", (float)C[128],(float)C[129],(float)C[130],(float)C[131]);
     printf("%f %f %f %f\r\n", (float)C[252],(float)C[253],(float)C[254],(float)C[255]);
      printf("%f %f %f %f\n", (float)C[A_NUM-8], (float)C[A_NUM-7], (float)C[A_NUM-6],
(float)C[A_NUM-5]);
      printf("%f %f %f %f\r\n",(float)C[A_NUM-4], (float)C[A_NUM-3], (float)C[A_NUM-2],
(float)C[A_NUM-1]);

    printf("Checking result ...\r\n");
      ierr = errorcheck(C, D, test_threshold);
      if (ierr==0) printf("Passed.\r\n");
}


int main(int argc, char *argv[])
{
      IF_SIM(int c;)
      co_architecture my_arch;

      printf("Addition Test\r\n");
      printf("----------------\n");

      // Get list of processes to execute from the configuration function in
      // Addition_hw.c (desktop simulation) or from co_init.c (host software).
      my_arch = co_initialize(NULL);
      // Execute processes
      co_execute(my_arch);

      IF_SIM(printf("Press Enter key to continue...\n");)
      IF_SIM(c = getc(stdin);)

      return(0);
}
```

3. Addition_hw.c

```
/****

XtremeData Addition Testbench
Authors: John Rothermel + Tri Dang
Version 1
Date: Nov 26, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C.

****/

#include <stdio.h>
#include "co.h"
#include "co_math.h"
#include "cosim_log.h"
#include "HeadAddition.h"

// Software process, defined in mmult_sw.c
extern void call_fpga(co_memory datamem, co_signal start, co_signal end);

// Define a wide element type: bits = N_UNROLLED * bits(NUMBER)
// Evidently, this element type can store 16 double precision number.
#define WIDE co_uint1024
typedef uint64 WIDE[UN_ROLLED];

// Define macros for moving individual floating-point numbers in/out of the
// WIDE type.  Different code used in hardware generation (#ifdef
// IMPULSE_C_SYNTHESIS) and in simulation.
//
// These macros are written for double-precision numbers and must be modified
// to do SGEMM.

#ifdef IMPULSE_C_SYNTHESIS

#undef co_bit_insert
#define GETELM(x,i) to_double((x)>>(960-64*(i)))
#define PUTELM(x,i,e) x=co_bit_insert(x,960-64*(i),64,double_bits(e))
#define ASSIGN(lhs,rhs) lhs=rhs

#else

#define GETELM(x,i) to_double(x[i])
#define PUTELM(x,i,e) x[i]=double_bits(e)
#define ASSIGN(lhs,rhs) memcpy(lhs,rhs,UN_ROLLED*sizeof(double))

#endif

#ifdef IMPULSE_C_SYNTHESIS
#define DECLMEM
#else
#define DECLMEM static
#endif

 // SLOT is the constant that define how many element required for array Ain
#define SLOT MEM_TRANS/UN_ROLLED
// CHUNK is the number of chunks required to process to complete the testbench
```

```
#define CHUNK A_NUM/MEM_TRANS
// Shared memories (FPGA block RAMs) used by ioproc and mmproc to store
// array A and C.
static WIDE Ain[2*SLOT], Cout[2*SLOT];
static WIDE Bin[2*SLOT];

// # of chunks of A read. A Chunk is defined as one piece of memory that is transfer to
// the FPGA for calculation at one time, which is represented as variable MEM_TRANS.
// In specific case where we have 2^20 numbers and trade in 2048 numbers at a time, 512 chunks
// is required to complete the calculation.
static uint32 nRead;
// # of chunks of A processed so far by mmproc.  This variable is shared
// by ioproc and mmproc to synchronize memory access.
static uint32 nDone;

/* Read inputs A and store output C in parallel with the
 * computation done by mmproc.  For example, while mmproc is computing
 * A[1] + B[1], ioproc is reading A[2] and B[2] and storing
 * the results from A[0] + B[0] back into SRAM.
 */
void ioproc(co_memory datamem, co_signal start, co_signal complete)
{
        uint32 j;
        uint32 offA, offB, offC;
   uint32 cHalf, aHalf;          // cHalf and aHalf is used to keep track of which chunk will be used
        int32 data;
   IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop
        IF_SIM(cosim_logwindow log = cosim_logwindow_create("ioproc");)

        do {  // One matrix multiplication per iteration
                offA = 0;
      offB = (A_NUM) * sizeof(double);
                offC = (A_NUM + B_NUM) * sizeof(double);
      nRead = 0;                        // nRead is also need to be reset so that mmprocess and ioprocess are at
the same index

                // Wait for mmproc to read matrix B
                co_signal_wait(start,&data);


                // Preload two chunks of A
                co_memory_readblock(datamem, offA, Ain, 2 * SLOT * sizeof(WIDE));
                offA += 2 * SLOT * sizeof(WIDE);
                co_memory_readblock(datamem, offB, Bin, 2 * SLOT * sizeof(WIDE));
      offB += 2 * SLOT * sizeof(WIDE);
      nRead = 2;

                j=0;
                do {
                        while (nDone<=j);  // Wait for mmproc to compute new results
          if ((j&1)==0) {   // if j is even, take the first chunk that is stored in the array
            aHalf = cHalf = 0;
          } else {          // if j is odd, take the second chunk
            aHalf = SLOT;
            cHalf = SLOT;
          }
```

```
                                // Store a chunk of results back to C in SRAM
                                co_memory_writeblock(datamem, offC, &Cout[cHalf] , SLOT *
sizeof(WIDE));
                                offC += SLOT * sizeof(WIDE);                // Move to the next chunk address
                                j++;
                                if (j == CHUNK) break;  // Done with entire computation
                                // Read another row each of A and C for the next computation
                                co_memory_readblock(datamem, offA, &Ain[aHalf], SLOT * sizeof(WIDE));
                                offA += SLOT * sizeof(WIDE);
            co_memory_readblock(datamem, offB, &Bin[aHalf], SLOT * sizeof(WIDE));
            offB += SLOT * sizeof(WIDE);
                                nRead++;

                        } while (1);
                        co_signal_post(complete,j);  // Signal CPU that computation is done

                        //IF_SIM(break;)
                        IF_SIM(if(!(--loopcnt)) break;)
                } while (1);  // Always running in hardware
        }


        /* Computes C = A + B.  Access to array A in shared block RAMs is
         * synchronized with ioproc by global variables.  Matrix B is stored entirely
         * in block RAM local to this process.
         */
        void mmproc(co_memory datamem, co_signal go, co_signal startio)
        {
           uint32 offA, offB, data;
           uint32 Half ;        //Again to keep track of which chunk will be used
                double m;
           int j;               // keep track of the chunks
           int k;               // A's index
           //DECLMEM WIDE Bin[B_NUM/UN_ROLLED];  //original
                WIDE aval, bval;     // buffer for one element of array A and B
                WIDE cres;          // buffer for one result element
           IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop

                //DECLMEM WIDE ctmp[C_NUM];
                IF_SIM(cosim_logwindow log = cosim_logwindow_create("mmproc");)

                // Array B is implemented by default in an Altera FPGA as a dual-port
                // synchronous RAM.
                //co_array_config(B,co_kind,"dualsync");

                do {  // One Addition per iteration
                        offA = 0;
                        offB = offA+(A_NUM * sizeof(double));


                        // Wait for the CPU to store A and C in SRAM
                        co_signal_wait(go,(int32*)&data);

                        // Compute C := A+B
                        //
                        // Read all of B from SRAM into local memory
```

```
                        //co_memory_readblock(datamem, offB, Bin, B_NUM * sizeof(double));

                        /////
                        // EAT (Edward Trexel) 11/30/2007 --
                        // Add co_par_break() to force co_signal_wait() and co_signal_post() into separate
                        //  stages.  Sometimes, even if the signals are uinrelated and won't cause a deadlock
                        //  situation with another process, the co_signal_post() won't occur as expected.
                        //  Previously in the DGEMM code, the co_memory_readblock() was creating this
separation.
                        co_par_break();
                        /////

                        // Tell ioproc it can start its overlapping IO to/from A and C
                        co_signal_post(startio,data);
            nDone= 0;               //Reset nDone

                        for (j=0; j < CHUNK; j++) {       //Chunk = 512
                                while (nRead <= j);  // Wait for data to be made available by ioproc
              if ((j&1) == 0){     // First chunk is selected when j is even
                Half = 0;
              } else {            // Second chunk is selected when j is odd
                Half = SLOT;
              }
              k = 0;              // k keep track of index in one chunk
                                do {
                #pragma co pipeline
                                        //IF_NSIM(cres=0;)
                  ASSIGN(aval, Ain[k + Half]);
                  //ASSIGN(bval, Bin[p]);
                  ASSIGN(bval, Bin[k + Half]);   // read in B into bval
                                        // Add UN_ROLLED floating-point values.
                                        // The Impulse C compiler will schedule all these operations so
                                        // they execute in a single clock cycle.
                                        PUTELM(cres,0,GETELM(aval,0)  +  GETELM(bval,0));
                                        PUTELM(cres,1,GETELM(aval,1)  +  GETELM(bval,1));
                                        PUTELM(cres,2,GETELM(aval,2)  +  GETELM(bval,2));
                                        PUTELM(cres,3,GETELM(aval,3)  +  GETELM(bval,3));
                                        PUTELM(cres,4,GETELM(aval,4)  +  GETELM(bval,4));
                                        PUTELM(cres,5,GETELM(aval,5)  +  GETELM(bval,5));
                                        PUTELM(cres,6,GETELM(aval,6)  +  GETELM(bval,6));
                                        PUTELM(cres,7,GETELM(aval,7)  +  GETELM(bval,7));
                                        PUTELM(cres,8,GETELM(aval,8)  +  GETELM(bval,8));
                                        PUTELM(cres,9,GETELM(aval,9)  +  GETELM(bval,9));
                                        PUTELM(cres,10,GETELM(aval,10)+  GETELM(bval,10));
                                        PUTELM(cres,11,GETELM(aval,11)+  GETELM(bval,11));
                                        PUTELM(cres,12,GETELM(aval,12)+  GETELM(bval,12));
                                        PUTELM(cres,13,GETELM(aval,13)+  GETELM(bval,13));
                                        PUTELM(cres,14,GETELM(aval,14)+  GETELM(bval,14));
                                        PUTELM(cres,15,GETELM(aval,15)+  GETELM(bval,15));

      /*
       * Commented out for UN_ROLLED == 16.  The FPGA has enough multiplier
       * resources to do 32 parallel double-precision MACs, but the Quartus II tools
       * cannot aggregate the various block RAMs on the device into a memory wide
       * enough to feed all 32 operators in parallel.  The memory would have to be:
       *   UN_ROLLED * sizeof(double) * 8 =
```

```
 *                  32 * 8 * 8 = 2048 bits wide
 * Unrolling by 32 may work for single-precision Addition (SGEMM)
 *
                                        PUTELM(cres,16,GETELM(aval,16)+GETELM(bval,16));
                                        PUTELM(cres,17,GETELM(aval,17)+GETELM(bval,17));
                                        PUTELM(cres,18,GETELM(aval,18)+GETELM(bval,18));
                                        PUTELM(cres,19,GETELM(aval,19)+GETELM(bval,19));
                                        PUTELM(cres,20,GETELM(aval,20)+GETELM(bval,20));
                                        PUTELM(cres,21,GETELM(aval,21)+GETELM(bval,21));
                                        PUTELM(cres,22,GETELM(aval,22)+GETELM(bval,22));
                                        PUTELM(cres,23,GETELM(aval,23)+GETELM(bval,23));
                                        PUTELM(cres,24,GETELM(aval,24)+GETELM(bval,24));
                                        PUTELM(cres,25,GETELM(aval,25)+GETELM(bval,25));
                                        PUTELM(cres,26,GETELM(aval,26)+GETELM(bval,26));
                                        PUTELM(cres,27,GETELM(aval,27)+GETELM(bval,27));
                                        PUTELM(cres,28,GETELM(aval,28)+GETELM(bval,28));
                                        PUTELM(cres,29,GETELM(aval,29)+GETELM(bval,29));
                                        PUTELM(cres,30,GETELM(aval,30)+GETELM(bval,30));
                                        PUTELM(cres,31,GETELM(aval,31)+GETELM(bval,31));
*/
                                        ASSIGN(Cout[k + Half],cres);
                                        k++;
                                } while (k < (SLOT));
                                nDone++;
                        } // end j
                        //IF_SIM(break;)
                        IF_SIM(if(!(--loopcnt)) break;)
                } while (1);  // Always running in hardware
}



// The following two functions implement a required Impulse C pattern
// Impulse C configuration function
// Describes all processes and the IO objects that connect them
void config_add(void *arg)
{
        co_signal startsig, donesig, iosig;
        co_memory shrmem;
        co_process cpu_proc, mm, io_proc;

        startsig = co_signal_create("start");
        donesig  = co_signal_create("done");
        iosig  = co_signal_create("iosig");
        // Associate the co_memory object with the default physical memory location
        // for the platform (second argument, "").  In the XD1000's case, this
        // default location is the QDR SRAM, where all matrices are stored.
        shrmem   = co_memory_create("data", "", ((A_NUM)+(B_NUM)+(A_NUM))*sizeof(double));

        cpu_proc = co_process_create("cpu_proc",(co_function)call_fpga, 3, shrmem, startsig, donesig);
        mm = co_process_create("mm",(co_function)mmproc,3,shrmem,startsig,iosig);
        io_proc = co_process_create("io_proc",(co_function)ioproc,3,shrmem,iosig,donesig);

        // Assign mmproc and ioproc to run as FPGA hardware
        co_process_config(mm, co_loc, "PE0");
        co_process_config(io_proc, co_loc, "PE0");
```

```
}

// Boilerplate code that tells the Impulse C hardware compiler to look at the
// config_add function for the application's "layout" of processes and IOs.
// In simulation, this code gives the simulation library a pointer to the
// configuration function so the library can execute it internally.
co_architecture co_initialize()
{
   return(co_architecture_create("mmex", "fpga", config_add, NULL));
}
```

B.3: Addition Benchmark Version 2 (one input in local memory)

1. HeadAddition.h

```
// This file provides the number of operations needed to test.
// This benchtest will deal with double floating point numbers.
// A_NUM = B_NUM = 2^16
// Variable mem_trans define the amount of memory is transfer for each memory transfer command in hardware.
// memory (bits) = mem_trans (# of doble precision number) * 64 (64 bits in 1 double precision number).
// Better explantion for this variable is in Hardware code.

#define A_NUM 65536
#define B_NUM 65536
// #define A_NUM 65536
// #define B_NUM 65536
#define MEM_TRANS 2048
#define UN_ROLLED 16
// This variable determine how many times the arrays will be recalculated
#define N_ITERATION 1000
```

2. Addition_sw.c

```
/****

XtremeData Addition Testbench
Author: John Rothermel + Tri Dang
Version 2 - One operand array in CPU memory one in local memory SRAM of FPGA
Date: Nov 26, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C.

****/

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <time.h>
#include "co.h"
#include "cosim_log.h"
#include "HeadAddition.h"

// This variable determine how many times the arrays will be recalculated
//#define N_ITERATION 1

// If compiling for the target platform
```

```
#if defined(IMPULSE_C_TARGET)
#define XD1000
#endif

extern co_architecture co_initialize(void *);

//static double D[A_NUM];

// Create Random numbers
double msrand( double *seed )
{
   static double a = 16807.;
   static double m = 2147483647.;
   double ret_val;
   static double temp;

   temp = a * *seed;
   *seed = temp - m * (float) ((int) (temp / m));
   ret_val = *seed / m;

   return ret_val;
}



// Initalize Array
void init_array(double *A, double seed)
{
  int j;
  int exp;
  //static double seed;
  extern double msrand();
  //seed = 2.3;
  for (j = 0; j < A_NUM; j++) {
       A[j] = msrand(&seed);
       exp = (int) 10 * msrand(&seed);  // use msrand to get a random float (between 0 and 1), multiply by
10, then truncate to int
       A[j] *= pow(-1,exp);  // exponentiate -1 to random exp
  }
  return;
}



// Clear Array
void clear_array(double *A)
{
  int j;
  for (j = 0; j < A_NUM; j++){
     A[j] = 0.0;
  }
}


/* Reference implementation of xGEMM
 * Runs on the Opteron CPU for comparison with the FPGA implementation
```

```
 */
void get_ref(double *A, double *B, double *D){
   double sum;
   int i;
   for (i = 0; i < A_NUM; i++){
     sum = 0;
     sum = A[i] + B[i];
     D[i] = sum;
   }
   //printf("\n the first result is %f\n", D[0]);
}



/* Verify that two results are equal (difference in every element <=
 * test_threshold)
 */
int errorcheck(double *C, double *D, float test_threshold)
{
  float relerror;
  int j;
  for ( j= 0; j < A_NUM; j++) {
       relerror = fabs(C[j] - D[j]) / fabs(D[j]);
       if (relerror > test_threshold) {
          printf("error at point %d \n", j);
          printf("check threshold \n");
          printf("CPU ref %f    FPGA calc %f \n",D[j] , C[j]);
                                fflush(0);
          return(-1);
       }
  }
  return(0);
}



#ifdef XD1000
/* Return current time from an OS timer.
 * Link with -lrt for clock_gettime (you must modify the generated Makefile)
 */
struct timespec curr_time()
{
            struct timespec ts;
            clock_gettime(CLOCK_REALTIME, &ts);
            return ts;
}
#define TIME_MINUS(a,b) (((a).tv_sec-(b).tv_sec)+(double)((a).tv_nsec-(b).tv_nsec)/1000000000.0)
#endif



/* Software process, runs on Opteron
 *
 * Initializes memory with array values, then runs the reference software and
 * FPGA-accelerated hardware versions.  Compares the output of both
 * implementations and prints timing results.
```

```
 */
void call_fpga(co_memory imgmem, co_signal start, co_signal end)
{
   uint32 data;
   int j, k;
            int ierr;
            double *A,*B,*C, *D;                    // A, B, C, D are the arrays that will be used.
            float test_threshold = 1.0e-5;
            float time_measured_fpga, time_measured_cpu;

#ifdef XD1000
            struct timespec now, then;
#endif
            IF_SIM(cosim_logwindow log = cosim_logwindow_create("call_fpga");)

   //Create arrays
   A = (double *) malloc((A_NUM)*sizeof(double));   // Operand array A
   if(A==NULL) printf("ERROR!\n");                  // Test if memory is properly allocated
   B = (double *) malloc((B_NUM)*sizeof(double));   // Operand array B
   if(B==NULL) printf("ERROR!\n");
   C = (double *) malloc((A_NUM)*sizeof(double));   // Result array C from FPGA calculation
   if(C==NULL) printf("ERROR!\n");
   D = (double *) malloc((A_NUM)*sizeof(double));   // Result array D from CPU calculation
   if(D==NULL) printf("ERROR!\n");

   // Initalize operands array
            init_array(A, 2.3);
            init_array(B, 3.2);
   clear_array(C);
   clear_array(D);

   printf("the first few values of array A are %f  %f  %f  %f\n", A[2048], A[2049], A[2050], A[2051]);
   printf("\nthe first few values of array B are %f  %f  %f  %f\n", B[2048], B[2049], B[2050], B[2051]);

#ifndef XD1000
            printf("Timing statistics only calculated on XD1000 system\n");
#endif

   printf("Running reference (CPU)...\r\n");

#ifdef XD1000
            now = curr_time();
#endif

   // Get reference values from the CPU - do N_ITERATIONS times
            for ( j = 0 ; j < N_ITERATION ; j++)
      get_ref(A, B, D);

#ifdef XD1000
            then = curr_time();
            time_measured_cpu = (float)TIME_MINUS(then, now);
            printf("time: %fs\n", time_measured_cpu);
#endif
   // Print out a few examples
            printf("%f %f %f %f\n", (float)D[0],(float)D[1],(float)D[2],(float)D[3]);
            printf("%f %f %f %f\r\n", (float)D[4],(float)D[5],(float)D[6],(float)D[7]);
```

```
        printf("%f %f %f %f\r\n", (float)D[2048],(float)D[2049],(float)D[2050],(float)D[2051]);
                printf("%f %f %f %f\n", (float)D[A_NUM-8], (float)D[A_NUM-7], (float)D[A_NUM-6],
(float)D[A_NUM-5]);
                printf("%f %f %f %f\r\n",(float)D[A_NUM-4], (float)D[A_NUM-3], (float)D[A_NUM-2],
(float)D[A_NUM-1]);

        printf("Running test (FPGA)...\r\n");

                // Write the A, B, and C matrices to the SRAM
                // The FPGA will wait for a signal, then read/write matrices in the SRAM
                // while computing the product.
                co_memory_writeblock(imgmem, 0, A, (A_NUM)*sizeof(double));
                co_memory_writeblock(imgmem, (A_NUM)*sizeof(double), B, (B_NUM)*sizeof(double));
                co_memory_writeblock(imgmem, ((A_NUM)+(B_NUM))*sizeof(double), C,
(A_NUM)*sizeof(double));


    #ifdef XD1000
                now = curr_time();
    #endif

        for ( k = 0; k < N_ITERATION; k++){
                // Signal the FPGA to start, then wait for the "done" signal
                        co_signal_post(start, 0);
                        // FPGA computes result...
                        co_signal_wait(end, (co_int32*)&data);
        }

    #ifdef XD1000
                then = curr_time();
                time_measured_fpga = (float)TIME_MINUS(then, now);
                printf("time: %fs\n", time_measured_fpga);
                printf("speedup: %fx performance of CPU\n\n", time_measured_cpu/time_measured_fpga);
    #endif


                // Read the FPGA-computed result out of SRAM into 'c' for verification
                co_memory_readblock(imgmem, ((A_NUM)+(B_NUM))*sizeof(double), C,
A_NUM*sizeof(double));

        // Print out a few examples
                printf("%f %f %f %f\n", (float)C[0],(float)C[1],(float)C[2],(float)C[3]);
                printf("%f %f %f %f\r\n", (float)C[4],(float)C[5],(float)C[6],(float)C[7]);
        printf("%f %f %f %f\r\n", (float)C[2048],(float)C[2049],(float)C[2050],(float)C[2051]);
        printf("%f %f %f %f\r\n", (float)C[252],(float)C[253],(float)C[254],(float)C[255]);
                printf("%f %f %f %f\n", (float)C[A_NUM-8], (float)C[A_NUM-7], (float)C[A_NUM-6],
(float)C[A_NUM-5]);
                printf("%f %f %f %f\r\n",(float)C[A_NUM-4], (float)C[A_NUM-3], (float)C[A_NUM-2],
(float)C[A_NUM-1]);

        printf("Checking result ...\r\n");
                ierr = errorcheck(C, D, test_threshold);
                if (ierr==0) printf("Passed.\r\n");
    }
```

```
int main(int argc, char *argv[])
{
	IF_SIM(int c;)
	co_architecture my_arch;

	printf("Addition Test\r\n");
	printf("----------------\n");

	// Get list of processes to execute from the configuration function in
	// Addition_hw.c (desktop simulation) or from co_init.c (host software).
	my_arch = co_initialize(NULL);
	// Execute processes
	co_execute(my_arch);

	IF_SIM(printf("Press Enter key to continue...\n");)
	IF_SIM(c = getc(stdin);)

	return(0);
}
```

3. Addition_hw.c

```
/****

XtremeData Addition Testbench
Authors: John Rothermel + Tri Dang
Version 2 - One operand array in CPU memory one in local memory SRAM of FPGA
Date: Nov 26, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C.

