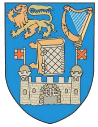


Trinity Centre for
High Performance Computing

Converting code to CUDA

A Cuda Poisson Solver

Jose Refojo

© 2009 TCHPC

Index

- Description of the idea
- Approach
- Implementation
- Results
- Lessons learned
- To do

© 2009 Trinity Centre for High Performance Computing

Description of the idea:

- Transform a simple 2d Poisson solver to cuda
- Start a series of sample code that can be used as a tutorial
- Compare the standard CPU results with the GPU ones

TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Approach

- Try to convert all of the functions so they run in the gpu
- Move all the arrays to the gpu
- Move as little as possible between the system memory and the video card
- Test everything – unit testing approach

TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Changes in structures:

From:

Serial Structure:

```
typedef float Field[NX+2][NY+2];
```

To:

Parallel Structure:

```
typedef float** Field;
```

 TCHPC Confidential © 2009 Trinity Centre for High Performance Computing

But not only that:

To each

```
Field myField=NULL;
```

we add the

```
float *myField_gpu=NULL;
```

that is going to be allocated in the gpu

© 2009 Trinity Centre for High Performance Computing

Implementation

Even more:

We want to allocate a Field as a 1d array, so we use:

```
void setUpField(Field *x, int dim_x, int dim_y) {
        int i,j;
        float *x1d = NULL;
        // x1d is the pointer to all the array malloced in one dimension
        x1d = (float*) malloc( (dim_x)*(dim_y)*sizeof(float) );
        // x will be just pointers to the one dimension array
        *x = (float**) malloc((dim_x)*sizeof(float*));
        for (i=0;i<dim_x;i++) {
                (*x)[i]=(&(x1d[i*dim_x]));
                for (j=0;j<dim_y;j++) {    (*x)[i][j]=0.0f;    }
        }
}
```


© 2009 Trinity Centre for High Performance Computing

And before I forget:

We make sure that a Field is freed correctly:

```
void freeField(Field *x) {
        // First we free the 1d array
        free ( &(*x)[0][0] );
        // Then the 2d array (pointers to pointers)
        free((*x));
}
```


TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Implementation

And we now, to move to the gpu:

We standardise the transfer to the gpu:

```
void passToGpu(Field a, float **a_gpu) {
        float *tmp;
        tmp = &( a[0][0] );
        // Allocate arrays a_gpu
        cudaMalloc ((void **) a_gpu, sizeof(float)*( (NX+2)*(NY+2) ));
        // Copy data from Field to dimension 1 array
        cudaMemcpy(*(a_gpu), tmp, sizeof(float)*( (NX+2)*(NY+2) ),
            cudaMemcpyHostToDevice);
}
```


TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

So finally:

Setting up our Fields and moving them to the gpu is easy!

```
// Initialise and set up
Field a=NULL;setUpField (&a);
float *a_gpu = NULL;
passToGpu(a, &a_gpu);
//
// Other cuda code goes here
//
// Then we free the memory
freeField(&a);
cudaFree(a_gpu);
```


TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Lets compare the mains:

This is how the serial main looks:

```
int main() {
     Field a,b;
    // Initialise them to 0
     zero(a);
     zero(b);
    // Place 1.0 values in the position 250,250 of b
     b[NX/2][NY/2]=1.0;
     solve(b,a);
     print(a);
}
```


TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Implementation

Lets compare the mains:

And this is how the cuda main looks like:

```
int main() {
        Field a=NULL; setUpField (&a); float *a_gpu = NULL;
        Field b=NULL; setUpField (&b); float *b_gpu = NULL;
        zero(a);  zero(b);
        b[NX/2][NY/2]=1.0;
        passToGpu(a, &a_gpu);   passToGpu(b, &b_gpu);
        GPUsolve(b,b_gpu,a,a_gpu);
        solve(b,a);
        freeField(&a); freeField(&b);
        cudaFree(a_gpu); cudaFree(b_gpu);
        return (1);
}
```

Not too bad so far....

© 2009 Trinity Centre for High Performance Computing

So lets have a look at the functions..