****/

#include <stdio.h>
#include "co.h"
#include "co_math.h"
#include "cosim_log.h"
#include "HeadAddition.h"

// Software process, defined in mmult_sw.c
extern void call_fpga(co_memory datamem, co_signal start, co_signal end);

// Define a wide element type: bits = N_UNROLLED * bits(NUMBER)
// Evidently, this element type can store 16 double precision number.
#define WIDE co_uint1024
typedef uint64 WIDE[UN_ROLLED];

// Define macros for moving individual floating-point numbers in/out of the
// WIDE type.  Different code used in hardware generation (#ifdef
// IMPULSE_C_SYNTHESIS) and in simulation.
//
// These macros are written for double-precision numbers and must be modified
// to do SGEMM.

#ifdef IMPULSE_C_SYNTHESIS

#undef co_bit_insert
#define GETELM(x,i) to_double((x)>>(960-64*(i)))
```

```
#define PUTELM(x,i,e) x=co_bit_insert(x,960-64*(i),64,double_bits(e))
#define ASSIGN(lhs,rhs) lhs=rhs

#else

#define GETELM(x,i) to_double(x[i])
#define PUTELM(x,i,e) x[i]=double_bits(e)
#define ASSIGN(lhs,rhs) memcpy(lhs,rhs,UN_ROLLED*sizeof(double))

#endif

#ifdef IMPULSE_C_SYNTHESIS
#define DECLMEM
#else
#define DECLMEM static
#endif

 // SLOT is the constant that define how many element required for array Ain
#define SLOT MEM_TRANS/UN_ROLLED
// CHUNK is the number of chunks required to process to complete the testbench
#define CHUNK A_NUM/MEM_TRANS
// Shared memories (FPGA block RAMs) used by ioproc and mmproc to store
// array A and C.
static WIDE Ain[2*SLOT], Cout[2*SLOT];
//static WIDE Bin[2*SLOT];

// # of chunks of A read. A Chunk is defined as one piece of memory that is transfer to
// the FPGA for calculation at one time, which is represented as variable MEM_TRANS.
// In specific case where we have 2^20 numbers and trade in 2048 numbers at a time, 512 chunks
// is required to complete the calculation.
static uint32 nRead;
// # of chunks of A processed so far by mmproc.  This variable is shared
// by ioproc and mmproc to synchronize memory access.
static uint32 nDone;

/* Read inputs A and store output C in parallel with the
 * computation done by mmproc.  For example, while mmproc is computing
 * A[1] + B[1], ioproc is reading A[2] and B[2] and storing
 * the results from A[0] + B[0] back into SRAM.
 */
void ioproc(co_memory datamem, co_signal start, co_signal complete)
{
        uint32 j;
        uint32 offA, offB, offC;
   uint32 cHalf, aHalf;          // cHalf and aHalf is used to keep track of which chunk will be used
        int32 data;
   IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop
        IF_SIM(cosim_logwindow log = cosim_logwindow_create("ioproc");)

        do {  // One matrix multiplication per iteration
                offA = 0;
      //offB = (A_NUM) * sizeof(double);
                offC = (A_NUM + B_NUM) * sizeof(double);
      nRead = 0;                        // nRead is also need to be reset so that mmprocess and ioprocess are at
the same index
```

```
                    // Wait for mmproc to read matrix B
                    co_signal_wait(start,&data);


                    // Preload two chunks of A
                    co_memory_readblock(datamem, offA, Ain, 2 * SLOT * sizeof(WIDE));
                    offA += 2 * SLOT * sizeof(WIDE);
                    //co_memory_readblock(datamem, offB, Bin, 2 * SLOT * sizeof(WIDE));
          //offB += 2 * SLOT * sizeof(WIDE);
          nRead = 2;

                    j=0;
                    do {
                              while (nDone<=j);  // Wait for mmproc to compute new results
            if ((j&1)==0) {   // if j is even, take the first chunk that is stored in the array
              aHalf = cHalf = 0;
            } else {          // if j is odd, take the second chunk
              aHalf = SLOT;
              cHalf = SLOT;
            }
                              // Store a chunk of results back to C in SRAM
                              co_memory_writeblock(datamem, offC, &Cout[cHalf] , SLOT *
sizeof(WIDE));
                              offC += SLOT * sizeof(WIDE);              // Move to the next chunk address
                              j++;
                              if (j == CHUNK) break;  // Done with entire computation
                              // Read another row each of A and C for the next computation
                              co_memory_readblock(datamem, offA, &Ain[aHalf], SLOT * sizeof(WIDE));
                              offA += SLOT * sizeof(WIDE);
            //co_memory_readblock(datamem, offB, &Bin[aHalf], SLOT * sizeof(WIDE));
            //offB += SLOT * sizeof(WIDE);
                              nRead++;

                    } while (1);
                    co_signal_post(complete,j);  // Signal CPU that computation is done

                    //IF_SIM(break;)
                    IF_SIM(if(!(--loopcnt)) break;)
              } while (1);  // Always running in hardware
    }


    /* Computes C = A + B.  Access to array A in shared block RAMs is
     * synchronized with ioproc by global variables.  Matrix B is stored entirely
     * in block RAM local to this process.
     */
    void mmproc(co_memory datamem, co_signal go, co_signal startio)
    {
       uint32 offA, offB, data;
       uint32 Half ;        //Again to keep track of which chunk will be used
       int j;               // keep track of the chunks
       int k;               // A's index
       int p ;              // B's index
       DECLMEM WIDE Bin[B_NUM/UN_ROLLED];  //original
              WIDE aval, bval;     // buffer for one element of array A and B
              WIDE cres;           // buffer for one result element
```

```
        IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop

                //DECLMEM WIDE ctmp[C_NUM];
                IF_SIM(cosim_logwindow log = cosim_logwindow_create("mmproc");)

                // Array B is implemented by default in an Altera FPGA as a dual-port
                // synchronous RAM.
                //co_array_config(B,co_kind,"dualsync");

                do {  // One Addition per iteration
                        offA = 0;
                        offB = offA+(A_NUM * sizeof(double));


                        // Wait for the CPU to store A and C in SRAM
                        co_signal_wait(go,(int32*)&data);

                        // Compute C := A+B
                        //
                        // Read all of B from SRAM into local memory
                        co_memory_readblock(datamem, offB, Bin, B_NUM * sizeof(double));

                        /////
                        // EAT 11/30/2007
                        // Add co_par_break() to force co_signal_wait() and co_signal_post() into separate
                        //  stages.  Sometimes, even if the signals are uinrelated and won't cause a deadlock
                        //  situation with another process, the co_signal_post() won't occur as expected.
                        //  Previously in the DGEMM code, the co_memory_readblock() was creating this
separation.
                        //co_par_break();
                        /////

                        // Tell ioproc it can start its overlapping IO to/from A and C
                        co_signal_post(startio,data);
                nDone= 0;               //Reset nDone
                p = 0;
                        for (j=0; j < CHUNK; j++) {
                                while (nRead <= j);  // Wait for data to be made available by ioproc
                  if ((j&1) == 0){     // First chunk is selected when j is even
                    Half = 0;
                  } else {            // Second chunk is selected when j is odd
                    Half = SLOT;
                  }
                  k = 0;              // k keep track of index in one chunk
                                do {
          //          #pragma co pipeline
                                        //IF_NSIM(cres=0;)
                    ASSIGN(aval, Ain[k + Half]);
                    ASSIGN(bval, Bin[p]);
                    //ASSIGN(bval, Bin[k + Half]);   // read in B into bval
                                        // Add UN_ROLLED floating-point values.
                                        // The Impulse C compiler will schedule all these operations so
                                        // they execute in a single clock cycle.
                                        PUTELM(cres,0,GETELM(aval,0)  +  GETELM(bval,0));
                                        PUTELM(cres,1,GETELM(aval,1)  +  GETELM(bval,1));
                                        PUTELM(cres,2,GETELM(aval,2)  +  GETELM(bval,2));
```

```
                                        PUTELM(cres,3,GETELM(aval,3)  +  GETELM(bval,3));
                                        PUTELM(cres,4,GETELM(aval,4)  +  GETELM(bval,4));
                                        PUTELM(cres,5,GETELM(aval,5)  +  GETELM(bval,5));
                                        PUTELM(cres,6,GETELM(aval,6)  +  GETELM(bval,6));
                                        PUTELM(cres,7,GETELM(aval,7)  +  GETELM(bval,7));
                                        PUTELM(cres,8,GETELM(aval,8)  +  GETELM(bval,8));
                                        PUTELM(cres,9,GETELM(aval,9)  +  GETELM(bval,9));
                                        PUTELM(cres,10,GETELM(aval,10)+  GETELM(bval,10));
                                        PUTELM(cres,11,GETELM(aval,11)+  GETELM(bval,11));
                                        PUTELM(cres,12,GETELM(aval,12)+  GETELM(bval,12));
                                        PUTELM(cres,13,GETELM(aval,13)+  GETELM(bval,13));
                                        PUTELM(cres,14,GETELM(aval,14)+  GETELM(bval,14));
                                        PUTELM(cres,15,GETELM(aval,15)+  GETELM(bval,15));

/*
 * Commented out for UN_ROLLED == 16.  The FPGA has enough multiplier
 * resources to do 32 parallel double-precision MACs, but the Quartus II tools
 * cannot aggregate the various block RAMs on the device into a memory wide
 * enough to feed all 32 operators in parallel.  The memory would have to be:
 *   UN_ROLLED * sizeof(double) * 8 =
 *                  32 * 8 * 8 = 2048 bits wide
 * Unrolling by 32 may work for single-precision Addition (SGEMM)
 *
                                        PUTELM(cres,16,GETELM(aval,16)+GETELM(bval,16));
                                        PUTELM(cres,17,GETELM(aval,17)+GETELM(bval,17));
                                        PUTELM(cres,18,GETELM(aval,18)+GETELM(bval,18));
                                        PUTELM(cres,19,GETELM(aval,19)+GETELM(bval,19));
                                        PUTELM(cres,20,GETELM(aval,20)+GETELM(bval,20));
                                        PUTELM(cres,21,GETELM(aval,21)+GETELM(bval,21));
                                        PUTELM(cres,22,GETELM(aval,22)+GETELM(bval,22));
                                        PUTELM(cres,23,GETELM(aval,23)+GETELM(bval,23));
                                        PUTELM(cres,24,GETELM(aval,24)+GETELM(bval,24));
                                        PUTELM(cres,25,GETELM(aval,25)+GETELM(bval,25));
                                        PUTELM(cres,26,GETELM(aval,26)+GETELM(bval,26));
                                        PUTELM(cres,27,GETELM(aval,27)+GETELM(bval,27));
                                        PUTELM(cres,28,GETELM(aval,28)+GETELM(bval,28));
                                        PUTELM(cres,29,GETELM(aval,29)+GETELM(bval,29));
                                        PUTELM(cres,30,GETELM(aval,30)+GETELM(bval,30));
                                        PUTELM(cres,31,GETELM(aval,31)+GETELM(bval,31));
*/
                                        ASSIGN(Cout[k + Half],cres);
                                        k++;
            p++;                        // p never reset until one entire computation is done
                                } while (k < (SLOT));
                                nDone++;
                        } // end j
                        //IF_SIM(break;)
                        IF_SIM(if(!(--loopcnt)) break;)
                } while (1);  // Always running in hardware
}



// The following two functions implement a required Impulse C pattern
// Impulse C configuration function
// Describes all processes and the IO objects that connect them
```

```
void config_add(void *arg)
{
        co_signal startsig, donesig, iosig;
        co_memory shrmem;
        co_process cpu_proc, mm, io_proc;

        startsig = co_signal_create("start");
        donesig  = co_signal_create("done");
        iosig  = co_signal_create("iosig");
        // Associate the co_memory object with the default physical memory location
        // for the platform (second argument, "").  In the XD1000's case, this
        // default location is the QDR SRAM, where all matrices are stored.
        shrmem   = co_memory_create("data", "", ((A_NUM)+(B_NUM)+(A_NUM))*sizeof(double));

        cpu_proc = co_process_create("cpu_proc",(co_function)call_fpga, 3, shrmem, startsig, donesig);
        mm = co_process_create("mm",(co_function)mmproc,3,shrmem,startsig,iosig);
        io_proc = co_process_create("io_proc",(co_function)ioproc,3,shrmem,iosig,donesig);

        // Assign mmproc and ioproc to run as FPGA hardware
        co_process_config(mm, co_loc, "PE0");
        co_process_config(io_proc, co_loc, "PE0");
}

// Boilerplate code that tells the Impulse C hardware compiler to look at the
// config_add function for the application's "layout" of processes and IOs.
// In simulation, this code gives the simulation library a pointer to the
// configuration function so the library can execute it internally.
co_architecture co_initialize()
{
   return(co_architecture_create("mmex", "fpga", config_add, NULL));
}
```

B.4: Addition Benchmark Version 3 (both inputs in local memory)

1. HeadAddition.h

```
// This file provides the number of operations needed to test.
// This benchtest will deal with double floating point numbers.
// A_NUM = B_NUM = 2^15
// Variable mem_trans define the amount of memory is transfer for each memory transfer command in hardware.
// memory (bits) = mem_trans (# of doble precision number) * 64 (64 bits in 1 double precision number).
// Better explantion for this variable is in Hardware code.

#define A_NUM 32768
#define B_NUM 32768
// #define A_NUM 65536
// #define B_NUM 65536
#define MEM_TRANS 2048
#define UN_ROLLED 16
// This variable determine how many times the arrays will be recalculated
#define N_ITERATION 3000
```

2. Addition_sw.c

```
/****
```

```
XtremeData Addition Testbench
Author: John Rothermel + Tri Dang
Version 3       - All data is stored in local memory
Date: Nov 26, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C.

****/

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <time.h>
#include "co.h"
#include "cosim_log.h"
#include "HeadAddition.h"

// This variable determine how many times the arrays will be recalculated
//#define N_ITERATION 1

// If compiling for the target platform
#if defined(IMPULSE_C_TARGET)
#define XD1000
#endif

extern co_architecture co_initialize(void *);

//static double D[A_NUM];

// Create Random numbers
double msrand( double *seed )
{
   static double a = 16807.;
   static double m = 2147483647.;
   double ret_val;
   static double temp;

   temp = a * *seed;
   *seed = temp - m * (float) ((int) (temp / m));
   ret_val = *seed / m;

   return ret_val;
}



// Initalize Array
void init_array(double *A, double seed)
{
  int j;
  int exp;
  //static double seed;
  extern double msrand();
  //seed = 2.3;
  for (j = 0; j < A_NUM; j++) {
       A[j] = msrand(&seed);
```

```
            exp = (int) 10 * msrand(&seed);  // use msrand to get a random float (between 0 and 1), multiply by
10, then truncate to int
            A[j] *= pow(-1,exp);  // exponentiate -1 to random exp
        }
        return;
    }



    // Clear Array
    void clear_array(double *A)
    {
        int j;
        for (j = 0; j < A_NUM; j++){
            A[j] = 0.0;
        }
    }


    /* Reference implementation of xGEMM
     * Runs on the Opteron CPU for comparison with the FPGA implementation
     */
    void get_ref(double *A, double *B, double *D){
        double sum;
        int i;
        for (i = 0; i < A_NUM; i++){
            sum = 0;
            sum = A[i] + B[i];
            D[i] = sum;
        }
        //printf("\n the first result is %f\n", D[0]);
    }



    /* Verify that two results are equal (difference in every element <=
     * test_threshold)
     */
    int errorcheck(double *C, double *D, float test_threshold)
    {
        float relerror;
        int j;
        for ( j= 0; j < A_NUM; j++) {
            relerror = fabs(C[j] - D[j]) / fabs(D[j]);
            if (relerror > test_threshold) {
                printf("error at point %d \n", j);
                printf("check threshold \n");
                printf("CPU ref %f    FPGA calc %f \n",D[j] , C[j]);
                                    fflush(0);
                return(-1);
            }
        }
        return(0);
    }
```

```
#ifdef XD1000
/* Return current time from an OS timer.
 * Link with -lrt for clock_gettime (you must modify the generated Makefile)
 */
struct timespec curr_time()
{
        struct timespec ts;
        clock_gettime(CLOCK_REALTIME, &ts);
        return ts;
}
#define TIME_MINUS(a,b) (((a).tv_sec-(b).tv_sec)+(double)((a).tv_nsec-(b).tv_nsec)/1000000000.0)
#endif



/* Software process, runs on Opteron
 *
 * Initializes memory with array values, then runs the reference software and
 * FPGA-accelerated hardware versions.  Compares the output of both
 * implementations and prints timing results.
 */
void call_fpga(co_memory imgmem, co_signal start, co_signal end)
{
   uint32 data;
   int j, k;
        int ierr;
        double *A,*B,*C, *D;                    // A, B, C, D are the arrays that will be used.
        float test_threshold = 1.0e-8;
        float time_measured_fpga, time_measured_cpu;

#ifdef XD1000
        struct timespec now, then;
#endif
        IF_SIM(cosim_logwindow log = cosim_logwindow_create("call_fpga");)

   //Create arrays
   A = (double *) malloc((A_NUM)*sizeof(double));   // Operand array A
   if(A==NULL) printf("ERROR!\n");                  // Test if memory is properly allocated
   B = (double *) malloc((B_NUM)*sizeof(double));   // Operand array B
   if(B==NULL) printf("ERROR!\n");
   C = (double *) malloc((A_NUM)*sizeof(double));   // Result array C from FPGA calculation
   if(C==NULL) printf("ERROR!\n");
   D = (double *) malloc((A_NUM)*sizeof(double));   // Result array D from CPU calculation
   if(D==NULL) printf("ERROR!\n");

   // Initalize operands array
        init_array(A, 2.3);
        init_array(B, 3.2);
   clear_array(C);
   clear_array(D);

   printf("the first few values of array A are %f  %f  %f  %f\n", A[0], A[1], A[A_NUM-2], A[A_NUM-1]);
   printf("\nthe first few values of array B are %f  %f  %f  %f\n", B[0], B[1], B[A_NUM-2], B[A_NUM-
1]);
```

```
	#ifndef XD1000
		printf("Timing statistics only calculated on XD1000 system\n");
	#endif

	   printf("Running reference (CPU)...\r\n");

	#ifdef XD1000
		now = curr_time();
	#endif

	   // Get reference values from the CPU - do N_ITERATIONS times
		for ( j = 0 ; j < N_ITERATION ; j++)
	     get_ref(A, B, D);

	#ifdef XD1000
		then = curr_time();
		time_measured_cpu = (float)TIME_MINUS(then, now);
		printf("time: %fs\n", time_measured_cpu);
	#endif
	   // Print out a few examples
		printf("%f %f %f %f\n", (float)D[0],(float)D[1],(float)D[2],(float)D[3]);
		printf("%f %f %f %f\r\n", (float)D[4],(float)D[5],(float)D[6],(float)D[7]);
		printf("%f %f %f %f\n", (float)D[A_NUM-8], (float)D[A_NUM-7], (float)D[A_NUM-6],
(float)D[A_NUM-5]);
		printf("%f %f %f %f\r\n",(float)D[A_NUM-4], (float)D[A_NUM-3], (float)D[A_NUM-2],
(float)D[A_NUM-1]);

	   printf("Running test (FPGA)...\r\n");

		// Write the A, B, and C matrices to the SRAM
		// The FPGA will wait for a signal, then read/write matrices in the SRAM
		// while computing the product.
		co_memory_writeblock(imgmem, 0, A, (A_NUM)*sizeof(double));
		co_memory_writeblock(imgmem, (A_NUM)*sizeof(double), B, (B_NUM)*sizeof(double));
		co_memory_writeblock(imgmem, ((A_NUM)+(B_NUM))*sizeof(double), C,
(A_NUM)*sizeof(double));


	#ifdef XD1000
		now = curr_time();
	#endif

	   for ( k = 0; k < N_ITERATION; k++){
		// Signal the FPGA to start, then wait for the "done" signal
			co_signal_post(start, 0);
			// FPGA computes result...
			co_signal_wait(end, (co_int32*)&data);
	   }

	#ifdef XD1000
		then = curr_time();
		time_measured_fpga = (float)TIME_MINUS(then, now);
		printf("time: %fs\n", time_measured_fpga);
		printf("speedup: %fx performance of CPU\n\n", time_measured_cpu/time_measured_fpga);
	#endif
```

```
            // Read the FPGA-computed result out of SRAM into 'c' for verification
            co_memory_readblock(imgmem, ((A_NUM)+(B_NUM))*sizeof(double), C,
A_NUM*sizeof(double));

        // Print out a few examples
            printf("%f %f %f %f\n", (float)C[0],(float)C[1],(float)C[2],(float)C[3]);
            printf("%f %f %f %f\r\n", (float)C[4],(float)C[5],(float)C[6],(float)C[7]);
        printf("%f %f %f %f\r\n", (float)C[128],(float)C[129],(float)C[130],(float)C[131]);
        printf("%f %f %f %f\r\n", (float)C[252],(float)C[253],(float)C[254],(float)C[255]);
            printf("%f %f %f %f\n", (float)C[A_NUM-8], (float)C[A_NUM-7], (float)C[A_NUM-6],
(float)C[A_NUM-5]);
            printf("%f %f %f %f\r\n",(float)C[A_NUM-4], (float)C[A_NUM-3], (float)C[A_NUM-2],
(float)C[A_NUM-1]);

        printf("Checking result ...\r\n");
            ierr = errorcheck(C, D, test_threshold);
            if (ierr==0) printf("Passed.\r\n");
    }


    int main(int argc, char *argv[])
    {
            IF_SIM(int c;)
            co_architecture my_arch;

            printf("Addition Test\r\n");
            printf("----------------\n");

            // Get list of processes to execute from the configuration function in
            // Addition_hw.c (desktop simulation) or from co_init.c (host software).
            my_arch = co_initialize(NULL);
            // Execute processes
            co_execute(my_arch);

            IF_SIM(printf("Press Enter key to continue...\n");)
            IF_SIM(c = getc(stdin);)

            return(0);
    }
```

3. Addition_hw.c

```
    /****

    XtremeData Addition Testbench
    Authors: John Rothermel + Tri Dang
    Version 3       - All data is stored in local memory
    Date: Nov 26, 2007
    Reference: Double precision Matrix Multiplication project provided by Impulse C.

    ****/

    #include <stdio.h>
    #include "co.h"
    #include "co_math.h"
    #include "cosim_log.h"
```

```
#include "HeadAddition.h"

// Software process, defined in mmult_sw.c
extern void call_fpga(co_memory datamem, co_signal start, co_signal end);

// Define a wide element type: bits = N_UNROLLED * bits(NUMBER)
// Evidently, this element type can store 16 double precision number.
#define WIDE co_uint1024
typedef uint64 WIDE[UN_ROLLED];

// Define macros for moving individual floating-point numbers in/out of the
// WIDE type.  Different code used in hardware generation (#ifdef
// IMPULSE_C_SYNTHESIS) and in simulation.
//
// These macros are written for double-precision numbers and must be modified
// to do SGEMM.

#ifdef IMPULSE_C_SYNTHESIS

#undef co_bit_insert
#define GETELM(x,i) to_double((x)>>(960-64*(i)))
#define PUTELM(x,i,e) x=co_bit_insert(x,960-64*(i),64,double_bits(e))
#define ASSIGN(lhs,rhs) lhs=rhs

#else

#define GETELM(x,i) to_double(x[i])
#define PUTELM(x,i,e) x[i]=double_bits(e)
#define ASSIGN(lhs,rhs) memcpy(lhs,rhs,UN_ROLLED*sizeof(double))

#endif

#ifdef IMPULSE_C_SYNTHESIS
#define DECLMEM
#else
#define DECLMEM static
#endif

 // SLOT is the constant that define how many element required for one chunk for array Cout
#define SLOT MEM_TRANS/UN_ROLLED    // SLOT = 128
// CHUNK is the number of chunks required to process to complete the testbench
#define CHUNK A_NUM/MEM_TRANS       // CHUNK  = 16

// Shared memories (FPGA block RAMs) used by ioproc and mmproc to store
// array C.

static WIDE Cout[2*SLOT];
//static WIDE Ain[2*SLOT], Cout[2*SLOT];
//static WIDE Bin[2*SLOT];

// # of chunks of C written. A Chunk is defined as one piece of memory that is transfer to
// the FPGA for calculation at one time, which is represented as variable MEM_TRANS.
// In specific case where we have 2^16 numbers and trade in 2048 numbers at a time, 512 chunks
// is required to complete the calculation.
static uint32 nRead;
// # of chunks of A processed so far by mmproc.  This variable is shared
```

```
// by ioproc and mmproc to synchronize memory access.
static uint32 nDone;

/* Store output C in parallel with the
 * computation done by mmproc.
 */
void ioproc(co_memory datamem, co_signal start, co_signal complete)
{
        uint32 j;
        uint32 offA, offB, offC;
   uint32 Half;          // cHalf and aHalf is used to keep track of which chunk will be used
        int32 data;
   IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop
        IF_SIM(cosim_logwindow log = cosim_logwindow_create("ioproc");)

        do {  // One matrix multiplication per iteration
                offA = 0;
      offB = (A_NUM) * sizeof(double);
                offC = (A_NUM + B_NUM) * sizeof(double);
      nRead = 1;                    // nRead is also need to be reset so that mmprocess and ioprocess are at
the same index

                // Wait for mmproc to read matrix B
                co_signal_wait(start,&data);


                // Preload two chunks of A
                //co_memory_readblock(datamem, offA, Ain, 2 * SLOT * sizeof(WIDE));
                //offA += 2 * SLOT * sizeof(WIDE);
                //co_memory_readblock(datamem, offB, Bin, 2 * SLOT * sizeof(WIDE));
      //offB += 2 * SLOT * sizeof(WIDE);
      //nRead = 2;

                j=0;
                do {
                        while (nDone <= j);  // Wait for mmproc to compute new results
          if ((j&1)==0) {   // if j is even, take the first chunk that is stored in the array
           Half = 0;
          } else {         // if j is odd, take the second chunk
           Half = SLOT;
          }
                        // Store a chunk of results back to C in SRAM
                        co_memory_writeblock(datamem, offC, &Cout[Half] , SLOT * sizeof(WIDE));
                        offC += SLOT * sizeof(WIDE);                    // Move to the next
chunk address
                        j++;
                        if (j == CHUNK) break;                          // Done with entire
computation
                        // Read another row each of A and C for the next computation
                        //co_memory_readblock(datamem, offA, &Ain[aHalf], SLOT * sizeof(WIDE));
                        //offA += SLOT * sizeof(WIDE);
          //co_memory_readblock(datamem, offB, &Bin[aHalf], SLOT * sizeof(WIDE));
          //offB += SLOT * sizeof(WIDE);
                        nRead++;

                } while (1);
```

```
                co_signal_post(complete,j);  // Signal CPU that computation is done

                //IF_SIM(break;)
                IF_SIM(if(!(--loopcnt)) break;)
            } while (1);  // Always running in hardware
    }


    /* Computes C = A + B.  Access to array A in shared block RAMs is
     * synchronized with ioproc by global variables.  Matrix B is stored entirely
     * in block RAM local to this process.
     */
    void mmproc(co_memory datamem, co_signal go, co_signal startio)
    {
       uint32 offA, offB, data;
       uint32 Half ;        //Again to keep track of which chunk will be used
       int j;              // keep track of the chunks
       int k;
       int m;               // A's and B's index

       DECLMEM WIDE Bin[B_NUM/UN_ROLLED];  // local memmory for array B
       DECLMEM WIDE Ain[A_NUM/UN_ROLLED];  // local memory for array A

            WIDE aval, bval;     // buffer for one element of array A and B
            WIDE cres;          // buffer for one result element
       IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop

            //DECLMEM WIDE ctmp[C_NUM];
            IF_SIM(cosim_logwindow log = cosim_logwindow_create("mmproc");)

            // Array B is implemented by default in an Altera FPGA as a dual-port
            // synchronous RAM.
            //co_array_config(B,co_kind,"dualsync");

            do {  // One Addition per iteration
                offA = 0;
                offB = offA+(A_NUM * sizeof(double));


                // Wait for the CPU to store A and C in SRAM
                co_signal_wait(go,(int32*)&data);

                // Read all of B from SRAM into local memory
                co_memory_readblock(datamem, offB, Bin, B_NUM * sizeof(double));

          // Read all of A from SRAM into local memory
          co_memory_readblock(datamem, offA, Ain, A_NUM * sizeof(double));

                //////
                // EAT 11/30/2007
                // Add co_par_break() to force co_signal_wait() and co_signal_post() into separate
                //  stages.  Sometimes, even if the signals are uinrelated and won't cause a deadlock
                //  situation with another process, the co_signal_post() won't occur as expected.
                //  Previously in the DGEMM code, the co_memory_readblock() was creating this
separation.
```

```
                    //co_par_break();
                    /////

                    // Tell ioproc it can start its overlapping IO to/from A and C
                    co_signal_post(startio,data);
        nDone= 0;               //Reset nDone
        m = 0;              // Initalize A and B index
                    for (j=0; j < CHUNK; j++) {       //Chunk = 32
                                while (nRead <= j);  // Wait for data to be made available by ioproc
          if ((j&1) == 0){     // First chunk is selected when j is even
            Half = 0;
          } else {            // Second chunk is selected when j is odd
            Half = SLOT;
          }
          k = 0;              // k keep track of index in one slot
                                do {
//          #pragma co pipeline
                                        //IF_NSIM(cres=0;)
            ASSIGN(aval, Ain[m]);   // read A into aval
            ASSIGN(bval, Bin[m]);   // read B into bval
                                        // Add UN_ROLLED floating-point values.
                                        // The Impulse C compiler will schedule all these operations so
                                        // they execute in a single clock cycle.
                                        PUTELM(cres,0,GETELM(aval,0)  +  GETELM(bval,0));
                                        PUTELM(cres,1,GETELM(aval,1)  +  GETELM(bval,1));
                                        PUTELM(cres,2,GETELM(aval,2)  +  GETELM(bval,2));
                                        PUTELM(cres,3,GETELM(aval,3)  +  GETELM(bval,3));
                                        PUTELM(cres,4,GETELM(aval,4)  +  GETELM(bval,4));
                                        PUTELM(cres,5,GETELM(aval,5)  +  GETELM(bval,5));
                                        PUTELM(cres,6,GETELM(aval,6)  +  GETELM(bval,6));
                                        PUTELM(cres,7,GETELM(aval,7)  +  GETELM(bval,7));
                                        PUTELM(cres,8,GETELM(aval,8)  +  GETELM(bval,8));
                                        PUTELM(cres,9,GETELM(aval,9)  +  GETELM(bval,9));
                                        PUTELM(cres,10,GETELM(aval,10)+  GETELM(bval,10));
                                        PUTELM(cres,11,GETELM(aval,11)+  GETELM(bval,11));
                                        PUTELM(cres,12,GETELM(aval,12)+  GETELM(bval,12));
                                        PUTELM(cres,13,GETELM(aval,13)+  GETELM(bval,13));
                                        PUTELM(cres,14,GETELM(aval,14)+  GETELM(bval,14));
                                        PUTELM(cres,15,GETELM(aval,15)+  GETELM(bval,15));

/*
 * Commented out for UN_ROLLED == 16.  The FPGA has enough multiplier
 * resources to do 32 parallel double-precision MACs, but the Quartus II tools
 * cannot aggregate the various block RAMs on the device into a memory wide
 * enough to feed all 32 operators in parallel.  The memory would have to be:
 *   UN_ROLLED * sizeof(double) * 8 =
 *                   32 * 8 * 8 = 2048 bits wide
 * Unrolling by 32 may work for single-precision Addition (SGEMM)
 *
                                        PUTELM(cres,16,GETELM(aval,16)+GETELM(bval,16));
                                        PUTELM(cres,17,GETELM(aval,17)+GETELM(bval,17));
                                        PUTELM(cres,18,GETELM(aval,18)+GETELM(bval,18));
                                        PUTELM(cres,19,GETELM(aval,19)+GETELM(bval,19));
                                        PUTELM(cres,20,GETELM(aval,20)+GETELM(bval,20));
                                        PUTELM(cres,21,GETELM(aval,21)+GETELM(bval,21));
                                        PUTELM(cres,22,GETELM(aval,22)+GETELM(bval,22));
```

```
                                        PUTELM(cres,23,GETELM(aval,23)+GETELM(bval,23));
                                        PUTELM(cres,24,GETELM(aval,24)+GETELM(bval,24));
                                        PUTELM(cres,25,GETELM(aval,25)+GETELM(bval,25));
                                        PUTELM(cres,26,GETELM(aval,26)+GETELM(bval,26));
                                        PUTELM(cres,27,GETELM(aval,27)+GETELM(bval,27));
                                        PUTELM(cres,28,GETELM(aval,28)+GETELM(bval,28));
                                        PUTELM(cres,29,GETELM(aval,29)+GETELM(bval,29));
                                        PUTELM(cres,30,GETELM(aval,30)+GETELM(bval,30));
                                        PUTELM(cres,31,GETELM(aval,31)+GETELM(bval,31));
*/
                                        ASSIGN(Cout[k + Half],cres);
                                        k++;
            m++;
                                } while (k < (SLOT));
                                nDone++;
                        } // end j
                        //IF_SIM(break;)
                        IF_SIM(if(!(--loopcnt)) break;)
                } while (1);  // Always running in hardware
}



// The following two functions implement a required Impulse C pattern
// Impulse C configuration function
// Describes all processes and the IO objects that connect them
void config_add(void *arg)
{
        co_signal startsig, donesig, iosig;
        co_memory shrmem;
        co_process cpu_proc, mm, io_proc;

        startsig = co_signal_create("start");
        donesig  = co_signal_create("done");
        iosig  = co_signal_create("iosig");
        // Associate the co_memory object with the default physical memory location
        // for the platform (second argument, "").  In the XD1000's case, this
        // default location is the QDR SRAM, where all matrices are stored.
        shrmem   = co_memory_create("data", "", ((A_NUM)+(B_NUM)+(A_NUM))*sizeof(double));

        cpu_proc = co_process_create("cpu_proc",(co_function)call_fpga, 3, shrmem, startsig, donesig);
        mm = co_process_create("mm",(co_function)mmproc,3,shrmem,startsig,iosig);
        io_proc = co_process_create("io_proc",(co_function)ioproc,3,shrmem,iosig,donesig);

        // Assign mmproc and ioproc to run as FPGA hardware
        co_process_config(mm, co_loc, "PE0");
        co_process_config(io_proc, co_loc, "PE0");
}

// Boilerplate code that tells the Impulse C hardware compiler to look at the
// config_add function for the application's "layout" of processes and IOs.
// In simulation, this code gives the simulation library a pointer to the
// configuration function so the library can execute it internally.
co_architecture co_initialize()
{
   return(co_architecture_create("mmex", "fpga", config_add, NULL));
```

```
}
```

B.5: Multiplication Benchmark Version

1. HeadMultiplication.h

```
// This file provides the number of operations needed to test.
// This benchtest will deal with double floating point numbers.
// A_NUM = B_NUM = 2^17
// Variable mem_trans define the amount of memory is transfer for each memory transfer command in hardware.
// memory (bits) = mem_trans (# of doble precision number) * 64 (64 bits in 1 double precision number).
// Better explantion for this variable is in Hardware code.

#define A_NUM 131072
#define B_NUM 131072
// #define A_NUM 65536
// #define B_NUM 65536
#define MEM_TRANS 256
#define UN_ROLLED 16
// This variable determine how many times the arrays will be recalculated
#define N_ITERATION 900
```

2. Multiplication_sw.c

```
/****

XtremeData Multiplication Testbench
Author: John Rothermel + Tri Dang
Version 1
Date: Nov 26, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C.

****/

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <time.h>
#include "co.h"
#include "cosim_log.h"
#include "HeadMultiplication.h"


// If compiling for the target platform
#if defined(IMPULSE_C_TARGET)
#define XD1000
#endif

extern co_architecture co_initialize(void *);

//static double D[A_NUM];

// Create Random numbers
double msrand( double *seed )
{
```

```
    static double a = 16807.;
    static double m = 2147483647.;
    double ret_val;
    static double temp;

    temp = a * *seed;
    *seed = temp - m * (float) ((int) (temp / m));
    ret_val = *seed / m;

    return ret_val;
}



// Initalize Array
void init_array(double *A, double seed)
{
  int j;
  int exp;
  //static double seed;
  extern double msrand();
  //seed = 2.3;
  for (j = 0; j < A_NUM; j++) {
       A[j] = msrand(&seed);  //get random floating point value based on seed
       exp = (int) 10 * msrand(&seed);  // use msrand to get a random float (between 0 and 1), multiply by
10, then truncate to int
       A[j] *= pow(-1,exp);  // exponentiate -1 to random exp
  }
  return;
}



// Clear Array
void clear_array(double *A)
{
  int j;
  for (j = 0; j < A_NUM; j++){
     A[j] = 0.0;
  }
}


/* Reference implementation of xGEMM
 * Runs on the Opteron CPU for comparison with the FPGA implementation
 */
void get_ref(double *A, double *B, double *D){
    double sum;
    int i;
    for (i = 0; i < A_NUM; i++){
      sum = 0;
      sum = A[i] * B[i];
      D[i] = sum;
    }
}
```

```
/* Verify that two results are equal (difference in every element <=
 * test_threshold)
 */
int errorcheck(double *C, double *D, float test_threshold)
{
  float relerror;
  int j;
  for ( j=0; j< A_NUM; j++) {
       relerror = fabs(C[j] - D[j]) / fabs(D[j]);
       if (relerror > test_threshold) {
          printf("error at point %d \n", j);
          printf("check threshold \n");
          printf("CPU ref %f    FPGA calc %f \n",D[j] , C[j]);
                                fflush(0);
          return(-1);
       }
  }
  return(0);
}



#ifdef XD1000
/* Return current time from an OS timer.
 * Link with -lrt for clock_gettime (you must modify the generated Makefile)
 */
struct timespec curr_time()
{
            struct timespec ts;
            clock_gettime(CLOCK_REALTIME, &ts);
            return ts;
}
#define TIME_MINUS(a,b) (((a).tv_sec-(b).tv_sec)+(double)((a).tv_nsec-(b).tv_nsec)/1000000000.0)
#endif



/* Software process, runs on Opteron
 *
 * Initializes memory with array values, then runs the reference software and
 * FPGA-accelerated hardware versions.  Compares the output of both
 * implementations and prints timing results.
 */
void call_fpga(co_memory imgmem, co_signal start, co_signal end)
{
   uint32 data;
   int j, k;
            int ierr;
            double *A,*B,*C, *D;                  // A, B, C, D are the arrays that will be used.
            float test_threshold = 1.0e-10;
            float time_measured_fpga, time_measured_cpu;

#ifdef XD1000
            struct timespec now, then;
```

```
    #endif
            IF_SIM(cosim_logwindow log = cosim_logwindow_create("call_fpga");)

       //Create arrays
       A = (double *) malloc((A_NUM)*sizeof(double));   // Operand array A
       if(A==NULL) printf("ERROR!\n");                  // Test if memory is properly allocated
       B = (double *) malloc((B_NUM)*sizeof(double));   // Operand array B
       if(B==NULL) printf("ERROR!\n");
       C = (double *) malloc((A_NUM)*sizeof(double));   // Result array C from FPGA calculation
       if(C==NULL) printf("ERROR!\n");
       D = (double *) malloc((A_NUM)*sizeof(double));   // Result array D from CPU calculation
       if(D==NULL) printf("ERROR!\n");

       // Initalize operands array
            init_array(A, 2.3);
            init_array(B, 3.2);
       clear_array(C);
       clear_array(D);

       printf("the first few values of array A are %f  %f  %f  %f\n", A[0], A[1], A[A_NUM-2], A[A_NUM-1]);
       printf("\nthe first few values of array B are %f  %f  %f  %f\n", B[0], B[1], B[A_NUM-2], B[A_NUM-
1]);

    #ifndef XD1000
            printf("Timing statistics only calculated on XD1000 system\n");
    #endif

       printf("Running reference (CPU)...\r\n");

    #ifdef XD1000
            now = curr_time();
    #endif

       // Get reference values from the CPU - do N_ITERATIONS times
            for ( j = 0 ; j < N_ITERATION ; j++)
         get_ref(A, B, D);

    #ifdef XD1000
            then = curr_time();
            time_measured_cpu = (float)TIME_MINUS(then, now);
            printf("time: %fs\n", time_measured_cpu);
    #endif
       // Print out a few examples
            printf("%f %f %f %f\n", (float)D[0],(float)D[1],(float)D[2],(float)D[3]);
            printf("%f %f %f %f\r\n", (float)D[4],(float)D[5],(float)D[6],(float)D[7]);
            printf("%f %f %f %f\n", (float)D[A_NUM-8], (float)D[A_NUM-7], (float)D[A_NUM-6],
(float)D[A_NUM-5]);
            printf("%f %f %f %f\r\n",(float)D[A_NUM-4], (float)D[A_NUM-3], (float)D[A_NUM-2],
(float)D[A_NUM-1]);

       printf("Running test (FPGA)...\r\n");

            // Write the A, B, and C matrices to the SRAM
            // The FPGA will wait for a signal, then read/write matrices in the SRAM
            // while computing the product.
            co_memory_writeblock(imgmem, 0, A, (A_NUM)*sizeof(double));
```

```
        co_memory_writeblock(imgmem, (A_NUM)*sizeof(double), B, (B_NUM)*sizeof(double));
        co_memory_writeblock(imgmem, ((A_NUM)+(B_NUM))*sizeof(double), C,
(A_NUM)*sizeof(double));


    #ifdef XD1000
        now = curr_time();
    #endif

      for ( k = 0; k < N_ITERATION; k++){
        // Signal the FPGA to start, then wait for the "done" signal
            co_signal_post(start, 0);
            // FPGA computes result...
            co_signal_wait(end, (co_int32*)&data);
      }


    #ifdef XD1000
        then = curr_time();
        time_measured_fpga = (float)TIME_MINUS(then, now);
        printf("time: %fs\n", time_measured_fpga);
        printf("speedup: %fx performance of CPU\n\n", time_measured_cpu/time_measured_fpga);
    #endif


        // Read the FPGA-computed result out of SRAM into 'c' for verification
        co_memory_readblock(imgmem, ((A_NUM)+(B_NUM))*sizeof(double), C,
A_NUM*sizeof(double));

      // Print out a few examples
        printf("%f %f %f %f\n", (float)C[0],(float)C[1],(float)C[2],(float)C[3]);
        printf("%f %f %f %f\r\n", (float)C[4],(float)C[5],(float)C[6],(float)C[7]);
      printf("%f %f %f %f\r\n", (float)C[128],(float)C[129],(float)C[130],(float)C[131]);
      printf("%f %f %f %f\r\n", (float)C[252],(float)C[253],(float)C[254],(float)C[255]);
        printf("%f %f %f %f\n", (float)C[A_NUM-8], (float)C[A_NUM-7], (float)C[A_NUM-6],
(float)C[A_NUM-5]);
        printf("%f %f %f %f\r\n",(float)C[A_NUM-4], (float)C[A_NUM-3], (float)C[A_NUM-2],
(float)C[A_NUM-1]);

      printf("Checking result ...\r\n");
        ierr = errorcheck(C, D, test_threshold);
        if (ierr==0) printf("Passed.\r\n");
    }


    int main(int argc, char *argv[])
    {
        IF_SIM(int c;)
        co_architecture my_arch;

        printf("Multiplication Test\r\n");
        printf("----------------\n");

        // Get list of processes to execute from the configuration function in
        // Addition_hw.c (desktop simulation) or from co_init.c (host software).
        my_arch = co_initialize(NULL);
```

```
        // Execute processes
        co_execute(my_arch);

        IF_SIM(printf("Press Enter key to continue...\n");)
        IF_SIM(c = getc(stdin);)

        return(0);
}
```

3. Multiplication_hw.c

```
/****

XtremeData Multiplication Testbench
Authors: John Rothermel + Tri Dang
Version 1
Date: Nov 26, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C.

****/

#include <stdio.h>
#include "co.h"
#include "cosim_log.h"
#include "HeadMultiplication.h"

// Software process, defined in mmult_sw.c
extern void call_fpga(co_memory datamem, co_signal start, co_signal end);

// Define a wide element type: bits = N_UNROLLED * bits(NUMBER)
// Evidently, this element type can store 16 double precision number.
#define WIDE co_uint1024
typedef uint64 WIDE[UN_ROLLED];

// Define macros for moving individual floating-point numbers in/out of the
// WIDE type.  Different code used in hardware generation (#ifdef
// IMPULSE_C_SYNTHESIS) and in simulation.
//
// These macros are written for double-precision numbers and must be modified
// to do SGEMM.