- Print() just prints the final result
- Zero() sets all the values of a Field to 0
- Copy() copies one Field into another
- Saxpy() does c = b + alpha*a
- Op() does b[i][k]=a[i][j]-0.25(a[i+1][j]+a[i-1][j]+a[i][j+1]+a[i][j-1])
- Dot() calculates the dot product of two matrices

TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Implementation

So we choose to parallelise:

- Print() just prints the final result
- Zero() sets all the values of a Field to 0
- Copy() copies one Field into another
- Saxpy() does c = b + alpha*a
- Op() does b[i][k]=a[i][j]-0.25(a[i+1][j]+a[i-1][j]+a[i][j+1]+a[i][j-1])
- Dot() calculates the dot product of two matrices

TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Implementation

Copy() is a good one to start with:

This is how the serial version looks like:

```
void copy(Field a, Field b) {
        int x,y;
        for (x=0;x<NX;x++) {
                for (y=0;y<NY;y++) {
                      b[x][y] = a[x][y];
                }
        }
}
```


© 2009 Trinity Centre for High Performance Computing

Copy() is a good one to start with:

Because we have two possibilities, we can do it as a 1d vector:

```
__global__ void GPUcopy1d(float *a, float *b, int Ntotx) {
        int idx=blockIdx.x*blockDim.x+threadIdx.x;

        if ( idx <Ntotx ) {
                b[idx] = a[idx];
        }
}
```

The problem here is that we have more than one dimension in y, we are overwriting the vectors – and that might not be a good idea since we don't know the timing!!

TCHPC Confidential

© 2009 Trinity Centre for High Performance Computing

Implementation

Copy() is a good one to start with:

Or we can do it as a 2d matrix:

```
__global__ void GPUcopy2d(float *a, float *b, int Ntotx, int Ntoty) {
        int idx=blockIdx.x*blockDim.x+threadIdx.x;
        int idy=blockIdx.y*blockDim.y+threadIdx.y;

        if ( idx <Ntotx ) {
                if ( idy <Ntoty ) {
                    b[idx+idy*Ntotx] = a[idx+idy*Ntotx];
                }
        }
}
```


© 2009 Trinity Centre for High Performance Computing

Saxpy() is easy as well:

This is how the serial version looks like:

```
void saxpy(float alpha, Field a, Field b, Field c) {
    int x,y;
    for (x=1;x<NX-1;x++) {
        for (y=1;y<NY-1;y++)
            c[x][y] = b[x][y] + alpha * a[x][y];
        }
    }
}
```


© 2009 Trinity Centre for High Performance Computing

Saxpy() is easy as well:

So the cuda version is:

```
__global__ void GPUsaxpy(float alpha, float *a, float *b, float *c, int Ntotx, int Ntoty) {
        int idx=blockIdx.x*blockDim.x+threadIdx.x;
        int idy=blockIdx.y*blockDim.y+threadIdx.y;
        if ( (idx>0)&&(idx<Ntotx-3) ) {
            if ( (idy>0)&&(idy<Ntoty-3) ) {
                c[idx+idy*Ntotx] = b[idx+idy*Ntotx] + alpha*a[idx+idy*Ntotx];
            }
        }
}
```

The inconvenient here is that it is not clear is alpha is on the GPU

© 2009 Trinity Centre for High Performance Computing

Op() is a little bit longer but:

This is how the serial version looks like:

```
void saxpy(float alpha, Field a, Field b, Field c) {
        int x,y;
        for (x=1;x<NX-1;x++) {
                for (y=1;y<NY-1;y++)
                    c[x][y] = b[x][y] + alpha * a[x][y];
                }
        }
}
```


TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Implementation

Op() is a little bit longer but:

So the GPU version is:

```
__global__ void GPUop(float *a,float *b, int Ntotx, int Ntoty) {
    int idx=blockIdx.x*blockDim.x+threadIdx.x;
    int idy=blockIdx.y*blockDim.y+threadIdx.y;
    if ( idx<NX ) {b[idx]=0.0;b[idx+(NY-1)*Ntotx]=0.0;}
    if ( idy<NY ) {b[idy*Ntotx] = 0.0; b[NX-1+idy*Ntotx] = 0.0; }
    if ( (idx>0)&&(idx<Ntotx-3) ) {
      if ( (idy>0)&&(idy<Ntoty-3) ) {
        b[idx+idy*Ntotx] = a[idx+idy*Ntotx]-0.25*(a[idx+1+idy*Ntotx] + a[idx-1+idy*Ntotx]
            + a[idx+(idy+1)*Ntotx] + a[idx+(idy-1)*Ntotx]);
      }
    }
}
```

We waste some threads but it's not too bad...

TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Dot() is the trickier by far:

This is how the serial version looks like:

```
float dot(Field a, Field b) {
        int x,y;
        float d=0.0;
        for (x=1;x<NX-1;x++) {
            for (y=1;y<NY-1;y++)
                d += a[x][y] * b[x][y];
            }
        }
        return d;
}
```


© 2009 Trinity Centre for High Performance Computing

Dot() is the trickier by far:

Now we have to add up all the elements in a matrix to a single value, and we do not have like a MPI_Reduce that helps us here...

So I went for the following approach:

- Step 1: Add up all the rows in parallel
- Step 2: Use a single thread to add up the vector

But, to do that I had to split the function in two...

© 2009 Trinity Centre for High Performance Computing

Dot() is the trickier by far:

So we have the parallel addition by rows:

```
__global__ void GPUdotArray(float *a, float *b, float *d_array, int Ntotx, int Ntoty) {
int i;
int idx=blockIdx.x*blockDim.x+threadIdx.x;
int idy=blockIdx.y*blockDim.y+threadIdx.y;
        if (idx==0) {
            if ( (idy>0)&&(idy<NX-1) ) {
                d_array[idy]=0.0f;
                for (i=1;i<NX-1;i++) {
                    d_array[idy] += a[i+idy*Ntotx]* b[i+idy*Ntotx];
                }
            }
        }
return;
}
```


TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Implementation

Dot() is the trickier by far:

And then the addition of the row vector:

```
__global__ void GPUdotSum(float *d, float *d_array, int Ntotx) {
        int idx=blockIdx.x*blockDim.x+threadIdx.x;
        int idy=blockIdx.y*blockDim.y+threadIdx.y;
        int i;
        if ( (idx==0)&&(idy==0) ) {
            d[0]=0.0f;
            for (i=0;i<Ntotx;i++) {
                d[0]+=d_array[i];
            }
        }
        return;
}
```


TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Dot() is the trickier by far:

So this is how a dot() call looks now

```
// Compute in rows
GPUdotArray<<<dimGrid,dimBlock>>>(r_gpu,r_gpu, array_gpu, NX+2,NY+2);
// Add up the rows
GPUdotSum<<<dimGrid,dimBlock>>>(myFloat_gpu, array_gpu, NY+2);
// Get the data back from the gpu
cudaMemcpy(&myFloat, myFloat_gpu, sizeof(float), cudaMemcpyDeviceToHost);
dot_r=myFloat;
```

Compared to what it was:

```
dot_r = dot(r,r);
```


© 2009 Trinity Centre for High Performance Computing

Solve() calls all of them:

Solve () starts up 3 additional fields, and calls the other functions:

```
void solve(Field b, Field a) {
 Field r,d,z;  float dot_r,dot_r_old,alpha,beta;  int restart;
 for (restart=0;restart<4;restart++)  {
   op(a,r);  saxpy(-1.0,r,b,r);
   copy(r,d); dot_r = dot(r,r);
   //printf("restart %d: r=%e\n",restart,dot_r);
   do {
     op(d,z);
     alpha = dot_r/ dot(d,z);
     saxpy( alpha,d,a,a);  saxpy(-alpha,z,r,r);
     dot_r_old = dot_r; dot_r = dot(r,r);
     beta = dot_r / dot_r_old;
     saxpy(beta,d,r,d);
     printf("%e\n",dot_r);
   }
   while (dot_r> 1.0e-12);
 }
}
```


© 2009 Trinity Centre for High Performance Computing

Solve() calls all of them:

GPUSolve () starts up 3 additional fields:

```
int i,j;
int restart;
float dot_r,dot_r_old,alpha,beta;
long int old_clock = clock();

Field r=NULL; setUpField (&r);
float *r_gpu = NULL;        passToGpu(r, &r_gpu);
Field d=NULL;setUpField (&d);
float *d_gpu = NULL;        passToGpu(d, &d_gpu);
Field z=NULL;setUpField (&z);
float *z_gpu = NULL;        passToGpu(z, &z_gpu);
```


TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Solve() calls all of them:

Then we have to set up the additional variables for the dot product:

```
float *testArrayOutput;
testArrayOutput = (float *) malloc ((NY+2) * sizeof(float) );
for (i=0;i<(NY+2);i++) {        testArrayOutput[i]=0.0f;  }

float *testArrayOutput_gpu;
cudaMalloc ((void **) &testArrayOutput_gpu, sizeof(float)*(NY+2));
cudaMemcpy(testArrayOutput_gpu, testArrayOutput, sizeof(float)*(NY+2),
    cudaMemcpyHostToDevice);

float testFloatOutput=0.0f;
float *testFloatOutput_gpu;
cudaMalloc ((void **) &testFloatOutput_gpu, sizeof(float));
cudaMemcpy(testFloatOutput_gpu,&testFloatOutput,sizeof(float), cudaMemcpyHostToDevice);
```


TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Solve() calls all of them:

Then we set up the block and threads:

```
// Compute the execution configuration
int block_size=22;

dim3 dimBlock(block_size,block_size);
dim3 dimGrid ( ((NX+2)/dimBlock.x) + (!((NX+2)%dimBlock.x)?0:1),
            ((NY+2)/dimBlock.y) + (!((NY+2)%dimBlock.y)?0:1) );
```


TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Solve() calls all of them:

Then we start the restart loop:

```
for (restart=0;restart<4;restart++) {
//op(a,r);
GPUop<<<dimGrid,dimBlock>>> (a_gpu, r_gpu, (NX+2), (NY+2));
//saxpy(-1.0,r,b,r);
GPUsaxpy<<<dimGrid,dimBlock>>> (-1.0,r_gpu,b_gpu,r_gpu,(NX+2),(NY+2));
//copy(r,d);
GPUcopy2d<<<dimGrid,dimBlock>>> (r_gpu,d_gpu,(NX+2),(NY+2));
```


© 2009 Trinity Centre for High Performance Computing

Solve() calls all of them:

Then the infamous dot product:

```
//dot_r = dot(r,r);
GPUdotArray<<<dimGrid,dimBlock>>>(r_gpu,r_gpu, testArrayOutput_gpu, NX+2,NY+2);
GPUdotSum<<<dimGrid,dimBlock>>>(testFloatOutput_gpu, testArrayOutput_gpu, NY+2);
cudaMemcpy(&testFloatOutput, testFloatOutput_gpu, sizeof(float), cudaMemcpyDeviceToHost);
dot_r=testFloatOutput;
```


© 2009 Trinity Centre for High Performance Computing

Implementation

Solve() calls all of them:

Then the iterative method starts:

```
do {
//op(d,z);
GPUop<<<dimGrid,dimBlock>>> (d_gpu, z_gpu, (NX+2), (NY+2));
//alpha = dot_r/ dot(d,z);
GPUdotArray<<<dimGrid,dimBlock>>>(d_gpu,z_gpu, testArrayOutput_gpu, NX+2,NY+2);
GPUdotSum<<<dimGrid,dimBlock>>>(testFloatOutput_gpu, testArrayOutput_gpu, NY+2);
cudaMemcpy(&testFloatOutput, testFloatOutput_gpu, sizeof(float), cudaMemcpyDeviceToHost);
alpha = dot_r/ testFloatOutput;
//saxpy( alpha,d,a,a);
GPUsaxpy<<<dimGrid,dimBlock>>> (alpha,d_gpu,a_gpu,a_gpu,(NX+2),(NY+2));
//saxpy(-alpha,z,r,r);
GPUsaxpy<<<dimGrid,dimBlock>>> (-alpha,z_gpu,r_gpu,r_gpu,(NX+2),(NY+2));
```


© 2009 Trinity Centre for High Performance Computing

Solve() calls all of them:

The iterative method finishes:

```
dot_r_old = dot_r;
//dot_r = dot(r,r);
GPUdotArray<<<dimGrid,dimBlock>>>(r_gpu,r_gpu, testArrayOutput_gpu, NX+2,NY+2);
GPUdotSum<<<dimGrid,dimBlock>>>(testFloatOutput_gpu, testArrayOutput_gpu, NY+2);
cudaMemcpy(&testFloatOutput, testFloatOutput_gpu, sizeof(float), cudaMemcpyDeviceToHost);
dot_r=testFloatOutput;
beta = dot_r / dot_r_old;
//saxpy(beta,d,r,d);
GPUsaxpy<<<dimGrid,dimBlock>>> (beta,d_gpu,r_gpu,d_gpu,(NX+2),(NY+2));
}while (dot_r> 1.0e-12);
printf("%e\n",dot_r);
}
```