#ifdef IMPULSE_C_SYNTHESIS

#undef co_bit_insert
#define GETELM(x,i) to_double((x)>>(960-64*(i)))
#define PUTELM(x,i,e) x=co_bit_insert(x,960-64*(i),64,double_bits(e))
#define ASSIGN(lhs,rhs) lhs=rhs

#else

#define GETELM(x,i) to_double(x[i])
#define PUTELM(x,i,e) x[i]=double_bits(e)
#define ASSIGN(lhs,rhs) memcpy(lhs,rhs,UN_ROLLED*sizeof(double))

#endif

#ifdef IMPULSE_C_SYNTHESIS
```

```
#define DECLMEM
#else
#define DECLMEM static
#endif

 // SLOT is the constant that define how many element required for array Ain
#define SLOT MEM_TRANS/UN_ROLLED
// CHUNK is the number of chunks required to process to complete the testbench
#define CHUNK A_NUM/MEM_TRANS
// Shared memories (FPGA block RAMs) used by ioproc and mmproc to store
// array A and C.
static WIDE Ain[2*SLOT], Cout[2*SLOT];
static WIDE Bin[2*SLOT];

// # of chunks of A read. A Chunk is defined as one piece of memory that is transfer to
// the FPGA for calculation at one time, which is represented as variable MEM_TRANS.
// In specific case where we have 2^20 numbers and trade in 2048 numbers at a time, 512 chunks
// is required to complete the calculation.
static uint32 nRead;
// # of chunks of A processed so far by mmproc.  This variable is shared
// by ioproc and mmproc to synchronize memory access.
static uint32 nDone;

/* Read inputs A and store output C in parallel with the
 * computation done by mmproc.  For example, while mmproc is computing
 * A[1] + B[1], ioproc is reading A[2] and B[2] and storing
 * the results from A[0] + B[0] back into SRAM.
 */
void ioproc(co_memory datamem, co_signal start, co_signal complete)
{
        uint32 j;
        uint32 offA, offB, offC;
   uint32 cHalf, aHalf;          // cHalf and aHalf is used to keep track of which chunk will be used
        int32 data;
   IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop
        IF_SIM(cosim_logwindow log = cosim_logwindow_create("ioproc");)

        do {  // One matrix multiplication per iteration
                offA = 0;
      offB = (A_NUM) * sizeof(double);
                offC = (A_NUM + B_NUM) * sizeof(double);
      nRead = 0;                        // nRead is also need to be reset so that mmprocess and ioprocess are at
the same index

                // Wait for mmproc to read matrix B
                co_signal_wait(start,&data);


                // Preload two chunks of A and B
                co_memory_readblock(datamem, offA, Ain, 2 * SLOT * sizeof(WIDE));
                offA += 2 * SLOT * sizeof(WIDE);
                co_memory_readblock(datamem, offB, Bin, 2 * SLOT * sizeof(WIDE));
      offB += 2 * SLOT * sizeof(WIDE);
      nRead = 2;

                j=0;
```

```
                do {
                        while (nDone<=j);  // Wait for mmproc to compute new results
            if ((j&1)==0) {   // if j is even, take the first chunk that is stored in the array
              aHalf = cHalf = 0;
            } else {        // if j is odd, take the second chunk
              aHalf = SLOT;
              cHalf = SLOT;
            }
                        // Store a chunk of results back to C in SRAM
                        co_memory_writeblock(datamem, offC, &Cout[cHalf] , SLOT *
sizeof(WIDE));
                        offC += SLOT * sizeof(WIDE);              // Move to the next chunk address
                        j++;
                        if (j == CHUNK) break;  // Done with entire computation
                        // Read another row each of A and C for the next computation
                        co_memory_readblock(datamem, offA, &Ain[aHalf], SLOT * sizeof(WIDE));
                        offA += SLOT * sizeof(WIDE);
            co_memory_readblock(datamem, offB, &Bin[aHalf], SLOT * sizeof(WIDE));
            offB += SLOT * sizeof(WIDE);
                        nRead++;

                } while (1);
                co_signal_post(complete,j);  // Signal CPU that computation is done

                //IF_SIM(break;)
                IF_SIM(if(!(--loopcnt)) break;)
            } while (1);  // Always running in hardware
    }


    /* Computes C = A + B.  Access to array A in shared block RAMs is
     * synchronized with ioproc by global variables.  Matrix B is stored entirely
     * in block RAM local to this process.
     */
    void mmproc(co_memory datamem, co_signal go, co_signal startio)
    {
       uint32 offA, offB, data;
       uint32 Half ;        //Again to keep track of which chunk will be used
            double m;
       int j;              // keep track of the chunks
       int k;              // A's index
       //DECLMEM WIDE Bin[B_NUM/UN_ROLLED];  //original
            WIDE aval, bval;     // buffer for one element of array A and B
            WIDE cres;          // buffer for one result element
       IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop

            //DECLMEM WIDE ctmp[C_NUM];
            IF_SIM(cosim_logwindow log = cosim_logwindow_create("mmproc");)

            // Array B is implemented by default in an Altera FPGA as a dual-port
            // synchronous RAM.
            //co_array_config(B,co_kind,"dualsync");

            do {  // One Addition per iteration
                    offA = 0;
                    offB = offA+(A_NUM * sizeof(double));
```

```
                    // Wait for the CPU to store A and C in SRAM
                    co_signal_wait(go,(int32*)&data);
                    // Compute C := A*B
                    //
                    // Read all of B from SRAM into local memory
                    //co_memory_readblock(datamem, offB, Bin, B_NUM * sizeof(double));

          /////
                    // EAT (Edward Trexel) 11/30/2007 --
                    // Add co_par_break() to force co_signal_wait() and co_signal_post() into separate
                    //  stages.  Sometimes, even if the signals are uinrelated and won't cause a deadlock
                    //  situation with another process, the co_signal_post() won't occur as expected.
                    //  Previously in the DGEMM code, the co_memory_readblock() was creating this
separation.
                    co_par_break();
                    /////

                    // Tell ioproc it can start its overlapping IO to/from A and C
                    co_signal_post(startio,data);
          nDone= 0;              //Reset nDone

                    for (j=0; j < CHUNK; j++) {       //Chunk = 512
                              while (nRead <= j);  // Wait for data to be made available by ioproc
            if ((j&1) == 0){     // First chunk is selected when j is even
              Half = 0;
            } else {            // Second chunk is selected when j is odd
              Half = SLOT;
            }
            k = 0;              // k keep track of index in one chunk
                              do {
              #pragma co pipeline
                                        //IF_NSIM(cres=0;)
              ASSIGN(aval, Ain[k + Half]);
              //ASSIGN(bval, Bin[p]);
              ASSIGN(bval, Bin[k + Half]);   // read in B into bval
                                        // Add UN_ROLLED floating-point values.
                                        // The Impulse C compiler will schedule all these operations so
                                        // they execute in a single clock cycle.
                                        PUTELM(cres,0,GETELM(aval,0)  *  GETELM(bval,0));
                                        PUTELM(cres,1,GETELM(aval,1)  *  GETELM(bval,1));
                                        PUTELM(cres,2,GETELM(aval,2)  *  GETELM(bval,2));
                                        PUTELM(cres,3,GETELM(aval,3)  *  GETELM(bval,3));
                                        PUTELM(cres,4,GETELM(aval,4)  *  GETELM(bval,4));
                                        PUTELM(cres,5,GETELM(aval,5)  *  GETELM(bval,5));
                                        PUTELM(cres,6,GETELM(aval,6)  *  GETELM(bval,6));
                                        PUTELM(cres,7,GETELM(aval,7)  *  GETELM(bval,7));
                                        PUTELM(cres,8,GETELM(aval,8)  *  GETELM(bval,8));
                                        PUTELM(cres,9,GETELM(aval,9)  *  GETELM(bval,9));
                                        PUTELM(cres,10,GETELM(aval,10)*  GETELM(bval,10));
                                        PUTELM(cres,11,GETELM(aval,11)*  GETELM(bval,11));
                                        PUTELM(cres,12,GETELM(aval,12)*  GETELM(bval,12));
                                        PUTELM(cres,13,GETELM(aval,13)*  GETELM(bval,13));
                                        PUTELM(cres,14,GETELM(aval,14)*  GETELM(bval,14));
                                        PUTELM(cres,15,GETELM(aval,15)*  GETELM(bval,15));
```

```
/*
 * Commented out for UN_ROLLED == 16.  The FPGA has enough multiplier
 * resources to do 32 parallel double-precision MACs, but the Quartus II tools
 * cannot aggregate the various block RAMs on the device into a memory wide
 * enough to feed all 32 operators in parallel.  The memory would have to be:
 *   UN_ROLLED * sizeof(double) * 8 =
 *                  32 * 8 * 8 = 2048 bits wide
 * Unrolling by 32 may work for single-precision Addition (SGEMM)
 *
                                        PUTELM(cres,16,GETELM(aval,16)+GETELM(bval,16));
                                        PUTELM(cres,17,GETELM(aval,17)+GETELM(bval,17));
                                        PUTELM(cres,18,GETELM(aval,18)+GETELM(bval,18));
                                        PUTELM(cres,19,GETELM(aval,19)+GETELM(bval,19));
                                        PUTELM(cres,20,GETELM(aval,20)+GETELM(bval,20));
                                        PUTELM(cres,21,GETELM(aval,21)+GETELM(bval,21));
                                        PUTELM(cres,22,GETELM(aval,22)+GETELM(bval,22));
                                        PUTELM(cres,23,GETELM(aval,23)+GETELM(bval,23));
                                        PUTELM(cres,24,GETELM(aval,24)+GETELM(bval,24));
                                        PUTELM(cres,25,GETELM(aval,25)+GETELM(bval,25));
                                        PUTELM(cres,26,GETELM(aval,26)+GETELM(bval,26));
                                        PUTELM(cres,27,GETELM(aval,27)+GETELM(bval,27));
                                        PUTELM(cres,28,GETELM(aval,28)+GETELM(bval,28));
                                        PUTELM(cres,29,GETELM(aval,29)+GETELM(bval,29));
                                        PUTELM(cres,30,GETELM(aval,30)+GETELM(bval,30));
                                        PUTELM(cres,31,GETELM(aval,31)+GETELM(bval,31));
*/
                                        ASSIGN(Cout[k + Half],cres);
                                        k++;
                                } while (k < (SLOT));
                                nDone++;
                        } // end j
                        //IF_SIM(break;)
                        IF_SIM(if(!(--loopcnt)) break;)
                } while (1);  // Always running in hardware
}



// The following two functions implement a required Impulse C pattern
// Impulse C configuration function
// Describes all processes and the IO objects that connect them
void config_mul(void *arg)
{
        co_signal startsig, donesig, iosig;
        co_memory shrmem;
        co_process cpu_proc, mm, io_proc;

        startsig = co_signal_create("start");
        donesig  = co_signal_create("done");
        iosig  = co_signal_create("iosig");
        // Associate the co_memory object with the default physical memory location
        // for the platform (second argument, "").  In the XD1000's case, this
        // default location is the QDR SRAM, where all matrices are stored.
        shrmem   = co_memory_create("data", "", ((A_NUM)+(B_NUM)+(A_NUM))*sizeof(double));
```

```
        cpu_proc = co_process_create("cpu_proc",(co_function)call_fpga, 3, shrmem, startsig, donesig);
        mm = co_process_create("mm",(co_function)mmproc,3,shrmem,startsig,iosig);
        io_proc = co_process_create("io_proc",(co_function)ioproc,3,shrmem,iosig,donesig);

        // Assign mmproc and ioproc to run as FPGA hardware
        co_process_config(mm, co_loc, "PE0");
        co_process_config(io_proc, co_loc, "PE0");
}

// Boilerplate code that tells the Impulse C hardware compiler to look at the
// config_add function for the application's "layout" of processes and IOs.
// In simulation, this code gives the simulation library a pointer to the
// configuration function so the library can execute it internally.
co_architecture co_initialize()
{
   return(co_architecture_create("mmex", "fpga", config_mul, NULL));
}
```

B.6: Division Benchmark

1. HeadDivision.h

```
// This file provides the number of operations needed to test.
// This benchtest will deal with double floating point numbers.
// A_NUM = B_NUM = 2^17
// Variable mem_trans define the amount of memory is transfer for each memory transfer command in hardware.
// memory (bits) = mem_trans (# of doble precision number) * 64 (64 bits in 1 double precision number).
// Better explantion for this variable is in Hardware code.

#define A_NUM 131072
#define B_NUM 131072
// #define A_NUM 65536
// #define B_NUM 65536
#define MEM_TRANS 2048
#define UN_ROLLED 8
// This variable determine how many times the arrays will be recalculated
#define N_ITERATION 1000
```

2. Division_sw.c

```
/****

XtremeData Division Testbench
Author: John Rothermel & Tri Dang
Version 1
Date: Nov 26, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C.

****/

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <time.h>
#include "co.h"
```

```
#include "cosim_log.h"
#include "HeadDivision.h"


// If compiling for the target platform
#if defined(IMPULSE_C_TARGET)
#define XD1000
#endif

extern co_architecture co_initialize(void *);

//static double D[A_NUM];

// Create Random numbers
double msrand( double *seed )
{
   static double a = 16807.;
   static double m = 2147483647.;
   double ret_val;
   static double temp;

   temp = a * *seed;
   *seed = temp - m * (float) ((int) (temp / m));
   ret_val = *seed / m;

   return ret_val;
}



// Initalize Array
void init_array(double *A, double seed)
{
  int j;
  int exp;
  //static double seed;
  extern double msrand();
  //seed = 2.3;
  for (j = 0; j < A_NUM; j++) {
       A[j] = msrand(&seed);  //get random floating point value based on seed
       exp = (int) 10 * msrand(&seed);  // use msrand to get a random float (between 0 and 1), multiply by
10, then truncate to int
       A[j] *= pow(-1,exp);  // exponentiate -1 to random exp
  }
  return;
}



// Clear Array
void clear_array(double *A)
{
  int j;
  for (j = 0; j < A_NUM; j++){
     A[j] = 0.0;
  }
```

```
}


/* Reference implementation of xGEMM
 * Runs on the Opteron CPU for comparison with the FPGA implementation
 */
void get_ref(double *A, double *B, double *D){
   double sum;
   int i;
   for (i = 0; i < A_NUM; i++){
     sum = 0;
     sum = A[i] / B[i];
     D[i] = sum;
   }
}



/* Verify that two results are equal (difference in every element <=
 * test_threshold)
 */
int errorcheck(double *C, double *D, float test_threshold)
{
  float relerror;
  int j;
  for ( j=0; j< A_NUM; j++) {
       relerror = fabs(C[j] - D[j]) / fabs(D[j]);
       if (relerror > test_threshold) {
          printf("error at point %d \n", j);
          printf("check threshold \n");
          printf("CPU ref %f    FPGA calc %f \n",D[j] , C[j]);
                                fflush(0);
          return(-1);
       }
  }
  return(0);
}



#ifdef XD1000
/* Return current time from an OS timer.
 * Link with -lrt for clock_gettime (you must modify the generated Makefile)
 */
struct timespec curr_time()
{
            struct timespec ts;
            clock_gettime(CLOCK_REALTIME, &ts);
            return ts;
}
#define TIME_MINUS(a,b) (((a).tv_sec-(b).tv_sec)+(double)((a).tv_nsec-(b).tv_nsec)/1000000000.0)
#endif



/* Software process, runs on Opteron
```

```
 *
 * Initializes memory with array values, then runs the reference software and
 * FPGA-accelerated hardware versions.  Compares the output of both
 * implementations and prints timing results.
 */
void call_fpga(co_memory imgmem, co_signal start, co_signal end)
{
   uint32 data;
   int j, k;
        int ierr;
        double *A,*B,*C, *D;                // A, B, C, D are the arrays that will be used.
        float test_threshold = 1.0e-5;
        float time_measured_fpga, time_measured_cpu;

#ifdef XD1000
        struct timespec now, then;
#endif
        IF_SIM(cosim_logwindow log = cosim_logwindow_create("call_fpga");)

   //Create arrays
   A = (double *) malloc((A_NUM)*sizeof(double));   // Operand array A
   if(A==NULL) printf("ERROR!\n");                  // Test if memory is properly allocated
   B = (double *) malloc((B_NUM)*sizeof(double));   // Operand array B
   if(B==NULL) printf("ERROR!\n");
   C = (double *) malloc((A_NUM)*sizeof(double));   // Result array C from FPGA calculation
   if(C==NULL) printf("ERROR!\n");
   D = (double *) malloc((A_NUM)*sizeof(double));   // Result array D from CPU calculation
   if(D==NULL) printf("ERROR!\n");

   // Initalize operands array
        init_array(A, 2.3);
        init_array(B, 3.2);
   clear_array(C);
   clear_array(D);

   printf("the first few values of array A are %f  %f  %f  %f\n", A[0], A[1], A[A_NUM-2], A[A_NUM-1]);
   printf("\nthe first few values of array B are %f  %f  %f  %f\n", B[0], B[1], B[A_NUM-2], B[A_NUM-
1]);

#ifndef XD1000
        printf("Timing statistics only calculated on XD1000 system\n");
#endif

   printf("Running reference (CPU)...\r\n");

#ifdef XD1000
        now = curr_time();
#endif

   // Get reference values from the CPU - do N_ITERATIONS times
        for ( j = 0 ; j < N_ITERATION ; j++)
     get_ref(A, B, D);

#ifdef XD1000
        then = curr_time();
        time_measured_cpu = (float)TIME_MINUS(then, now);
```

```
            printf("time: %fs\n", time_measured_cpu);
    #endif
        // Print out a few examples
            printf("%f %f %f %f\n", (float)D[0],(float)D[1],(float)D[2],(float)D[3]);
            printf("%f %f %f %f\r\n", (float)D[4],(float)D[5],(float)D[6],(float)D[7]);
            printf("%f %f %f %f\n", (float)D[A_NUM-8], (float)D[A_NUM-7], (float)D[A_NUM-6],
(float)D[A_NUM-5]);
            printf("%f %f %f %f\r\n",(float)D[A_NUM-4], (float)D[A_NUM-3], (float)D[A_NUM-2],
(float)D[A_NUM-1]);

        printf("Running test (FPGA)...\r\n");

            // Write the A, B, and C matrices to the SRAM
            // The FPGA will wait for a signal, then read/write matrices in the SRAM
            // while computing the product.
            co_memory_writeblock(imgmem, 0, A, (A_NUM)*sizeof(double));
            co_memory_writeblock(imgmem, (A_NUM)*sizeof(double), B, (B_NUM)*sizeof(double));
            co_memory_writeblock(imgmem, ((A_NUM)+(B_NUM))*sizeof(double), C,
(A_NUM)*sizeof(double));


    #ifdef XD1000
            now = curr_time();
    #endif

        for ( k = 0; k < N_ITERATION; k++){
            // Signal the FPGA to start, then wait for the "done" signal
                co_signal_post(start, 0);
                // FPGA computes result...
                co_signal_wait(end, (co_int32*)&data);
        }

    #ifdef XD1000
            then = curr_time();
            time_measured_fpga = (float)TIME_MINUS(then, now);
            printf("time: %fs\n", time_measured_fpga);
            printf("speedup: %fx performance of CPU\n\n", time_measured_cpu/time_measured_fpga);
    #endif


            // Read the FPGA-computed result out of SRAM into 'c' for verification
            co_memory_readblock(imgmem, ((A_NUM)+(B_NUM))*sizeof(double), C,
A_NUM*sizeof(double));

        // Print out a few examples
            printf("%f %f %f %f\n", (float)C[0],(float)C[1],(float)C[2],(float)C[3]);
            printf("%f %f %f %f\r\n", (float)C[4],(float)C[5],(float)C[6],(float)C[7]);
        printf("%f %f %f %f\r\n", (float)C[128],(float)C[129],(float)C[130],(float)C[131]);
        printf("%f %f %f %f\r\n", (float)C[252],(float)C[253],(float)C[254],(float)C[255]);
            printf("%f %f %f %f\n", (float)C[A_NUM-8], (float)C[A_NUM-7], (float)C[A_NUM-6],
(float)C[A_NUM-5]);
            printf("%f %f %f %f\r\n",(float)C[A_NUM-4], (float)C[A_NUM-3], (float)C[A_NUM-2],
(float)C[A_NUM-1]);

        printf("Checking result ...\r\n");
            ierr = errorcheck(C, D, test_threshold);
```

```
        if (ierr==0) printf("Passed.\r\n");
}


int main(int argc, char *argv[])
{
        IF_SIM(int c;)
        co_architecture my_arch;

        printf("Division Test\r\n");
        printf("---------------\n");

        // Get list of processes to execute from the configuration function in
        // Division_hw.c (desktop simulation) or from co_init.c (host software).
        my_arch = co_initialize(NULL);
        // Execute processes
        co_execute(my_arch);

        IF_SIM(printf("Press Enter key to continue...\n");)
        IF_SIM(c = getc(stdin);)

        return(0);
}
```

3. Division_hw.c

```
/****

XtremeData Division Testbench
Authors: John Rothermel & Tri Dang
Version 1
Date: Nov 26, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C.

****/

#include <stdio.h>
#include "co.h"
#include "cosim_log.h"
#include "HeadDivision.h"

// Software process, defined in mmult_sw.c
extern void call_fpga(co_memory datamem, co_signal start, co_signal end);

// Define a wide element type: bits = N_UNROLLED * bits(NUMBER)
// Evidently, this element type can store 8 double precision number.
#define WIDE co_uint512
typedef uint64 WIDE[UN_ROLLED];

// Define macros for moving individual floating-point numbers in/out of the
// WIDE type.  Different code used in hardware generation (#ifdef
// IMPULSE_C_SYNTHESIS) and in simulation.
//
// These macros are written for double-precision numbers and must be modified
// to do SGEMM.

#ifdef IMPULSE_C_SYNTHESIS
```

```
#undef co_bit_insert
#define GETELM(x,i) to_double((x)>>(448-64*(i)))
#define PUTELM(x,i,e) x=co_bit_insert(x,448-64*(i),64,double_bits(e))
#define ASSIGN(lhs,rhs) lhs=rhs

#else

#define GETELM(x,i) to_double(x[i])
#define PUTELM(x,i,e) x[i]=double_bits(e)
#define ASSIGN(lhs,rhs) memcpy(lhs,rhs,UN_ROLLED*sizeof(double))

#endif

#ifdef IMPULSE_C_SYNTHESIS
#define DECLMEM
#else
#define DECLMEM static
#endif

 // SLOT is the constant that define how many element required for array Ain
#define SLOT MEM_TRANS/UN_ROLLED
// CHUNK is the number of chunks required to process to complete the testbench
#define CHUNK A_NUM/MEM_TRANS
// Shared memories (FPGA block RAMs) used by ioproc and mmproc to store
// array A and C.
static WIDE Ain[2*SLOT], Cout[2*SLOT], Bin[2*SLOT];

// # of chunks of A read. A Chunk is defined as one piece of memory that is transfer to
// the FPGA for calculation at one time, which is represented as variable MEM_TRANS.
// In specific case where we have 2^20 numbers and trade in 2048 numbers at a time, 512 chunks
// is required to complete the calculation.
static uint32 nRead;
// # of chunks of A processed so far by mmproc.  This variable is shared
// by ioproc and mmproc to synchronize memory access.
static uint32 nDone;

/* Read inputs A and store output C in parallel with the
 * computation done by mmproc.  For example, while mmproc is computing
 * A[1] / B[1], ioproc is reading A[2] and B[2] and storing
 * the results from A[0] / B[0] back into SRAM.
 */
void ioproc(co_memory datamem, co_signal start, co_signal complete)
{
        uint32 j;
        uint32 offA, offC, offB;
   uint32 cHalf, aHalf;          // cHalf and aHalf is used to keep track of which chunk will be used
        int32 data;
   IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop
        IF_SIM(cosim_logwindow log = cosim_logwindow_create("ioproc");)

        do {  // One matrix multiplication per iteration
                offA = 0;
      offB = (A_NUM) * sizeof(double);
                offC = (A_NUM + B_NUM) * sizeof(double);
```

```
        nRead = 0;                      // nRead is also need to be reset so that mmprocess and ioprocess are at
the same index
                        // Wait for mmproc to read matrix B
                        co_signal_wait(start,&data);

                        // Preload two chunks of A
                        co_memory_readblock(datamem, offA, Ain, 2 * SLOT * sizeof(WIDE));
                        offA += 2 * SLOT * sizeof(WIDE);
        co_memory_readblock(datamem, offB, Bin, 2 * SLOT * sizeof(WIDE));
        offB += 2 * SLOT * sizeof(WIDE);
        nRead = 2;

                        j=0;
                        do {
                                while (nDone<=j);  // Wait for mmproc to compute new results
          if ((j&1)==0) {   // if j is even, take the first chunk that is stored in the array
            aHalf = cHalf = 0;
          } else {          // if j is odd, take the second chunk
            aHalf = SLOT;
            cHalf = SLOT;
          }
                                // Store a chunk of results back to C in SRAM
                                co_memory_writeblock(datamem, offC, &Cout[cHalf] , SLOT *
sizeof(WIDE));
                                offC += SLOT * sizeof(WIDE);                // Move to the next chunk address
                                j++;
                                if (j == CHUNK) break;  // Done with entire computation
                                // Read another row each of A and C for the next computation
                                co_memory_readblock(datamem, offA, &Ain[aHalf], SLOT * sizeof(WIDE));
                                offA += SLOT * sizeof(WIDE);
        co_memory_readblock(datamem, offB, &Bin[aHalf], SLOT * sizeof(WIDE));
        offB += SLOT * sizeof(WIDE);
                                nRead++;
                        } while (1);
                        co_signal_post(complete,j);  // Signal CPU that computation is done
                        //IF_SIM(break;)
                        IF_SIM(if(!(--loopcnt)) break;)
          } while (1);  // Always running in hardware
    }


    /* Computes C = A + B.  Access to array A in shared block RAMs is
     * synchronized with ioproc by global variables.  Matrix B is stored entirely
     * in block RAM local to this process.
     */
    void mmproc(co_memory datamem, co_signal go, co_signal startio)
    {
       uint32 offA, offB, data;
       uint32 Half ;        //Again to keep track of which chunk will be used
              double m;
       int j;               // keep track of the chunks
       int k;               // A's index
       //DECLMEM WIDE Bin[B_NUM/UN_ROLLED];
              WIDE aval, bval;     // buffer for one element of array A and B
              WIDE cres;          // buffer for one result element
       IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop
```

```
  //DECLMEM WIDE ctmp[C_NUM];
  IF_SIM(cosim_logwindow log = cosim_logwindow_create("mmproc");)

  // Array B is implemented by default in an Altera FPGA as a dual-port
  // synchronous RAM.
  //co_array_config(B,co_kind,"dualsync");

  do {  // One Addition per iteration
        offA = 0;
        offB = offA+(A_NUM * sizeof(double));


        // Wait for the CPU to store A and C in SRAM
        co_signal_wait(go,(int32*)&data);
        // Compute C := A+B
        //
        // Read all of B from SRAM into local memory
        //co_memory_readblock(datamem, offB, Bin, B_NUM * sizeof(double));

                        /////
        // EAT (Edward Trexel) 11/30/2007 --
        // Add co_par_break() to force co_signal_wait() and co_signal_post() into separate
        //  stages.  Sometimes, even if the signals are uinrelated and won't cause a deadlock
        //  situation with another process, the co_signal_post() won't occur as expected.
        //  Previously in the DGEMM code, the co_memory_readblock() was creating this
separation.
        co_par_break();
        /////

        // Tell ioproc it can start its overlapping IO to/from A and C
        co_signal_post(startio,data);
        nDone= 0;

        for (j=0; j < CHUNK; j++) {
                while (nRead <= j);  // Wait for data to be made available by ioproc
    if ((j&1) == 0){     // First chunk is selected when j is even
      Half = 0;
    } else {            // Second chunk is selected when j is odd
       Half = SLOT;
    }
    k = 0;              // k keep track of index in one chunk
                do {
      #pragma co pipeline
                        //IF_NSIM(cres=0;)
       ASSIGN(aval, Ain[k + Half]);
       ASSIGN(bval, Bin[k + Half]);
                        // Add UN_ROLLED floating-point values.
                        // The Impulse C compiler will schedule all these operations so
                        // they execute in a single clock cycle.
                        PUTELM(cres,0,GETELM(aval,0)  /  GETELM(bval,0));
                        PUTELM(cres,1,GETELM(aval,1)  /  GETELM(bval,1));
                        PUTELM(cres,2,GETELM(aval,2)  /  GETELM(bval,2));
                        PUTELM(cres,3,GETELM(aval,3)  /  GETELM(bval,3));
                        PUTELM(cres,4,GETELM(aval,4)  /  GETELM(bval,4));
```

```
                                        PUTELM(cres,5,GETELM(aval,5)  /  GETELM(bval,5));
                                        PUTELM(cres,6,GETELM(aval,6)  /  GETELM(bval,6));
                                        PUTELM(cres,7,GETELM(aval,7)  /  GETELM(bval,7));
                                        //PUTELM(cres,8,GETELM(aval,8)  /  GETELM(bval,8));
                                        //PUTELM(cres,9,GETELM(aval,9)  /  GETELM(bval,9));
                                        //PUTELM(cres,10,GETELM(aval,10)/  GETELM(bval,10));
                                        //PUTELM(cres,11,GETELM(aval,11)/  GETELM(bval,11));
                                        //PUTELM(cres,12,GETELM(aval,12)/  GETELM(bval,12));
                                        //PUTELM(cres,13,GETELM(aval,13)/  GETELM(bval,13));
                                        //PUTELM(cres,14,GETELM(aval,14)/  GETELM(bval,14));
                                        //PUTELM(cres,15,GETELM(aval,15)/  GETELM(bval,15));

/*
 * Commented out for UN_ROLLED == 16.  The FPGA has enough multiplier
 * resources to do 32 parallel double-precision MACs, but the Quartus II tools
 * cannot aggregate the various block RAMs on the device into a memory wide
 * enough to feed all 32 operators in parallel.  The memory would have to be:
 *   UN_ROLLED * sizeof(double) * 8 =
 *                   32 * 8 * 8 = 2048 bits wide
 * Unrolling by 32 may work for single-precision Addition (SGEMM)
 *
                                        PUTELM(cres,16,GETELM(aval,16)+GETELM(bval,16));
                                        PUTELM(cres,17,GETELM(aval,17)+GETELM(bval,17));
                                        PUTELM(cres,18,GETELM(aval,18)+GETELM(bval,18));
                                        PUTELM(cres,19,GETELM(aval,19)+GETELM(bval,19));
                                        PUTELM(cres,20,GETELM(aval,20)+GETELM(bval,20));
                                        PUTELM(cres,21,GETELM(aval,21)+GETELM(bval,21));
                                        PUTELM(cres,22,GETELM(aval,22)+GETELM(bval,22));
                                        PUTELM(cres,23,GETELM(aval,23)+GETELM(bval,23));
                                        PUTELM(cres,24,GETELM(aval,24)+GETELM(bval,24));
                                        PUTELM(cres,25,GETELM(aval,25)+GETELM(bval,25));
                                        PUTELM(cres,26,GETELM(aval,26)+GETELM(bval,26));
                                        PUTELM(cres,27,GETELM(aval,27)+GETELM(bval,27));
                                        PUTELM(cres,28,GETELM(aval,28)+GETELM(bval,28));
                                        PUTELM(cres,29,GETELM(aval,29)+GETELM(bval,29));
                                        PUTELM(cres,30,GETELM(aval,30)+GETELM(bval,30));
                                        PUTELM(cres,31,GETELM(aval,31)+GETELM(bval,31));
*/
                                        ASSIGN(Cout[k + Half],cres);
                                        k++;
                                } while (k < (SLOT));
                                nDone++;
                        } // end j
                        //IF_SIM(break;)
                        IF_SIM(if(!(--loopcnt)) break;)
                } while (1);  // Always running in hardware
}



// The following two functions implement a required Impulse C pattern
// Impulse C configuration function
// Describes all processes and the IO objects that connect them
void config_div(void *arg)
{
        co_signal startsig, donesig, iosig;
```

```
        co_memory shrmem;
        co_process cpu_proc, mm, io_proc;

        startsig = co_signal_create("start");
        donesig  = co_signal_create("done");
        iosig  = co_signal_create("iosig");
        // Associate the co_memory object with the default physical memory location
        // for the platform (second argument, "").  In the XD1000's case, this
        // default location is the QDR SRAM, where all matrices are stored.
        shrmem   = co_memory_create("data", "", ((A_NUM)+(B_NUM)+(A_NUM))*sizeof(double));

        cpu_proc = co_process_create("cpu_proc",(co_function)call_fpga, 3, shrmem, startsig, donesig);
        mm = co_process_create("mm",(co_function)mmproc,3,shrmem,startsig,iosig);
        io_proc = co_process_create("io_proc",(co_function)ioproc,3,shrmem,iosig,donesig);

        // Assign mmproc and ioproc to run as FPGA hardware
        co_process_config(mm, co_loc, "PE0");
        co_process_config(io_proc, co_loc, "PE0");
}

// Boilerplate code that tells the Impulse C hardware compiler to look at the
// config_add function for the application's "layout" of processes and IOs.
// In simulation, this code gives the simulation library a pointer to the
// configuration function so the library can execute it internally.
co_architecture co_initialize()
{
   return(co_architecture_create("mmex", "fpga", config_div, NULL));
}
```

B.7: Sine Benchmark

1. HeadSine.h

```
//This file provides the number of operations needed to test.
//This benchtest will deal with double precision floating point number
//A_NUM = 2^17
//Variable mem_trans defines the amount of memory is transfer for each memory transfer command in hardware
// memory (bits) = mem_trans (# of doble precision number) * 64 (64 bits in 1 double precision number).
// Better explantion for this variable is in Hardware code.

#define A_NUM 131072
#define MEM_TRANS 1024
#define UN_ROLLED 8
// This variable determine how many times the arrays will be recalculated
#define N_ITERATION 300
```

2. Sine_sw.c

```
/***

XtremeData Sine Testbench
Authors: John Rothermel & Tri Dang
Version 1
Date: Nov 28, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C
```

```
***/

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <time.h>
#include "co.h"
#include "cosim_log.h"
#include "HeadSine.h"

#define PI 3.141592654


// If compiling for the target platform
#if defined(IMPULSE_C_TARGET)
#define XD1000
#endif

extern co_architecture co_initialize(void *);

//static double D[A_NUM];

// Create Random numbers
double msrand( double *seed )
{
   static double a = 16807.;
   static double m = 2147483647.;
   double ret_val;
   static double temp;

   temp = a * *seed;
   *seed = temp - m * (float) ((int) (temp / m));
   ret_val = *seed / m;

   return ret_val;
}



// Initalize Array
void init_array(double *A, double seed)
{
  int j, exp;
  double a;
  //static double seed;
  extern double msrand();
  //seed = 2.3;
  for (j = 0; j < A_NUM; j++) {
       a = msrand(&seed);
       exp = (int) 10*msrand(&seed);  // use msrand to get a random float (between 0 and 1), multiply by 10,
then truncate to int
       //a *= pow(-1,exp)*100;  // exponentiate -1 to random exp and increase the range
       if(abs(a) >= (PI/2)) {
         A[j] = pow(-1,exp)*fmod(a,PI/2);   // normalize to (-PI/2 PI/2)
       }
```

```
        else A[j] = pow(-1,exp)*a;
    }
    return;
}



// Clear Array
void clear_array(double *A)
{
    int j;
    for (j = 0; j < A_NUM; j++){
        A[j] = 0.0;
    }
}


/* Reference implementation of xGEMM
 * Runs on the Opteron CPU for comparison with the FPGA implementation
 */
void get_ref(double *A, double *D){
    double sum;
    int i;
    for (i = 0; i < A_NUM; i++){
        sum = 0;
        sum = sin(A[i]);
        D[i] = sum;
    }
}



/* Verify that two results are equal (difference in every element <=
 * test_threshold)
 */
int errorcheck(double *C, double *D, float test_threshold)
{
    float relerror;
    int j;
    for ( j=0; j< A_NUM; j++) {
        relerror = fabs(C[j] - D[j]) / fabs(D[j]);
        if (relerror > test_threshold) {
            printf("error at point %d \n", j);
            printf("check threshold \n");
            printf("CPU ref %f    FPGA calc %f \n",D[j] , C[j]);
                                    fflush(0);
            return(-1);
        }
    }
    return(0);
}



#ifdef XD1000
/* Return current time from an OS timer.
```

```
 * Link with -lrt for clock_gettime (you must modify the generated Makefile)
 */
struct timespec curr_time()
{
        struct timespec ts;
        clock_gettime(CLOCK_REALTIME, &ts);
        return ts;
}
#define TIME_MINUS(a,b) (((a).tv_sec-(b).tv_sec)+(double)((a).tv_nsec-(b).tv_nsec)/1000000000.0)
#endif



/* Software process, runs on Opteron
 *
 * Initializes memory with array values, then runs the reference software and
 * FPGA-accelerated hardware versions.  Compares the output of both
 * implementations and prints timing results.
 */
void call_fpga(co_memory imgmem, co_signal start, co_signal end)
{
   uint32 data;
   int j, k;
        int ierr;
        double *A,*C, *D;               // A, C, D are the arrays that will be used.
        float test_threshold = 1.0e-7;
        float time_measured_fpga, time_measured_cpu;

#ifdef XD1000
        struct timespec now, then;
#endif
        IF_SIM(cosim_logwindow log = cosim_logwindow_create("call_fpga");)

   //Create arrays
   A = (double *) malloc((A_NUM)*sizeof(double));   // Operand array A
   if(A==NULL) printf("ERROR!\n");                  // Test if memory is properly allocated
   C = (double *) malloc((A_NUM)*sizeof(double));   // Result array C from FPGA calculation
   if(C==NULL) printf("ERROR!\n");
   D = (double *) malloc((A_NUM)*sizeof(double));   // Result array D from CPU calculation
   if(D==NULL) printf("ERROR!\n");

   // Initalize operands array
        init_array(A, 2.3);
   A[2] = 0.867 ;
   A[A_NUM-2] = 0;
   clear_array(C);
   clear_array(D);

   printf("the first few values of array A are %f  %f  %f  %f  %f\n", A[0], A[1], A[2], A[A_NUM-2],
A[A_NUM-1]);

#ifndef XD1000
        printf("Timing statistics only calculated on XD1000 system\n");
#endif

   printf("Running reference (CPU)...\r\n");
```

```
	#ifdef XD1000
		now = curr_time();
	#endif

	  // Get reference values from the CPU - do N_ITERATIONS times
		for ( j = 0 ; j < N_ITERATION ; j++)
	    get_ref(A, D);

	#ifdef XD1000
		then = curr_time();
		time_measured_cpu = (float)TIME_MINUS(then, now);
		printf("time: %fs\n", time_measured_cpu);
	#endif
	  // Print out a few examples
		printf("%f %f %f %f\n", (float)D[0],(float)D[1],(float)D[2],(float)D[3]);
		printf("%f %f %f %f\r\n", (float)D[4],(float)D[5],(float)D[6],(float)D[7]);
		printf("%f %f %f %f\n", (float)D[A_NUM-8], (float)D[A_NUM-7], (float)D[A_NUM-6],
(float)D[A_NUM-5]);
		printf("%f %f %f %f\r\n",(float)D[A_NUM-4], (float)D[A_NUM-3], (float)D[A_NUM-2],
(float)D[A_NUM-1]);

	  printf("Running test (FPGA)...\r\n");

		// Write the A and C matrices to the SRAM
		// The FPGA will wait for a signal, then read/write matrices in the SRAM
		// while computing the product.
		co_memory_writeblock(imgmem, 0, A, (A_NUM)*sizeof(double));
		co_memory_writeblock(imgmem, (A_NUM)*sizeof(double), C, (A_NUM)*sizeof(double));


	#ifdef XD1000
		now = curr_time();
	#endif

	  for ( k = 0; k < N_ITERATION; k++){
		// Signal the FPGA to start, then wait for the "done" signal
			co_signal_post(start, 0);
			// FPGA computes result...
			co_signal_wait(end, (co_int32*)&data);
	  }


	#ifdef XD1000
		then = curr_time();
		time_measured_fpga = (float)TIME_MINUS(then, now);
		printf("time: %fs\n", time_measured_fpga);
		printf("speedup: %fx performance of CPU\n\n", time_measured_cpu/time_measured_fpga);
	#endif


		// Read the FPGA-computed result out of SRAM into 'c' for verification
		co_memory_readblock(imgmem, (A_NUM)*sizeof(double), C, A_NUM*sizeof(double));

	  // Print out a few examples
		printf("%f %f %f %f\n", (float)C[0],(float)C[1],(float)C[2],(float)C[3]);
```

```
            printf("%f %f %f %f\r\n", (float)C[4],(float)C[5],(float)C[6],(float)C[7]);
        printf("%f %f %f %f\r\n", (float)C[128],(float)C[129],(float)C[130],(float)C[131]);
        printf("%f %f %f %f\r\n", (float)C[252],(float)C[253],(float)C[254],(float)C[255]);
            printf("%f %f %f %f\n", (float)C[A_NUM-8], (float)C[A_NUM-7], (float)C[A_NUM-6],
(float)C[A_NUM-5]);
            printf("%f %f %f %f\r\n",(float)C[A_NUM-4], (float)C[A_NUM-3], (float)C[A_NUM-2],
(float)C[A_NUM-1]);

        printf("Checking result ...\r\n");
            ierr = errorcheck(C, D, test_threshold);
            if (ierr==0) printf("Passed.\r\n");
    }


    int main(int argc, char *argv[])
    {
            IF_SIM(int c;)
            co_architecture my_arch;

            printf("Sine function Test\r\n");
            printf("----------------\n");

            // Get list of processes to execute from the configuration function in
            // Addition_hw.c (desktop simulation) or from co_init.c (host software).
            my_arch = co_initialize(NULL);
            // Execute processes
            co_execute(my_arch);

            IF_SIM(printf("Press Enter key to continue...\n");)
            IF_SIM(c = getc(stdin);)

            return(0);
    }
```

3. Sine_hw.c

```
    /****

    XtremeData Sine Testbench
    Authors: John Rothermel + Tri Dang
    Version 1
    Date: Nov 28, 2007
    Reference: Double precision Matrix Multiplication project provided by Impulse C.

    ****/

    #include <stdio.h>
    #include "co.h"
    #include "cosim_log.h"
    #include "HeadSine.h"

    // Software process, defined in mmult_sw.c
    extern void call_fpga(co_memory datamem, co_signal start, co_signal end);

    // Define a wide element type: bits = N_UNROLLED * bits(NUMBER)
    // Evidently, this element type can store 16 double precision number.
    #define WIDE co_uint512
```

```
typedef uint64 WIDE[UN_ROLLED];

// Define macros for moving individual floating-point numbers in/out of the
// WIDE type.  Different code used in hardware generation (#ifdef
// IMPULSE_C_SYNTHESIS) and in simulation.
//
// These macros are written for double-precision numbers and must be modified
// to do SGEMM.