© 2009 Trinity Centre for High Performance Computing

Solve() calls all of them:

And then we clear up the allocated memory and print the time:

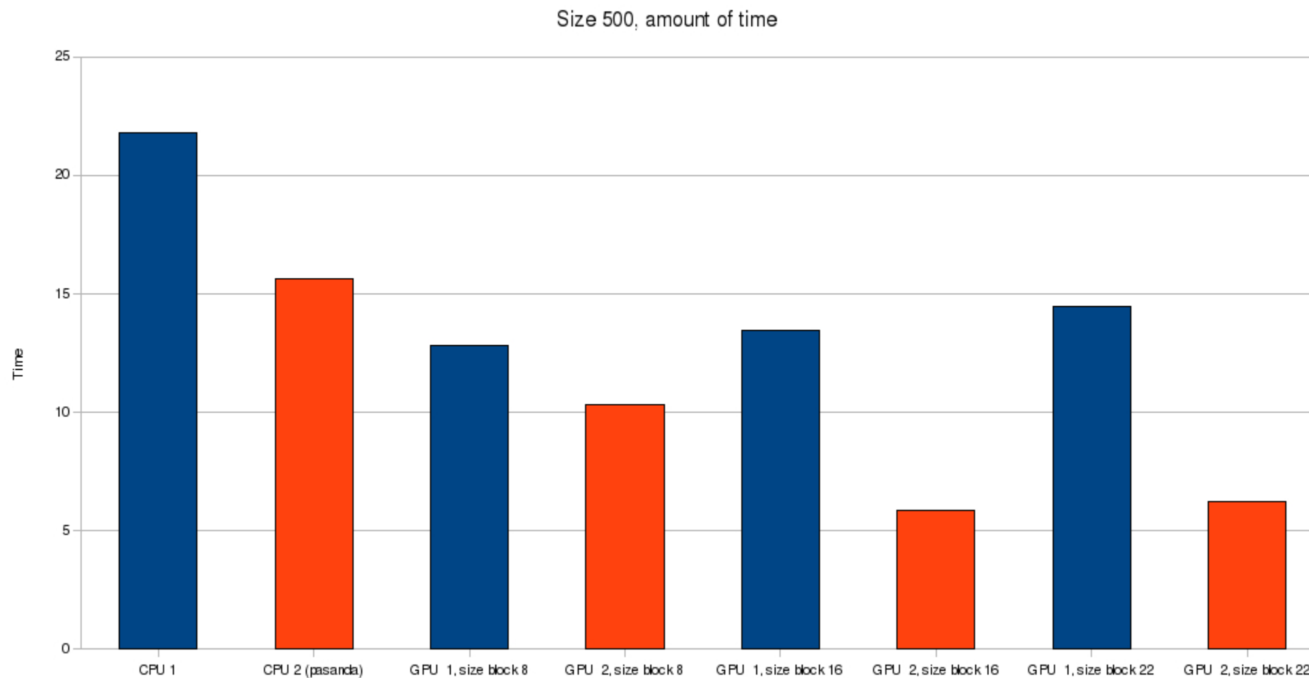
```
freeField(&r);  cudaFree(r_gpu);
freeField(&d); cudaFree(d_gpu);
freeField(&z); cudaFree(z_gpu);

cudaFree(testArrayOutput_gpu);
cudaFree(testFloatOutput_gpu);
printf("GPUsolve total time: %f seconds\n",(float)(clock()-old_clock)*0.000001f );
```


TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Results

So this is a comparison of the performance:



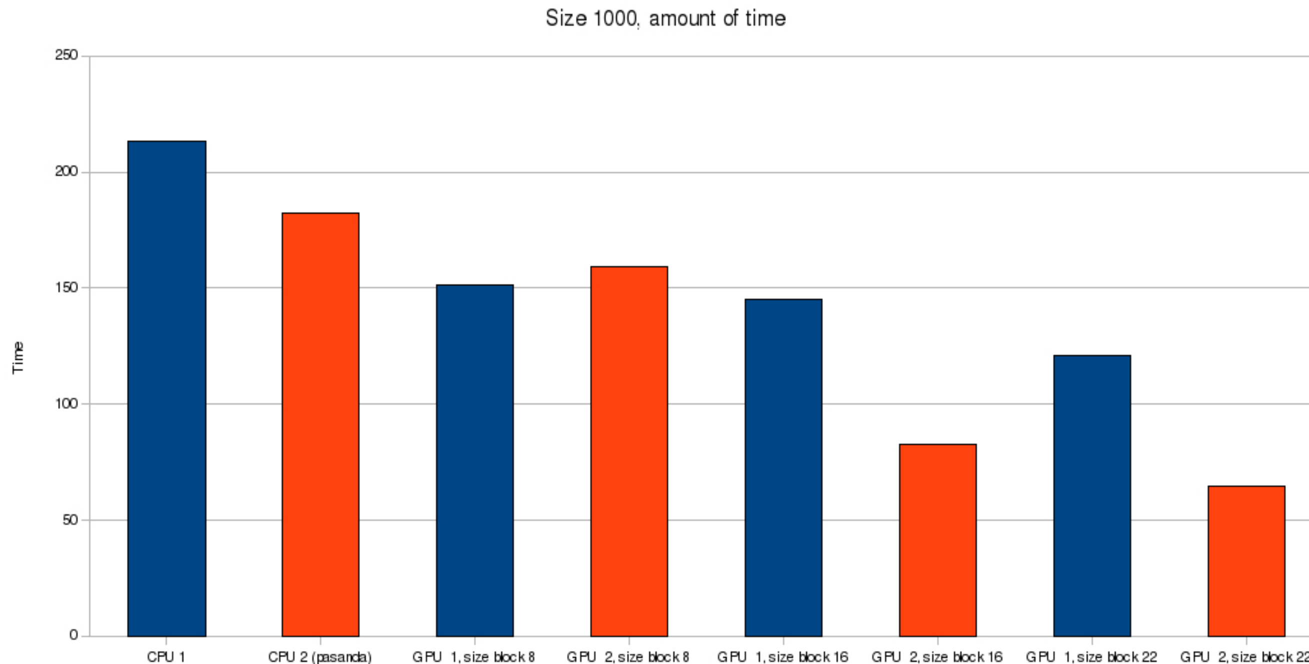
The first machine is a Intel Xeon 3.00 GHz CPU (2048KB cache) CPU with a Nvidia Gforce 9800GTX video card

The second machine is pasanda with a Intel Xeon 2.60 GHz CPU (6144KB cache) CPU and a TESLA C1060 card

TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

Results

So this is a comparison of the performance:



The first machine is a Intel Xeon 3.00 GHz CPU (2048KB cache) CPU with a Nvidia Gforce 9800GTX video card

The second machine is pasanda with a Intel Xeon 2.60 GHz CPU (6144KB cache) CPU and a TESLA C1060 card

© 2009 Trinity Centre for High Performance Computing

Results

Improving the performance:

As you can see in the graphs, using the right block size makes a big difference!

Size	Amount of time compared with the CPU
500	37.36% (pasanda), 47.27% (TCHPC test machine)
1000	35.63% (pasanda), 56.75% (TCHPC test machine)

Just as reference, let's compare the number of multiprocessors in each card:

GPU	Number of Multiprocessors
Geforce 9800GTX	16
Tesla C1060	30

© 2009 Trinity Centre for High Performance Computing

Lessons learned

- Always do a test program for every gpu function that you run
- Very careful with how you distribute the data, as it is easy to forget to update things
- You do not get segmentation faults, so debugging is hard
- Automate all the tasks that you can, as it makes things easier to debug
- It is more complex than OpenMP

TCHPC Confidential
© 2009 Trinity Centre for High Performance Computing

More lessons learned

- Extra attention must be taken with temporary variables that we tend to assume that are re-initialised every time that we call a function
- The Tesla C1060 is certainly more powerful than a Geforce 9800GTX
- Moving memory from GPU to CPU and back takes a lot of time – it added more than 10 seconds in some cases!
- I need more practice implementing cuda code!!

© 2009 Trinity Centre for High Performance Computing

To do

- Move all the data to the shared memory: 3 copies pf each dataset... What could possibly go wrong??
- Not bringing the result of the dot product back to the cpu
- Test memory coalescing
- Testing more about the block sizes – how does the performance improve or decrease when we take a divisor? Should we adjust it in runtime to improve the performance?

© 2009 Trinity Centre for High Performance Computing

Thank you!

Website:

www.tchpc.tcd.ie

Email:

jose@tchpc.tcd.ie

© 2009 Trinity Centre for High Performance Computing