#ifdef IMPULSE_C_SYNTHESIS

#undef co_bit_insert
#define GETELM(x,i) to_double((x)>>(448-64*(i)))
#define PUTELM(x,i,e) x=co_bit_insert(x,448-64*(i),64,double_bits(e))
#define ASSIGN(lhs,rhs) lhs=rhs

#else

#define GETELM(x,i) to_double(x[i])
#define PUTELM(x,i,e) x[i]=double_bits(e)
#define ASSIGN(lhs,rhs) memcpy(lhs,rhs,UN_ROLLED*sizeof(double))

#endif

#ifdef IMPULSE_C_SYNTHESIS
#define DECLMEM
#else
#define DECLMEM static
#endif

 // SLOT is the constant that define how many element required for array Ain
#define SLOT MEM_TRANS/UN_ROLLED
// CHUNK is the number of chunks required to process to complete the testbench
#define CHUNK A_NUM/MEM_TRANS
// Shared memories (FPGA block RAMs) used by ioproc and mmproc to store
// array A and C.
static WIDE Ain[2*SLOT], Cout[2*SLOT];

// # of chunks of A read. A Chunk is defined as one piece of memory that is transfer to
// the FPGA for calculation at one time, which is represented as variable MEM_TRANS.
// In specific case where we have 2^20 numbers and trade in 2048 numbers at a time, 512 chunks
// is required to complete the calculation.
static uint32 nRead;
// # of chunks of A processed so far by mmproc.  This variable is shared
// by ioproc and mmproc to synchronize memory access.
static uint32 nDone;

/* Read inputs A and store output C in parallel with the
 * computation done by mmproc.  For example, while mmproc is computing
 * cos(A[1]), ioproc is reading A[2] and storing
 * the results from sqrt(A[0]) back into SRAM.
 */
void ioproc(co_memory datamem, co_signal start, co_signal complete)
{
        uint32 j;
        uint32 offA, offC;
```

```
        uint32 cHalf, aHalf;            // cHalf and aHalf is used to keep track of which chunk will be used
                int32 data;
        IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop
                IF_SIM(cosim_logwindow log = cosim_logwindow_create("ioproc");)

                do {  // One matrix multiplication per iteration
                        offA = 0;
                        offC = (A_NUM) * sizeof(double);
           nRead = 0;

                        // Wait for mmproc to start iosignal
                        co_signal_wait(start,&data);

                        // Preload two chunks of A
                        co_memory_readblock(datamem, offA, Ain, 2 * SLOT * sizeof(WIDE));
                        offA += 2 * SLOT * sizeof(WIDE);
           nRead = 2;

                        j=0;
                        do {
                                while (nDone<=j);  // Wait for mmproc to compute new results
              if ((j&1)==0) {   // if j is even, take the first chunk that is stored in the array
                aHalf = cHalf = 0;
              } else {          // if j is odd, take the second chunk
                aHalf = SLOT;
                cHalf = SLOT;
              }
                                // Store a chunk of results back to C in SRAM
                                co_memory_writeblock(datamem, offC, &Cout[cHalf] , SLOT *
sizeof(WIDE));
                                offC += SLOT * sizeof(WIDE);                // Move to the next chunk address
                                j++;
                                if (j == CHUNK) break;  // Done with entire computation
                                // Read another row each of A and C for the next computation
                                co_memory_readblock(datamem, offA, &Ain[aHalf], SLOT * sizeof(WIDE));
                                offA += SLOT * sizeof(WIDE);
                                nRead++;
                        } while (1);
                        co_signal_post(complete,j);  // Signal CPU that computation is done
                        //IF_SIM(break;)
                        IF_SIM(if(!(--loopcnt)) break;)
                } while (1);  // Always running in hardware
        }

        /* This function defines VHDL sine function. Taylor's series
         *
         */
        double my_sine(double x)
        {

        #pragma CO PRIMITIVE
           double result_sine ;
           int i;
           double x_2 = x*x;
           double coef[] = {1, -0.16666666667, 8.333333333e-3, -1.984126984e-4, 2.755731992e-6}; // reciprocal
coefficients of taylor series, 5 terms
```

```
        double x_term[5] ;                                          // init array to 1.0, recursive for loop, needs
1.0 so it doesn't clear the array
        x_term[0] = x;
        // recursively set the next term in x to its given power, need to do because the tools will not allocate
enough resources to do in parallel
        for(i=1; i<5; i++) {
          x_term[i] = x_term[i-1] * x_2;
        }
        result_sine = 0;
        //result_sine = coef[0]*x_term[0] + coef[1]*x_term[1] + coef[2]*x_term[2] + coef[3]*x_term[3] +
coef[4]*x_term[4];
        for (i = 0; i < 5; i++){
          result_sine += coef[i]*x_term[i];
        }
        return result_sine;
    }

    // double my_sine(double x)
    // {
    // #pragma CO PRIMITIVE
    //     double result_sine;
    //     // recursively set the next term in x to its given power, need to do because the tools will not allocate
enough resources to do in parallel
    //     //result_sine = x + (x*x*x)*coef[1] + (x*x*x*x*x)*coef[2] + (x*x*x*x*x*x*x)*coef[3] +
(x*x*x*x*x*x*x*x*x)*coef[4];
    //     result_sine = x - (x*x*x)/6 + (x*x*x*x*x)/120 - (x*x*x*x*x*x*x)/5040 +
(x*x*x*x*x*x*x*x*x)/362880;  // taylor series x - x^3/3! + x^5/5! - x^7/7! + x^9/9! - x/11!
    //     return result_sine;
    // }


    /* Computes C = sin(A).  Access to array A in shared block RAMs is
     * synchronized with ioproc by global variables.
     */
    void mmproc(co_memory datamem, co_signal go, co_signal startio)
    {
       uint32 offA, offB, data;
       uint32 Half ;        //Again to keep track of which chunk will be used
            double m;
       int j;              // keep track of the chunks
       int k;               // A's index

            WIDE aval;          // buffer for one element of array A
            WIDE cres;          // buffer for one result element
       IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop

            //DECLMEM WIDE ctmp[C_NUM];
            IF_SIM(cosim_logwindow log = cosim_logwindow_create("mmproc");)


            do {  // One sine calculation per iteration
                  offA = 0;
                  offB = offA+(A_NUM * sizeof(double));


                  // Wait for the CPU to store A and C in SRAM
```

```
                co_signal_wait(go,(int32*)&data);

                /////
                // EAT (Edward Trexel) 11/30/2007 --
                // Add co_par_break() to force co_signal_wait() and co_signal_post() into separate
                //  stages.  Sometimes, even if the signals are uinrelated and won't cause a deadlock
                //  situation with another process, the co_signal_post() won't occur as expected.
                //  Previously in the DGEMM code, the co_memory_readblock() was creating this
separation.
                co_par_break();
                /////

                // Compute C := sin(A)
                //
                // Read all of B from SRAM into local memory
                //co_memory_readblock(datamem, offB, Bin, B_NUM * sizeof(double));

                // Tell ioproc it can start its overlapping IO to/from A and C
                co_signal_post(startio,data);
                nDone= 0;

                for (j=0; j < CHUNK; j++) {       //Chunk = 256
                        while (nRead <= j);  // Wait for data to be made available by ioproc
            if ((j&1) == 0){     // First chunk is selected when j is even
              Half = 0;
            } else {            // Second chunk is selected when j is odd
              Half = SLOT;
            }
            k = 0;              // k keep track of index in one chunk
                        do {
              //#pragma co pipeline
                                //IF_NSIM(cres=0;)
                ASSIGN(aval, Ain[k + Half]);
                //ASSIGN(bval, Bin[p]);
                                // Add UN_ROLLED floating-point values.
                                // The Impulse C compiler will schedule all these operations so
                                // they execute in a single clock cycle.
                                PUTELM(cres,0,my_sine(GETELM(aval,0)));
                                PUTELM(cres,1,my_sine(GETELM(aval,1)));
                                PUTELM(cres,2,my_sine(GETELM(aval,2)));
                                PUTELM(cres,3,my_sine(GETELM(aval,3)));
                                PUTELM(cres,4,my_sine(GETELM(aval,4)));
                                PUTELM(cres,5,my_sine(GETELM(aval,5)));
                                PUTELM(cres,6,my_sine(GETELM(aval,6)));
                                PUTELM(cres,7,my_sine(GETELM(aval,7)));
                                //PUTELM(cres,8,my_sine(GETELM(aval,8)));
                                //PUTELM(cres,9,my_sine(GETELM(aval,9)));
                                //PUTELM(cres,10,my_sine(GETELM(aval,10)));
                                //PUTELM(cres,11,my_sine(GETELM(aval,11)));
                                //PUTELM(cres,12,my_sine(GETELM(aval,12)));
                                //PUTELM(cres,13,my_sine(GETELM(aval,13)));
                                //PUTELM(cres,14,my_sine(GETELM(aval,14)));
                                //PUTELM(cres,15,my_sine(GETELM(aval,15)));

                                ASSIGN(Cout[k + Half],cres);
                                k++;
```

```
                        } while (k < (SLOT));
                        nDone++;
                    } // end j
                    //IF_SIM(break;)
                    IF_SIM(if(!(--loopcnt)) break;)
                } while (1);  // Always running in hardware
}


// The following two functions implement a required Impulse C pattern
// Impulse C configuration function
// Describes all processes and the IO objects that connect them
void config_sine(void *arg)
{
        co_signal startsig, donesig, iosig;
        co_memory shrmem;
        co_process cpu_proc, mm, io_proc;

        startsig = co_signal_create("start");
        donesig  = co_signal_create("done");
        iosig  = co_signal_create("iosig");
        // Associate the co_memory object with the default physical memory location
        // for the platform (second argument, "").  In the XD1000's case, this
        // default location is the QDR SRAM, where all matrices are stored.
        shrmem   = co_memory_create("data", "", ((A_NUM)+(A_NUM))*sizeof(double));

        cpu_proc = co_process_create("cpu_proc",(co_function)call_fpga, 3, shrmem, startsig, donesig);
        mm = co_process_create("mm",(co_function)mmproc,3,shrmem,startsig,iosig);
        io_proc = co_process_create("io_proc",(co_function)ioproc,3,shrmem,iosig,donesig);

        // Assign mmproc and ioproc to run as FPGA hardware
        co_process_config(mm, co_loc, "PE0");
        co_process_config(io_proc, co_loc, "PE0");
}

// Boilerplate code that tells the Impulse C hardware compiler to look at the
// config_add function for the application's "layout" of processes and IOs.
// In simulation, this code gives the simulation library a pointer to the
// configuration function so the library can execute it internally.
co_architecture co_initialize()
{
   return(co_architecture_create("mmex", "fpga", config_sine, NULL));
}
```

B.8: Cosine Benchmark Version 1

1. HeadCosine.h

```
//This file provides the number of operations needed to test.
//This benchtest will deal with double precision floating point number
//A_NUM = 2^17
//Variable mem_trans defines the amount of memory is transfer for each memory transfer command in hardware
// memory (bits) = mem_trans (# of doble precision number) * 64 (64 bits in 1 double precision number).
```

```
// Better explantion for this variable is in Hardware code.

#define A_NUM 131072
#define MEM_TRANS 1024
#define UN_ROLLED 8
// This variable determine how many times the arrays will be recalculated
#define N_ITERATION 300
```

2. Cosine_sw.c

```
/***

XtremeData Cosine Testbench
Authors: John Rothermel & Tri Dang
Version 1
Date: Nov 28, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C

***/

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <time.h>
#include "co.h"
#include "cosim_log.h"
#include "HeadCosine.h"

#define PI 3.141592654


// If compiling for the target platform
#if defined(IMPULSE_C_TARGET)
#define XD1000
#endif

extern co_architecture co_initialize(void *);

//static double D[A_NUM];

// Create Random numbers
double msrand( double *seed )
{
   static double a = 16807.;
   static double m = 2147483647.;
   double ret_val;
   static double temp;

   temp = a * *seed;
   *seed = temp - m * (float) ((int) (temp / m));
   ret_val = *seed / m;

   return ret_val;
}
```

```
// Initalize Array
void init_array(double *A, double seed)
{
  int j, exp;
  double a;
  //static double seed;
  extern double msrand();
  //seed = 2.3;
  for (j = 0; j < A_NUM; j++) {
      a = msrand(&seed);
      exp = (int) 10*msrand(&seed);  // use msrand to get a random float (between 0 and 1), multiply by 10,
then truncate to int
      //a *= pow(-1,exp)*100;  // exponentiate -1 to random exp and increase the range
      if(abs(a) >= (PI/2)) {
        A[j] = pow(-1,exp)*fmod(a,PI/2);   // normalize to (-PI/2 PI/2)
      }
      else A[j] = pow(-1,exp)*a;
  }
  return;
}



// Clear Array
void clear_array(double *A)
{
  int j;
  for (j = 0; j < A_NUM; j++){
     A[j] = 0.0;
  }
}


/* Reference implementation of xGEMM
 * Runs on the Opteron CPU for comparison with the FPGA implementation
 */
void get_ref(double *A, double *D){
   double sum;
   int i;
   for (i = 0; i < A_NUM; i++){
     sum = 0;
     sum = cos(A[i]);
     D[i] = sum;
   }
}



/* Verify that two results are equal (difference in every element <=
 * test_threshold)
 */
int errorcheck(double *C, double *D, float test_threshold)
{
  float relerror;
  int j;
  for ( j=0; j< A_NUM; j++) {
```

```
        relerror = fabs(C[j] - D[j]) / fabs(D[j]);
        if (relerror > test_threshold) {
          printf("error at point %d \n", j);
          printf("check threshold \n");
          printf("CPU ref %f    FPGA calc %f \n",D[j] , C[j]);
                                fflush(0);
          return(-1);
        }
    }
    return(0);
}



#ifdef XD1000
/* Return current time from an OS timer.
 * Link with -lrt for clock_gettime (you must modify the generated Makefile)
 */
struct timespec curr_time()
{
            struct timespec ts;
            clock_gettime(CLOCK_REALTIME, &ts);
            return ts;
}
#define TIME_MINUS(a,b) (((a).tv_sec-(b).tv_sec)+(double)((a).tv_nsec-(b).tv_nsec)/1000000000.0)
#endif



/* Software process, runs on Opteron
 *
 * Initializes memory with array values, then runs the reference software and
 * FPGA-accelerated hardware versions.  Compares the output of both
 * implementations and prints timing results.
 */
void call_fpga(co_memory imgmem, co_signal start, co_signal end)
{
    uint32 data;
    int j, k;
            int ierr;
            double *A,*C, *D;                   // A, C, D are the arrays that will be used.
            float test_threshold = 1.0e-5;
            float time_measured_fpga, time_measured_cpu;

#ifdef XD1000
            struct timespec now, then;
#endif
            IF_SIM(cosim_logwindow log = cosim_logwindow_create("call_fpga");)

    //Create arrays
    A = (double *) malloc((A_NUM)*sizeof(double));   // Operand array A
    if(A==NULL) printf("ERROR!\n");                  // Test if memory is properly allocated
    C = (double *) malloc((A_NUM)*sizeof(double));   // Result array C from FPGA calculation
    if(C==NULL) printf("ERROR!\n");
    D = (double *) malloc((A_NUM)*sizeof(double));   // Result array D from CPU calculation
    if(D==NULL) printf("ERROR!\n");
```

```
        // Initalize operands array
                init_array(A, 2.3);
        A[2] = 0.867 ;
        A[A_NUM-2] = 0;
        clear_array(C);
        clear_array(D);

        printf("the first few values of array A are %f  %f  %f  %f  %f\n", A[0], A[1], A[2], A[A_NUM-2],
A[A_NUM-1]);

    #ifndef XD1000
                printf("Timing statistics only calculated on XD1000 system\n");
    #endif

        printf("Running reference (CPU)...\r\n");

    #ifdef XD1000
                now = curr_time();
    #endif

        // Get reference values from the CPU - do N_ITERATIONS times
                for ( j = 0 ; j < N_ITERATION ; j++)
          get_ref(A, D);

    #ifdef XD1000
                then = curr_time();
                time_measured_cpu = (float)TIME_MINUS(then, now);
                printf("time: %fs\n", time_measured_cpu);
    #endif
        // Print out a few examples
                printf("%f %f %f %f\n", (float)D[0],(float)D[1],(float)D[2],(float)D[3]);
                printf("%f %f %f %f\r\n", (float)D[4],(float)D[5],(float)D[6],(float)D[7]);
                printf("%f %f %f %f\n", (float)D[A_NUM-8], (float)D[A_NUM-7], (float)D[A_NUM-6],
(float)D[A_NUM-5]);
                printf("%f %f %f %f\r\n",(float)D[A_NUM-4], (float)D[A_NUM-3], (float)D[A_NUM-2],
(float)D[A_NUM-1]);

        printf("Running test (FPGA)...\r\n");

                // Write the A and C matrices to the SRAM
                // The FPGA will wait for a signal, then read/write matrices in the SRAM
                // while computing the product.
                co_memory_writeblock(imgmem, 0, A, (A_NUM)*sizeof(double));
                co_memory_writeblock(imgmem, (A_NUM)*sizeof(double), C, (A_NUM)*sizeof(double));


    #ifdef XD1000
                now = curr_time();
    #endif

        for ( k = 0; k < N_ITERATION; k++){
                // Signal the FPGA to start, then wait for the "done" signal
                        co_signal_post(start, 0);
                        // FPGA computes result...
                        co_signal_wait(end, (co_int32*)&data);
```

```
    }


#ifdef XD1000
        then = curr_time();
        time_measured_fpga = (float)TIME_MINUS(then, now);
        printf("time: %fs\n", time_measured_fpga);
        printf("speedup: %fx performance of CPU\n\n", time_measured_cpu/time_measured_fpga);
#endif


        // Read the FPGA-computed result out of SRAM into 'c' for verification
        co_memory_readblock(imgmem, (A_NUM)*sizeof(double), C, A_NUM*sizeof(double));

    // Print out a few examples
        printf("%f %f %f %f\n", (float)C[0],(float)C[1],(float)C[2],(float)C[3]);
        printf("%f %f %f %f\r\n", (float)C[4],(float)C[5],(float)C[6],(float)C[7]);
    printf("%f %f %f %f\r\n", (float)C[128],(float)C[129],(float)C[130],(float)C[131]);
    printf("%f %f %f %f\r\n", (float)C[252],(float)C[253],(float)C[254],(float)C[255]);
        printf("%f %f %f %f\n", (float)C[A_NUM-8], (float)C[A_NUM-7], (float)C[A_NUM-6],
(float)C[A_NUM-5]);
        printf("%f %f %f %f\r\n",(float)C[A_NUM-4], (float)C[A_NUM-3], (float)C[A_NUM-2],
(float)C[A_NUM-1]);

    printf("Checking result ...\r\n");
        ierr = errorcheck(C, D, test_threshold);
        if (ierr==0) printf("Passed.\r\n");
}


int main(int argc, char *argv[])
{
        IF_SIM(int c;)
        co_architecture my_arch;

        printf("Cosine function Test\r\n");
        printf("----------------\n");

        // Get list of processes to execute from the configuration function in
        // Addition_hw.c (desktop simulation) or from co_init.c (host software).
        my_arch = co_initialize(NULL);
        // Execute processes
        co_execute(my_arch);

        IF_SIM(printf("Press Enter key to continue...\n");)
        IF_SIM(c = getc(stdin);)

        return(0);
}
```

3. Cosine_hw.c

```
/****

XtremeData Cos Testbench
Authors: John Rothermel + Tri Dang
Version 1
```

Date: Nov 28, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C.

****/

```
#include <stdio.h>
#include "co.h"
#include "cosim_log.h"
#include "HeadCosine.h"

// Software process, defined in mmult_sw.c
extern void call_fpga(co_memory datamem, co_signal start, co_signal end);

// Define a wide element type: bits = N_UNROLLED * bits(NUMBER)
// Evidently, this element type can store 16 double precision number.
#define WIDE co_uint512
typedef uint64 WIDE[UN_ROLLED];

// Define macros for moving individual floating-point numbers in/out of the
// WIDE type.  Different code used in hardware generation (#ifdef
// IMPULSE_C_SYNTHESIS) and in simulation.
//
// These macros are written for double-precision numbers and must be modified
// to do SGEMM.

#ifdef IMPULSE_C_SYNTHESIS

#undef co_bit_insert
#define GETELM(x,i) to_double((x)>>(448-64*(i)))
#define PUTELM(x,i,e) x=co_bit_insert(x,448-64*(i),64,double_bits(e))
#define ASSIGN(lhs,rhs) lhs=rhs

#else

#define GETELM(x,i) to_double(x[i])
#define PUTELM(x,i,e) x[i]=double_bits(e)
#define ASSIGN(lhs,rhs) memcpy(lhs,rhs,UN_ROLLED*sizeof(double))

#endif

#ifdef IMPULSE_C_SYNTHESIS
#define DECLMEM
#else
#define DECLMEM static
#endif

 // SLOT is the constant that define how many element required for array Ain
#define SLOT MEM_TRANS/UN_ROLLED
// CHUNK is the number of chunks required to process to complete the testbench
#define CHUNK A_NUM/MEM_TRANS
// Shared memories (FPGA block RAMs) used by ioproc and mmproc to store
// array A and C.
static WIDE Ain[2*SLOT], Cout[2*SLOT];

// # of chunks of A read. A Chunk is defined as one piece of memory that is transfer to
// the FPGA for calculation at one time, which is represented as variable MEM_TRANS.
```

```
// In specific case where we have 2^20 numbers and trade in 2048 numbers at a time, 512 chunks
// is required to complete the calculation.
static uint32 nRead;
// # of chunks of A processed so far by mmproc.  This variable is shared
// by ioproc and mmproc to synchronize memory access.
static uint32 nDone;

/* Read inputs A and store output C in parallel with the
 * computation done by mmproc.  For example, while mmproc is computing
 * cos(A[1]), ioproc is reading A[2] and storing
 * the results from sqrt(A[0]) back into SRAM.
 */
void ioproc(co_memory datamem, co_signal start, co_signal complete)
{
        uint32 j;
        uint32 offA, offC;
   uint32 cHalf, aHalf;          // cHalf and aHalf is used to keep track of which chunk will be used
        int32 data;
   IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop
        IF_SIM(cosim_logwindow log = cosim_logwindow_create("ioproc");)

        do {  // One matrix multiplication per iteration
                offA = 0;
                offC = (A_NUM) * sizeof(double);
      nRead = 0;

                // Wait for mmproc to start iosignal
                co_signal_wait(start,&data);

                // Preload two chunks of A
                co_memory_readblock(datamem, offA, Ain, 2 * SLOT * sizeof(WIDE));
                offA += 2 * SLOT * sizeof(WIDE);
      nRead = 2;

                j=0;
                do {
                        while (nDone<=j);  // Wait for mmproc to compute new results
          if ((j&1)==0) {   // if j is even, take the first chunk that is stored in the array
           aHalf = cHalf = 0;
          } else {         // if j is odd, take the second chunk
           aHalf = SLOT;
           cHalf = SLOT;
          }
                        // Store a chunk of results back to C in SRAM
                        co_memory_writeblock(datamem, offC, &Cout[cHalf] , SLOT *
sizeof(WIDE));
                        offC += SLOT * sizeof(WIDE);                // Move to the next chunk address
                        j++;
                        if (j == CHUNK) break;  // Done with entire computation
                        // Read another row each of A and C for the next computation
                        co_memory_readblock(datamem, offA, &Ain[aHalf], SLOT * sizeof(WIDE));
                        offA += SLOT * sizeof(WIDE);
                        nRead++;
                } while (1);
                co_signal_post(complete,j);  // Signal CPU that computation is done
                //IF_SIM(break;)
```

```
                    IF_SIM(if(!(--loopcnt)) break;)
                } while (1);  // Always running in hardware
        }

        /* This function defines VHDL sine function. Taylor's series
         *
         */
        // double my_cos(double x)
        // {
        // #pragma CO PRIMITIVE
        //     double result;
        //     // taylor series cos(x) = 1 - x^2/2! + x^4/4! - x^6/6! + x^8/8! - x^10/10! + x^12/12!
        //     result = 1 - (x*x)/2 + (x*x*x*x)/24 - (x*x*x*x*x*x)/720 + (x*x*x*x*x*x*x*x)/40320 -
(x*x*x*x*x*x*x*x*x*x*x*x)/3628800 + (x*x*x*x*x*x*x*x*x*x*x*x*x*x)/479001600;
        //     return result;
        // }

        double my_cos(double x)
        {

        #pragma CO PRIMITIVE
           double result;
           int i;
           double x_2 = x*x;
           // reciprocal coefficients of cosine taylor series, 7 terms
           double coef[] = {1, -0.5, 0.04166666667, -1.388888888889e-3, 2.48015873e-5, 2.755731922e-7,
2.087675699e-9};
           double x_term[7] ;
           x_term[0] = 1;
           // recursively set the next term in x to its given power, need to do because the tools will not allocate
enough resources to do in parallel
           for(i=1; i<7; i++) {
             x_term[i] = x_term[i-1] * x_2;
           }
           result = 0;
           //result = coef[0]*x_term[0] + coef[1]*x_term[1] + coef[2]*x_term[2] + coef[3]*x_term[3] +
coef[4]*x_term[4] \
                  coef[5]*x_term[5] + coef[6]*x_term[6];
           for (i = 0; i < 7; i++){
             result += coef[i]*x_term[i];
           }
           return result;
        }



        /* Computes C = sin(A).  Access to array A in shared block RAMs is
         * synchronized with ioproc by global variables.
         */
        void mmproc(co_memory datamem, co_signal go, co_signal startio)
        {
           uint32 offA, offB, data;
           uint32 Half ;         //Again to keep track of which chunk will be used
                  double m;
           int j;                // keep track of the chunks
           int k;                // A's index
```

```
        WIDE aval;          // buffer for one element of array A
        WIDE cres;          // buffer for one result element
    IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop

        //DECLMEM WIDE ctmp[C_NUM];
        IF_SIM(cosim_logwindow log = cosim_logwindow_create("mmproc");)


        do {  // One sine calculation per iteration
                offA = 0;
                offB = offA+(A_NUM * sizeof(double));


                // Wait for the CPU to store A and C in SRAM
                co_signal_wait(go,(int32*)&data);

                /////
                // EAT (Edward Trexel) 11/30/2007 --
                // Add co_par_break() to force co_signal_wait() and co_signal_post() into separate
                //  stages.  Sometimes, even if the signals are uinrelated and won't cause a deadlock
                //  situation with another process, the co_signal_post() won't occur as expected.
                //  Previously in the DGEMM code, the co_memory_readblock() was creating this
separation.
                co_par_break();
                /////
                // Compute C := cos(A)
                //
                // Read all of B from SRAM into local memory
                //co_memory_readblock(datamem, offB, Bin, B_NUM * sizeof(double));

                // Tell ioproc it can start its overlapping IO to/from A and C
                co_signal_post(startio,data);
                nDone= 0;

                for (j=0; j < CHUNK; j++) {       //Chunk = 256
                        while (nRead <= j);  // Wait for data to be made available by ioproc
            if ((j&1) == 0){     // First chunk is selected when j is even
              Half = 0;
            } else {            // Second chunk is selected when j is odd
              Half = SLOT;
            }
            k = 0;              // k keep track of index in one chunk
                                do {
    //          #pragma co pipeline
                                        //IF_NSIM(cres=0;)
             ASSIGN(aval, Ain[k + Half]);
             //ASSIGN(bval, Bin[p]);
                                        // Add UN_ROLLED floating-point values.
                                        // The Impulse C compiler will schedule all these operations so
                                        // they execute in a single clock cycle.
                                        PUTELM(cres,0,my_cos(GETELM(aval,0)));
                                        PUTELM(cres,1,my_cos(GETELM(aval,1)));
                                        PUTELM(cres,2,my_cos(GETELM(aval,2)));
                                        PUTELM(cres,3,my_cos(GETELM(aval,3)));
                                        PUTELM(cres,4,my_cos(GETELM(aval,4)));
```

```
                                        PUTELM(cres,5,my_cos(GETELM(aval,5)));
                                        PUTELM(cres,6,my_cos(GETELM(aval,6)));
                                        PUTELM(cres,7,my_cos(GETELM(aval,7)));
                                        //PUTELM(cres,8,my_cos(GETELM(aval,8)));
                                        //PUTELM(cres,9,my_cos(GETELM(aval,9)));
                                        //PUTELM(cres,10,my_cos(GETELM(aval,10)));
                                        //PUTELM(cres,11,my_cos(GETELM(aval,11)));
                                        //PUTELM(cres,12,my_cos(GETELM(aval,12)));
                                        //PUTELM(cres,13,my_cos(GETELM(aval,13)));
                                        //PUTELM(cres,14,my_cos(GETELM(aval,14)));
                                        //PUTELM(cres,15,my_cos(GETELM(aval,15)));

                                        ASSIGN(Cout[k + Half],cres);
                                        k++;
                                } while (k < (SLOT));
                                nDone++;
                        } // end j
                        //IF_SIM(break;)
                        IF_SIM(if(!(--loopcnt)) break;)
                } while (1);  // Always running in hardware
}



// The following two functions implement a required Impulse C pattern
// Impulse C configuration function
// Describes all processes and the IO objects that connect them
void config_cos(void *arg)
{
        co_signal startsig, donesig, iosig;
        co_memory shrmem;
        co_process cpu_proc, mm, io_proc;

        startsig = co_signal_create("start");
        donesig  = co_signal_create("done");
        iosig  = co_signal_create("iosig");
        // Associate the co_memory object with the default physical memory location
        // for the platform (second argument, "").  In the XD1000's case, this
        // default location is the QDR SRAM, where all matrices are stored.
        shrmem   = co_memory_create("data", "", ((A_NUM)+(A_NUM))*sizeof(double));

        cpu_proc = co_process_create("cpu_proc",(co_function)call_fpga, 3, shrmem, startsig, donesig);
        mm = co_process_create("mm",(co_function)mmproc,3,shrmem,startsig,iosig);
        io_proc = co_process_create("io_proc",(co_function)ioproc,3,shrmem,iosig,donesig);

        // Assign mmproc and ioproc to run as FPGA hardware
        co_process_config(mm, co_loc, "PE0");
        co_process_config(io_proc, co_loc, "PE0");
}

// Boilerplate code that tells the Impulse C hardware compiler to look at the
// config_add function for the application's "layout" of processes and IOs.
// In simulation, this code gives the simulation library a pointer to the
// configuration function so the library can execute it internally.
co_architecture co_initialize()
{
```

```
    return(co_architecture_create("mmex", "fpga", config_cos, NULL));
}
```

B.9: Cosine Benchmark Version 2

1. Cosinev2.h

```
//This file provides the number of operations needed to test.
//This benchtest will deal with double precision floating point number
//A_NUM = 2^17
//Variable mem_trans defines the amount of memory is transfer for each memory transfer command in hardware
// memory (bits) = mem_trans (# of doble precision number) * 64 (64 bits in 1 double precision number).
// Better explantion for this variable is in Hardware code.

#define A_NUM 131072
#define MEM_TRANS 1024
#define UN_ROLLED 8
// This variable determine how many times the arrays will be recalculated
#define N_ITERATION 300
```

2. Cosinev2_sw.c

```
/***

XtremeData Cosine Testbench
Authors: John Rothermel & Tri Dang
Version 2
Date: Nov 28, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C

***/

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <time.h>
#include "co.h"
#include "cosim_log.h"
#include "Cosinev2.h"

#define PI 3.141592654


// If compiling for the target platform
#if defined(IMPULSE_C_TARGET)
#define XD1000
#endif

extern co_architecture co_initialize(void *);

//static double D[A_NUM];

// Create Random numbers
double msrand( double *seed )
{
```

```
    static double a = 16807.;
    static double m = 2147483647.;
    double ret_val;
    static double temp;

    temp = a * *seed;
    *seed = temp - m * (float) ((int) (temp / m));
    ret_val = *seed / m;

    return ret_val;
}



// Initalize Array
void init_array(double *A, double seed)
{
  int j, exp;
  double a;
  //static double seed;
  extern double msrand();
  //seed = 2.3;
  for (j = 0; j < A_NUM; j++) {
       a = msrand(&seed);
       exp = (int) 10*msrand(&seed);  // use msrand to get a random float (between 0 and 1), multiply by 10,
then truncate to int
       //a *= pow(-1,exp)*100;  // exponentiate -1 to random exp and increase the range
       if(abs(a) >= (PI/2)) {
         A[j] = pow(-1,exp)*fmod(a,PI/2);   // normalize to (-PI/2 PI/2)
       }
       else A[j] = pow(-1,exp)*a;
  }
  return;
}



// Clear Array
void clear_array(double *A)
{
  int j;
  for (j = 0; j < A_NUM; j++){
     A[j] = 0.0;
  }
}


/* Reference implementation of xGEMM
 * Runs on the Opteron CPU for comparison with the FPGA implementation
 */
void get_ref(double *A, double *D){
    double sum;
    int i;
    for (i = 0; i < A_NUM; i++){
      sum = 0;
      sum = cos(A[i]);
```

```
      D[i] = sum;
    }
}



/* Verify that two results are equal (difference in every element <=
 * test_threshold)
 */
int errorcheck(double *C, double *D, float test_threshold)
{
  float relerror;
  int j;
  for ( j=0; j< A_NUM; j++) {
       relerror = fabs(C[j] - D[j]) / fabs(D[j]);
       if (relerror > test_threshold) {
          printf("error at point %d \n", j);
          printf("check threshold \n");
          printf("CPU ref %f    FPGA calc %f \n",D[j] , C[j]);
                                fflush(0);
          return(-1);
       }
  }
  return(0);
}



#ifdef XD1000
/* Return current time from an OS timer.
 * Link with -lrt for clock_gettime (you must modify the generated Makefile)
 */
struct timespec curr_time()
{
            struct timespec ts;
            clock_gettime(CLOCK_REALTIME, &ts);
            return ts;
}
#define TIME_MINUS(a,b) (((a).tv_sec-(b).tv_sec)+(double)((a).tv_nsec-(b).tv_nsec)/1000000000.0)
#endif



/* Software process, runs on Opteron
 *
 * Initializes memory with array values, then runs the reference software and
 * FPGA-accelerated hardware versions.  Compares the output of both
 * implementations and prints timing results.
 */
void call_fpga(co_memory imgmem, co_signal start, co_signal end)
{
   uint32 data;
   int j, k;
            int ierr;
            double *A,*C, *D;                   // A, C, D are the arrays that will be used.
            float test_threshold = 1.0e-5;
```

```
        float time_measured_fpga, time_measured_cpu;

    #ifdef XD1000
        struct timespec now, then;
    #endif
        IF_SIM(cosim_logwindow log = cosim_logwindow_create("call_fpga");)

      //Create arrays
      A = (double *) malloc((A_NUM)*sizeof(double));   // Operand array A
      if(A==NULL) printf("ERROR!\n");                  // Test if memory is properly allocated
      C = (double *) malloc((A_NUM)*sizeof(double));   // Result array C from FPGA calculation
      if(C==NULL) printf("ERROR!\n");
      D = (double *) malloc((A_NUM)*sizeof(double));   // Result array D from CPU calculation
      if(D==NULL) printf("ERROR!\n");

      // Initalize operands array
        init_array(A, 2.3);
      A[2] = 0.867 ;
      A[A_NUM-2] = 0;
      clear_array(C);
      clear_array(D);

      printf("the first few values of array A are %f  %f  %f  %f  %f\n", A[0], A[1], A[2], A[A_NUM-2],
A[A_NUM-1]);

    #ifndef XD1000
        printf("Timing statistics only calculated on XD1000 system\n");
    #endif

      printf("Running reference (CPU)...\r\n");

    #ifdef XD1000
        now = curr_time();
    #endif

      // Get reference values from the CPU - do N_ITERATIONS times
        for ( j = 0 ; j < N_ITERATION ; j++)
        get_ref(A, D);

    #ifdef XD1000
        then = curr_time();
        time_measured_cpu = (float)TIME_MINUS(then, now);
        printf("time: %fs\n", time_measured_cpu);
    #endif
      // Print out a few examples
        printf("%f %f %f %f\n", (float)D[0],(float)D[1],(float)D[2],(float)D[3]);
        printf("%f %f %f %f\r\n", (float)D[4],(float)D[5],(float)D[6],(float)D[7]);
        printf("%f %f %f %f\n", (float)D[A_NUM-8], (float)D[A_NUM-7], (float)D[A_NUM-6],
(float)D[A_NUM-5]);
        printf("%f %f %f %f\r\n",(float)D[A_NUM-4], (float)D[A_NUM-3], (float)D[A_NUM-2],
(float)D[A_NUM-1]);

      printf("Running test (FPGA)...\r\n");

        // Write the A and C matrices to the SRAM
        // The FPGA will wait for a signal, then read/write matrices in the SRAM
```

```
            // while computing the product.
            co_memory_writeblock(imgmem, 0, A, (A_NUM)*sizeof(double));
            co_memory_writeblock(imgmem, (A_NUM)*sizeof(double), C, (A_NUM)*sizeof(double));


    #ifdef XD1000
            now = curr_time();
    #endif

       for ( k = 0; k < N_ITERATION; k++){
            // Signal the FPGA to start, then wait for the "done" signal
                    co_signal_post(start, 0);
                    // FPGA computes result...
                    co_signal_wait(end, (co_int32*)&data);
       }


    #ifdef XD1000
            then = curr_time();
            time_measured_fpga = (float)TIME_MINUS(then, now);
            printf("time: %fs\n", time_measured_fpga);
            printf("speedup: %fx performance of CPU\n\n", time_measured_cpu/time_measured_fpga);
    #endif


            // Read the FPGA-computed result out of SRAM into 'c' for verification
            co_memory_readblock(imgmem, (A_NUM)*sizeof(double), C, A_NUM*sizeof(double));

       // Print out a few examples
            printf("%f %f %f %f\n", (float)C[0],(float)C[1],(float)C[2],(float)C[3]);
            printf("%f %f %f %f\r\n", (float)C[4],(float)C[5],(float)C[6],(float)C[7]);
       printf("%f %f %f %f\r\n", (float)C[128],(float)C[129],(float)C[130],(float)C[131]);
       printf("%f %f %f %f\r\n", (float)C[252],(float)C[253],(float)C[254],(float)C[255]);
            printf("%f %f %f %f\n", (float)C[A_NUM-8], (float)C[A_NUM-7], (float)C[A_NUM-6],
(float)C[A_NUM-5]);
            printf("%f %f %f %f\r\n",(float)C[A_NUM-4], (float)C[A_NUM-3], (float)C[A_NUM-2],
(float)C[A_NUM-1]);

       printf("Checking result ...\r\n");
            ierr = errorcheck(C, D, test_threshold);
            if (ierr==0) printf("Passed.\r\n");
    }


    int main(int argc, char *argv[])
    {
            IF_SIM(int c;)
            co_architecture my_arch;

            printf("Cosine function Test\r\n");
            printf("----------------\n");

            // Get list of processes to execute from the configuration function in
            // Addition_hw.c (desktop simulation) or from co_init.c (host software).
            my_arch = co_initialize(NULL);
            // Execute processes
```

```
        co_execute(my_arch);

        IF_SIM(printf("Press Enter key to continue...\n");)
        IF_SIM(c = getc(stdin);)

        return(0);
}
```

3. Cosinev2_hw.c

```
/****

XtremeData Cos Testbench
Authors: John Rothermel + Tri Dang
Version 2
Date: Nov 28, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C.

****/

#include <stdio.h>
#include "co.h"
#include "cosim_log.h"
#include "Cosinev2.h"

// Software process, defined in mmult_sw.c
extern void call_fpga(co_memory datamem, co_signal start, co_signal end);

// Define a wide element type: bits = N_UNROLLED * bits(NUMBER)
// Evidently, this element type can store 16 double precision number.
#define WIDE co_uint512
typedef uint64 WIDE[UN_ROLLED];

// Define macros for moving individual floating-point numbers in/out of the
// WIDE type.  Different code used in hardware generation (#ifdef
// IMPULSE_C_SYNTHESIS) and in simulation.
//
// These macros are written for double-precision numbers and must be modified
// to do SGEMM.

#ifdef IMPULSE_C_SYNTHESIS

#undef co_bit_insert
#define GETELM(x,i) to_double((x)>>(448-64*(i)))
#define PUTELM(x,i,e) x=co_bit_insert(x,448-64*(i),64,double_bits(e))
#define ASSIGN(lhs,rhs) lhs=rhs

#else

#define GETELM(x,i) to_double(x[i])
#define PUTELM(x,i,e) x[i]=double_bits(e)
#define ASSIGN(lhs,rhs) memcpy(lhs,rhs,UN_ROLLED*sizeof(double))

#endif

#ifdef IMPULSE_C_SYNTHESIS
#define DECLMEM
```

```
#else
#define DECLMEM static
#endif

 // SLOT is the constant that define how many element required for array Ain
#define SLOT MEM_TRANS/UN_ROLLED
// CHUNK is the number of chunks required to process to complete the testbench
#define CHUNK A_NUM/MEM_TRANS
// Shared memories (FPGA block RAMs) used by ioproc and mmproc to store
// array A and C.
static WIDE Ain[2*SLOT], Cout[2*SLOT];

// # of chunks of A read. A Chunk is defined as one piece of memory that is transfer to
// the FPGA for calculation at one time, which is represented as variable MEM_TRANS.
// In specific case where we have 2^20 numbers and trade in 2048 numbers at a time, 512 chunks
// is required to complete the calculation.
static uint32 nRead;
// # of chunks of A processed so far by mmproc.  This variable is shared
// by ioproc and mmproc to synchronize memory access.
static uint32 nDone;

/* Read inputs A and store output C in parallel with the
 * computation done by mmproc.  For example, while mmproc is computing
 * cos(A[1]), ioproc is reading A[2] and storing
 * the results from sqrt(A[0]) back into SRAM.
 */
void ioproc(co_memory datamem, co_signal start, co_signal complete)
{
        uint32 j;
        uint32 offA, offC;
   uint32 cHalf, aHalf;          // cHalf and aHalf is used to keep track of which chunk will be used
        int32 data;
   IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop
        IF_SIM(cosim_logwindow log = cosim_logwindow_create("ioproc");)

        do {  // One matrix multiplication per iteration
                offA = 0;
                offC = (A_NUM) * sizeof(double);
     nRead = 0;

                // Wait for mmproc to start iosignal
                co_signal_wait(start,&data);

                // Preload two chunks of A
                co_memory_readblock(datamem, offA, Ain, 2 * SLOT * sizeof(WIDE));
                offA += 2 * SLOT * sizeof(WIDE);
     nRead = 2;

                j=0;
                do {
                        while (nDone<=j);  // Wait for mmproc to compute new results
       if ((j&1)==0) {   // if j is even, take the first chunk that is stored in the array
         aHalf = cHalf = 0;
       } else {        // if j is odd, take the second chunk
         aHalf = SLOT;
         cHalf = SLOT;
```

```
            }
                                    // Store a chunk of results back to C in SRAM
                                    co_memory_writeblock(datamem, offC, &Cout[cHalf] , SLOT *
sizeof(WIDE));
                                    offC += SLOT * sizeof(WIDE);                // Move to the next chunk address
                                    j++;
                                    if (j == CHUNK) break;  // Done with entire computation
                                    // Read another row each of A and C for the next computation
                                    co_memory_readblock(datamem, offA, &Ain[aHalf], SLOT * sizeof(WIDE));
                                    offA += SLOT * sizeof(WIDE);
                                    nRead++;
                            } while (1);
                            co_signal_post(complete,j);  // Signal CPU that computation is done
                            //IF_SIM(break;)
                            IF_SIM(if(!(--loopcnt)) break;)
                    } while (1);  // Always running in hardware
        }


        double my_cos(double x)
        {

        #pragma CO PRIMITIVE
            double result;
            int i;
            double x_2 = x*x;
            // reciprocal coefficients of cosine taylor series, 7 terms
            double coef[] = {1, -0.5, 0.04166666667, -1.388888888889e-3, 2.48015873e-5, 2.755731922e-7,
2.087675699e-9};
            double x_term[7] ;
            x_term[0] = 1;
            // recursively set the next term in x to its given power, need to do because the tools will not allocate
enough resources to do in parallel
            for(i=1; i<7; i++) {
              x_term[i] = x_term[i-1] * x_2;
            }
            result = 0;
            //result = coef[0]*x_term[0] + coef[1]*x_term[1] + coef[2]*x_term[2] + coef[3]*x_term[3] +
coef[4]*x_term[4] \
                    coef[5]*x_term[5] + coef[6]*x_term[6];
            for (i = 0; i < 7; i++){
              result += coef[i]*x_term[i];
            }
            return result;
        }


        /* Computes C = sin(A).  Access to array A in shared block RAMs is
         * synchronized with ioproc by global variables.
         */
        void mmproc(co_memory datamem, co_signal go, co_signal startio)
        {
            uint32 offA, offB, data;
            uint32 Half ;        //Again to keep track of which chunk will be used
                    double m;
            int j;              // keep track of the chunks
```

```
int k;              // A's index

        WIDE aval;          // buffer for one element of array A
        WIDE cres;          // buffer for one result element
IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop

        //DECLMEM WIDE ctmp[C_NUM];
        IF_SIM(cosim_logwindow log = cosim_logwindow_create("mmproc");)


        do {  // One sine calculation per iteration
                offA = 0;
                offB = offA+(A_NUM * sizeof(double));


                // Wait for the CPU to store A and C in SRAM
                co_signal_wait(go,(int32*)&data);

                /////
                // EAT (Edward Trexel) 11/30/2007 --
                // Add co_par_break() to force co_signal_wait() and co_signal_post() into separate
                //  stages.  Sometimes, even if the signals are uinrelated and won't cause a deadlock
                //  situation with another process, the co_signal_post() won't occur as expected.
                //  Previously in the DGEMM code, the co_memory_readblock() was creating this
separation.
                co_par_break();
                /////
                // Compute C := cos(A)
                //
                // Read all of B from SRAM into local memory
                //co_memory_readblock(datamem, offB, Bin, B_NUM * sizeof(double));

                // Tell ioproc it can start its overlapping IO to/from A and C
                co_signal_post(startio,data);
                nDone= 0;

                for (j=0; j < CHUNK; j++) {       //Chunk = 256
                        while (nRead <= j);  // Wait for data to be made available by ioproc
          if ((j&1) == 0){     // First chunk is selected when j is even
            Half = 0;
          } else {            // Second chunk is selected when j is odd
            Half = SLOT;
          }
          k = 0;              // k keep track of index in one chunk
                                do {
            #pragma co pipeline
                                        //IF_NSIM(cres=0;)
             ASSIGN(aval, Ain[k + Half]);
             //ASSIGN(bval, Bin[p]);
                                        // Add UN_ROLLED floating-point values.
                                        // The Impulse C compiler will schedule all these operations so
                                        // they execute in a single clock cycle.
                                        PUTELM(cres,0,my_cos(GETELM(aval,0)));
                                        PUTELM(cres,1,my_cos(GETELM(aval,1)));
                                        PUTELM(cres,2,my_cos(GETELM(aval,2)));
                                        PUTELM(cres,3,my_cos(GETELM(aval,3)));
```

```
                                        PUTELM(cres,4,my_cos(GETELM(aval,4)));
                                        PUTELM(cres,5,my_cos(GETELM(aval,5)));
                                        PUTELM(cres,6,my_cos(GETELM(aval,6)));
                                        PUTELM(cres,7,my_cos(GETELM(aval,7)));
                                        //PUTELM(cres,8,my_cos(GETELM(aval,8)));
                                        //PUTELM(cres,9,my_cos(GETELM(aval,9)));
                                        //PUTELM(cres,10,my_cos(GETELM(aval,10)));
                                        //PUTELM(cres,11,my_cos(GETELM(aval,11)));
                                        //PUTELM(cres,12,my_cos(GETELM(aval,12)));
                                        //PUTELM(cres,13,my_cos(GETELM(aval,13)));
                                        //PUTELM(cres,14,my_cos(GETELM(aval,14)));
                                        //PUTELM(cres,15,my_cos(GETELM(aval,15)));

                                        ASSIGN(Cout[k + Half],cres);
                                        k++;
                                } while (k < (SLOT));
                                nDone++;
                        } // end j
                        //IF_SIM(break;)
                        IF_SIM(if(!(--loopcnt)) break;)
                } while (1);  // Always running in hardware
}



// The following two functions implement a required Impulse C pattern
// Impulse C configuration function
// Describes all processes and the IO objects that connect them
void config_cos(void *arg)
{
        co_signal startsig, donesig, iosig;
        co_memory shrmem;
        co_process cpu_proc, mm, io_proc;

        startsig = co_signal_create("start");
        donesig  = co_signal_create("done");
        iosig  = co_signal_create("iosig");
        // Associate the co_memory object with the default physical memory location
        // for the platform (second argument, "").  In the XD1000's case, this
        // default location is the QDR SRAM, where all matrices are stored.
        shrmem   = co_memory_create("data", "", ((A_NUM)+(A_NUM))*sizeof(double));

        cpu_proc = co_process_create("cpu_proc",(co_function)call_fpga, 3, shrmem, startsig, donesig);
        mm = co_process_create("mm",(co_function)mmproc,3,shrmem,startsig,iosig);
        io_proc = co_process_create("io_proc",(co_function)ioproc,3,shrmem,iosig,donesig);

        // Assign mmproc and ioproc to run as FPGA hardware
        co_process_config(mm, co_loc, "PE0");
        co_process_config(io_proc, co_loc, "PE0");
}

// Boilerplate code that tells the Impulse C hardware compiler to look at the
// config_add function for the application's "layout" of processes and IOs.
// In simulation, this code gives the simulation library a pointer to the
// configuration function so the library can execute it internally.
co_architecture co_initialize()
```

```
{
   return(co_architecture_create("mmex", "fpga", config_cos, NULL));
}
```

B.10: Natural Logarithm Benchmark

1. Head_Ln.h

```
//This file provides the number of operations needed to test.
//This benchtest will deal with double precision floating point number
//A_NUM = 2^17
//Variable mem_trans defines the amount of memory is transfer for each memory transfer command in hardware
// memory (bits) = mem_trans (# of doble precision number) * 64 (64 bits in 1 double precision number).
// Better explantion for this variable is in Hardware code.

#define A_NUM 131072
#define MEM_TRANS 2048
#define UN_ROLLED 8
// This variable determine how many times the arrays will be recalculated
#define N_ITERATION 300
```

2. Ln_sw.c

```
/***

XtremeData Sine Testbench
Authors: John Rothermel & Tri Dang
Version 1
Date: Nov 28, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C

***/

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <time.h>
#include "co.h"
#include "cosim_log.h"
#include "Head_Ln.h"

#define PI 3.141592654


// If compiling for the target platform
#if defined(IMPULSE_C_TARGET)
#define XD1000
#endif

extern co_architecture co_initialize(void *);

//static double D[A_NUM];

// Create Random numbers
double msrand( double *seed )
```

```
{
   static double a = 16807.;
   static double m = 2147483647.;
   double ret_val;
   static double temp;

   temp = a * *seed;
   *seed = temp - m * (float) ((int) (temp / m));
   ret_val = *seed / m;

   return ret_val;
}



// Initalize Array
void init_array(double *A, double seed)
{
  int j, exp;
  double a;
  //static double seed;
  extern double msrand();
  //seed = 2.3;
  for (j = 0; j < A_NUM; j++) {
       a = msrand(&seed);
       exp = (int) 100*msrand(&seed);  // use msrand to get a random float (between 0 and 1), multiply by
10, then truncate to int
       A[j] = exp+a;                 // Only postive real number
  }
  return;
}



// Clear Array
void clear_array(double *A)
{
  int j;
  for (j = 0; j < A_NUM; j++){
     A[j] = 0.0;
  }
}


/* Reference implementation of xGEMM
 * Runs on the Opteron CPU for comparison with the FPGA implementation
 */
void get_ref(double *A, double *D){
   double sum;
   int i;
   for (i = 0; i < A_NUM; i++){
     sum = 0;
     sum = log(A[i]);    // take natural log
     D[i] = sum;
   }
}
```

```
/* Verify that two results are equal (difference in every element <=
 * test_threshold)
 */
int errorcheck(double *C, double *D, float test_threshold)
{
  float relerror;
  int j;
  for ( j=0; j< A_NUM; j++) {
       relerror = fabs(C[j] - D[j]) / fabs(D[j]);
       if (relerror > test_threshold) {
          printf("error at point %d \n", j);
          printf("check threshold \n");
          printf("CPU ref %f    FPGA calc %f \n",D[j] , C[j]);
                                fflush(0);
          return(-1);
       }
  }
  return(0);
}



#ifdef XD1000
/* Return current time from an OS timer.
 * Link with -lrt for clock_gettime (you must modify the generated Makefile)
 */
struct timespec curr_time()
{
            struct timespec ts;
            clock_gettime(CLOCK_REALTIME, &ts);
            return ts;
}
#define TIME_MINUS(a,b) (((a).tv_sec-(b).tv_sec)+(double)((a).tv_nsec-(b).tv_nsec)/1000000000.0)
#endif



/* Software process, runs on Opteron
 *
 * Initializes memory with array values, then runs the reference software and
 * FPGA-accelerated hardware versions.  Compares the output of both
 * implementations and prints timing results.
 */
void call_fpga(co_memory imgmem, co_signal start, co_signal end)
{
   uint32 data;
   int j, k;
            int ierr;
            double *A,*C, *D;                   // A, C, D are the arrays that will be used.
            float test_threshold = 1.0e-3;       // only accurate to 0.1% relative error
            float time_measured_fpga, time_measured_cpu;

#ifdef XD1000
```

```
            struct timespec now, then;
    #endif
            IF_SIM(cosim_logwindow log = cosim_logwindow_create("call_fpga");)

       //Create arrays
       A = (double *) malloc((A_NUM)*sizeof(double));   // Operand array A
       if(A==NULL) printf("ERROR!\n");                  // Test if memory is properly allocated
       C = (double *) malloc((A_NUM)*sizeof(double));   // Result array C from FPGA calculation
       if(C==NULL) printf("ERROR!\n");
       D = (double *) malloc((A_NUM)*sizeof(double));   // Result array D from CPU calculation
       if(D==NULL) printf("ERROR!\n");

       // Initalize operands array
            init_array(A, 2.3);
       A[2] = 0.0000000000867 ;
       A[A_NUM-2] = 0;
       clear_array(C);
       clear_array(D);

       printf("the first few values of array A are %f  %f  %f  %f  %f\n", A[0], A[1], A[2], A[A_NUM-2],
A[A_NUM-1]);

    #ifndef XD1000
            printf("Timing statistics only calculated on XD1000 system\n");
    #endif

       printf("Running reference (CPU)...\r\n");

    #ifdef XD1000
            now = curr_time();
    #endif

       // Get reference values from the CPU - do N_ITERATIONS times
            for ( j = 0 ; j < N_ITERATION ; j++)
         get_ref(A, D);

    #ifdef XD1000
            then = curr_time();
            time_measured_cpu = (float)TIME_MINUS(then, now);
            printf("time: %fs\n", time_measured_cpu);
    #endif
       // Print out a few examples
            printf("%f %f %f %f\n", (float)D[0],(float)D[1],(float)D[2],(float)D[3]);
            printf("%f %f %f %f\r\n", (float)D[4],(float)D[5],(float)D[6],(float)D[7]);
            printf("%f %f %f %f\n", (float)D[A_NUM-8], (float)D[A_NUM-7], (float)D[A_NUM-6],
(float)D[A_NUM-5]);
            printf("%f %f %f %f\r\n",(float)D[A_NUM-4], (float)D[A_NUM-3], (float)D[A_NUM-2],
(float)D[A_NUM-1]);

       printf("Running test (FPGA)...\r\n");

            // Write the A and C matrices to the SRAM
            // The FPGA will wait for a signal, then read/write matrices in the SRAM
            // while computing the product.
            co_memory_writeblock(imgmem, 0, A, (A_NUM)*sizeof(double));
            co_memory_writeblock(imgmem, (A_NUM)*sizeof(double), C, (A_NUM)*sizeof(double));
```

```
	#ifdef XD1000
		now = curr_time();
	#endif

	   for ( k = 0; k < N_ITERATION; k++){
		// Signal the FPGA to start, then wait for the "done" signal
			co_signal_post(start, 0);
			// FPGA computes result...
			co_signal_wait(end, (co_int32*)&data);
	   }


	#ifdef XD1000
		then = curr_time();
		time_measured_fpga = (float)TIME_MINUS(then, now);
		printf("time: %fs\n", time_measured_fpga);
		printf("speedup: %fx performance of CPU\n\n", time_measured_cpu/time_measured_fpga);
	#endif


		// Read the FPGA-computed result out of SRAM into 'c' for verification
		co_memory_readblock(imgmem, (A_NUM)*sizeof(double), C, A_NUM*sizeof(double));

	   // Print out a few examples
		printf("%f %f %f %f\n", (float)C[0],(float)C[1],(float)C[2],(float)C[3]);
		printf("%f %f %f %f\r\n", (float)C[4],(float)C[5],(float)C[6],(float)C[7]);
	   printf("%f %f %f %f\r\n", (float)C[128],(float)C[129],(float)C[130],(float)C[131]);
	   printf("%f %f %f %f\r\n", (float)C[252],(float)C[253],(float)C[254],(float)C[255]);
		printf("%f %f %f %f\n", (float)C[A_NUM-8], (float)C[A_NUM-7], (float)C[A_NUM-6],
(float)C[A_NUM-5]);
		printf("%f %f %f %f\r\n",(float)C[A_NUM-4], (float)C[A_NUM-3], (float)C[A_NUM-2],
(float)C[A_NUM-1]);

	   printf("Checking result ...\r\n");
		ierr = errorcheck(C, D, test_threshold);
		if (ierr==0) printf("Passed.\r\n");
	}


	int main(int argc, char *argv[])
	{
		IF_SIM(int c;)
		co_architecture my_arch;

		printf("Natual logarithm function Test\r\n");
		printf("----------------\n");

		// Get list of processes to execute from the configuration function in
		// Addition_hw.c (desktop simulation) or from co_init.c (host software).
		my_arch = co_initialize(NULL);
		// Execute processes
		co_execute(my_arch);

		IF_SIM(printf("Press Enter key to continue...\n");)
```

```
        IF_SIM(c = getc(stdin);)

        return(0);
}
```

3. Ln_hw.c

```
/****

XtremeData Sine Testbench
Authors: John Rothermel + Tri Dang
Version 1
Date: Nov 28, 2007
Reference: Double precision Matrix Multiplication project provided by Impulse C.

****/

#include <stdio.h>
#include "co.h"
#include "cosim_log.h"
#include "Head_Ln.h"

// Software process, defined in mmult_sw.c
extern void call_fpga(co_memory datamem, co_signal start, co_signal end);

// Define a wide element type: bits = N_UNROLLED * bits(NUMBER)
// Evidently, this element type can store 16 double precision number.
#define WIDE co_uint512
typedef uint64 WIDE[UN_ROLLED];

// Define macros for moving individual floating-point numbers in/out of the
// WIDE type.  Different code used in hardware generation (#ifdef
// IMPULSE_C_SYNTHESIS) and in simulation.
//
// These macros are written for double-precision numbers and must be modified
// to do SGEMM.

#ifdef IMPULSE_C_SYNTHESIS

#undef co_bit_insert
#define GETELM(x,i) to_double((x)>>(448-64*(i)))
#define PUTELM(x,i,e) x=co_bit_insert(x,448-64*(i),64,double_bits(e))
#define ASSIGN(lhs,rhs) lhs=rhs

#else

#define GETELM(x,i) to_double(x[i])
#define PUTELM(x,i,e) x[i]=double_bits(e)
#define ASSIGN(lhs,rhs) memcpy(lhs,rhs,UN_ROLLED*sizeof(double))

#endif

#ifdef IMPULSE_C_SYNTHESIS
#define DECLMEM
#else
#define DECLMEM static
#endif
```

```
 // SLOT is the constant that define how many element required for array Ain
#define SLOT MEM_TRANS/UN_ROLLED
// CHUNK is the number of chunks required to process to complete the testbench
#define CHUNK A_NUM/MEM_TRANS
// Shared memories (FPGA block RAMs) used by ioproc and mmproc to store
// array A and C.
static WIDE Ain[2*SLOT], Cout[2*SLOT];

// # of chunks of A read. A Chunk is defined as one piece of memory that is transfer to
// the FPGA for calculation at one time, which is represented as variable MEM_TRANS.
// In specific case where we have 2^20 numbers and trade in 2048 numbers at a time, 512 chunks
// is required to complete the calculation.
static uint32 nRead;
// # of chunks of A processed so far by mmproc.  This variable is shared
// by ioproc and mmproc to synchronize memory access.
static uint32 nDone;

/* Read inputs A and store output C in parallel with the
 * computation done by mmproc.  For example, while mmproc is computing
 * cos(A[1]), ioproc is reading A[2] and storing
 * the results from sqrt(A[0]) back into SRAM.
 */
void ioproc(co_memory datamem, co_signal start, co_signal complete)
{
        uint32 j;
        uint32 offA, offC;
   uint32 cHalf, aHalf;          // cHalf and aHalf is used to keep track of which chunk will be used
        int32 data;
   IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop
        IF_SIM(cosim_logwindow log = cosim_logwindow_create("ioproc");)

        do {  // One matrix multiplication per iteration
                offA = 0;
                offC = (A_NUM) * sizeof(double);
     nRead = 0;

                // Wait for mmproc to start iosignal
                co_signal_wait(start,&data);

                // Preload two chunks of A
                co_memory_readblock(datamem, offA, Ain, 2 * SLOT * sizeof(WIDE));
                offA += 2 * SLOT * sizeof(WIDE);
     nRead = 2;

                j=0;
                do {
                        while (nDone<=j);  // Wait for mmproc to compute new results
       if ((j&1)==0) {   // if j is even, take the first chunk that is stored in the array
         aHalf = cHalf = 0;
       } else {          // if j is odd, take the second chunk
         aHalf = SLOT;
         cHalf = SLOT;
       }
                        // Store a chunk of results back to C in SRAM
```

```
                                co_memory_writeblock(datamem, offC, &Cout[cHalf] , SLOT *
sizeof(WIDE));
                                offC += SLOT * sizeof(WIDE);                // Move to the next chunk address
                                j++;
                                if (j == CHUNK) break;  // Done with entire computation
                                // Read another row each of A and C for the next computation
                                co_memory_readblock(datamem, offA, &Ain[aHalf], SLOT * sizeof(WIDE));
                                offA += SLOT * sizeof(WIDE);
                                nRead++;
                        } while (1);
                        co_signal_post(complete,j);  // Signal CPU that computation is done
                        //IF_SIM(break;)
                        IF_SIM(if(!(--loopcnt)) break;)
                } while (1);  // Always running in hardware
        }

        /* These functions define VHDL ln() function. Due to some unknow post synthesis issues, calling multiple
         * primitive functions within a primitive function must be avoided in this version of Impulse C, 3a.11
         * To calculate natural logarithm, apply this equation: lg(x)/lg(e) = ln(x)
         */

        double my_ln(double x)
        {
        #pragma CO PRIMITIVE
           co_int64 tmp;
           double exp;
           co_uint64 tmp2;
           double mantissa;
           double invalid;
           double tmp1 ;
           double result;
           double man;
           int i;
           // init array
           double x_term[6];
                   // coefficients array, 6 terms
                   double coef[] = {1.4425449290, -0.7181451002, 0.4575485901, -0.2779042655, 0.1217970128, -
0.0258411662};

           invalid = to_double(0xFFF0000000000000LL);
           if ( x <= 0) return invalid;  //if non positive input return invalid value

           // First obtain exponent value
           tmp = double_bits(x);                        // get bits of x, store in tmp for manipulation
           tmp = (tmp & 0x7FFFFFFFFFFFFFFFLL) >> 52;     // get rid of sign bit then shift right to get only
exponent
           tmp = (tmp - 0x3FF);                         // Obtain the correct value by subtracting the bias exponent, 1023 (
3FF in hex)
           exp = (double) tmp;
           //exp = FX2REAL64(tmp,0);                    // This command might be the cause for the FPGA to hang up
... check co_math.h ... casting double
                                        // type to tmp does not work on the FPGA side for this version of
Codeveloper 3a.11

           // Then obtain mantissa value
           tmp2 = double_bits(x);
```

```
        // "and" operation to obtain the mantissa bit and insert the hidden bit "1" to obtain the correct mantissa
        tmp2 = (tmp2 & 0x000FFFFFFFFFFFFLL) + 0x3FF0000000000000LL;
        mantissa = to_double(tmp2);                             // algorithm approximates lg(x+1) so subtract 1

        // Calculate result
        man = mantissa - 1;
        x_term[0] = man;
        // recursively set the next term in x to its given power, need to do because the tools will not allocate
enough resources to do in parallel
        for(i=1; i<6; i++){
          x_term[i] = x_term[i-1] * man;
        }
        tmp1 = 0;
        // This approximation is obtained through forum of Google group called Digital Signal Processing using
computer
        // Author : robert bristow-johnson
        // Available online: groups.google.com/group.dsp/msg/8ba6bd6fe2474876
        //lg(1+man) =          (1.4425449290 * man)\
                    +       (-0.7181451002 * man*man)\
                    +        (0.4575485901 * man*man*man)\
                    +       (-0.2779042655 * man*man*man*man)\
                    +        (0.1217970128 * man*man*man*man*man)\
                    +       (-0.0258411662 * man*man*man*man*man*man);
        for (i = 0; i < 6; i++){
        tmp1 += coef[i]*x_term[i];
        }
        result = (exp + tmp1)*0.693147180559945309417232121458l8;  //divide by lg(e) , binary logarithm of
e
        return result;
    }



    /* Computes C = sin(A).  Access to array A in shared block RAMs is
     * synchronized with ioproc by global variables.
     */
    void mmproc(co_memory datamem, co_signal go, co_signal startio)
    {
        uint32 offA, offB, data;
        uint32 Half ;        //Again to keep track of which chunk will be used
                double m;
        int j;              // keep track of the chunks
        int k;               // A's index

                WIDE aval;          // buffer for one element of array A
                WIDE cres;          // buffer for one result element
        IF_SIM(int loopcnt = N_ITERATION;)   // counter for the loop

                //DECLMEM WIDE ctmp[C_NUM];
                IF_SIM(cosim_logwindow log = cosim_logwindow_create("mmproc");)


                do {  // One sine calculation per iteration
                        offA = 0;
                        offB = offA+(A_NUM * sizeof(double));
```

```
                    // Wait for the CPU to store A and C in SRAM
                    co_signal_wait(go,(int32*)&data);

                    /////
                    // EAT (Edward Trexel) 11/30/2007 --
                    // Add co_par_break() to force co_signal_wait() and co_signal_post() into separate
                    //  stages.  Sometimes, even if the signals are uinrelated and won't cause a deadlock
                    //  situation with another process, the co_signal_post() won't occur as expected.
                    //  Previously in the DGEMM code, the co_memory_readblock() was creating this
separation.
                    co_par_break();
                    /////
                    // Compute C := sin(A)
                    //
                    // Read all of B from SRAM into local memory
                    //co_memory_readblock(datamem, offB, Bin, B_NUM * sizeof(double));

                    // Tell ioproc it can start its overlapping IO to/from A and C
                    co_signal_post(startio,data);

          nDone= 0;
                    for (j=0; j < CHUNK; j++) {       //Chunk = 256
                              while (nRead <= j);          // Wait for data to be made available by ioproc
            if ((j&1) == 0){             // First chunk is selected when j is even
              Half = 0;
            } else {                  // Second chunk is selected when j is odd
              Half = SLOT;
            }
            k = 0;                    // k keep track of index in one chunk
                              do {
    //          #pragma co pipeline
                                        //IF_NSIM(cres=0;)
              ASSIGN(aval, Ain[k + Half]);
              //ASSIGN(bval, Bin[p]);
                                        // Add UN_ROLLED floating-point values.
                                        // The Impulse C compiler will schedule all these operations so
                                        // they execute in a single clock cycle.
                                        PUTELM(cres,0,my_ln(GETELM(aval,0)));
                                        PUTELM(cres,1,my_ln(GETELM(aval,1)));
                                        PUTELM(cres,2,my_ln(GETELM(aval,2)));
                                        PUTELM(cres,3,my_ln(GETELM(aval,3)));
                                        PUTELM(cres,4,my_ln(GETELM(aval,4)));
                                        PUTELM(cres,5,my_ln(GETELM(aval,5)));
                                        PUTELM(cres,6,my_ln(GETELM(aval,6)));
                                        PUTELM(cres,7,my_ln(GETELM(aval,7)));
    //                                  PUTELM(cres,8,my_ln(GETELM(aval,8)));
    //                                  PUTELM(cres,9,my_ln(GETELM(aval,9)));
    //                                  PUTELM(cres,10,my_ln(GETELM(aval,10)));
    //                                  PUTELM(cres,11,my_ln(GETELM(aval,11)));
    //                                  PUTELM(cres,12,my_ln(GETELM(aval,12)));
    //                                  PUTELM(cres,13,my_ln(GETELM(aval,13)));
    //                                  PUTELM(cres,14,my_ln(GETELM(aval,14)));
    //                                  PUTELM(cres,15,my_ln(GETELM(aval,15)));

                                        ASSIGN(Cout[k + Half],cres);
```

```
                                k++;
                            } while (k < (SLOT));
                            nDone++;
                        } // end j
                        //IF_SIM(break;)
                        IF_SIM(if(!(--loopcnt)) break;)
                } while (1);  // Always running in hardware
}



// The following two functions implement a required Impulse C pattern
// Impulse C configuration function
// Describes all processes and the IO objects that connect them
void config_ln(void *arg)
{
        co_signal startsig, donesig, iosig;
        co_memory shrmem;
        co_process cpu_proc, mm, io_proc;

        startsig = co_signal_create("start");
        donesig  = co_signal_create("done");
        iosig  = co_signal_create("iosig");
        // Associate the co_memory object with the default physical memory location
        // for the platform (second argument, "").  In the XD1000's case, this
        // default location is the QDR SRAM, where all matrices are stored.
        shrmem   = co_memory_create("data", "", ((A_NUM)+(A_NUM))*sizeof(double));

        cpu_proc = co_process_create("cpu_proc",(co_function)call_fpga, 3, shrmem, startsig, donesig);
        mm = co_process_create("mm",(co_function)mmproc,3,shrmem,startsig,iosig);
        io_proc = co_process_create("io_proc",(co_function)ioproc,3,shrmem,iosig,donesig);

        // Assign mmproc and ioproc to run as FPGA hardware
        co_process_config(mm, co_loc, "PE0");
        co_process_config(io_proc, co_loc, "PE0");
}

// Boilerplate code that tells the Impulse C hardware compiler to look at the
// config_add function for the application's "layout" of processes and IOs.
// In simulation, this code gives the simulation library a pointer to the
// configuration function so the library can execute it internally.
co_architecture co_initialize()
{
   return(co_architecture_create("mmex", "fpga", config_ln, NULL));
}
```