ECE 6374 PARALLEL COMPUTATIONS

Project 3 Report

By

Bijon Sahu(1647781)

&

Manoj Kumar Cebol Sundarrajan(1620546)

Date

04/30/2018

Matrix transpose and multiplication are fundamental building block for scientific computing. Moreover, the algorithmic patterns of matrix multiplication are representative. Many other algorithms share similar optimization techniques as matrix multiplication. Therefore, matrix multiplication is one of the most important examples in learning parallel programming.

**The matrixMul Problem**

Given an M x K matrix A and a K x N matrix B, multiply A with B and store the result into a M x N matrix C.

The matrixMul example on this page will show several techniques to optimize matrix multiplication on GPU. Most of them are generic, which can be applied to other applications.

**These techniques are:**

1. Tiling

2. Memory coalescing

3. Avoiding memory bank conflicts

4. Computation Optimization.

5. Loop unrolling

6. Prefetching

**Below are the types of optimization performed as part of the project 3 for MATRIX MULTIPLICATION.**

**Type of Multiplication Optimization:**

1. **Naïve CPU :**This is the normal CPU matrix multiplication with the below mentioned code .

**Code:**

void main(){

define A, B, C

for i = 0 to M do

for j = 0 to N do

/\* compute element C(i,j) \*/

for k = 0 to K do

C(i,j) <= C(i,j) + A(i,k) \* B(k,j)

end

end

end

}

1. **Naïve GPU :** A naive implementation on GPUs assigns one thread to compute one element of matrix C. Each thread loads one row of matrix A and one column of matrix B from global memory, do the inner product, and store the result back to matrix C in the global memory.

**Code :**

\_\_global\_\_ void matrixMul(A\_gpu,B\_gpu,C\_gpu,K){

temp <= 0

i <= blockIdx.y \* blockDim.y + threadIdx.y // Row i of matrix C

j <= blockIdx.x \* blockDim.x + threadIdx.x // Column j of matrix C

for k = 0 to K-1 do

accu <= accu + A\_gpu(i,k) \* B\_gpu(k,j)

end

C\_gpu(i,j) <= accu

}

**3. Cache GPU :** The blocking size can be determined using the formula

*Where B is the block size. Cache size here in our case is 32KB.*

Solving,

It is found that Block Size for int(2 byte) is 32 elements .

The implemented for loop is the below code :

\_\_global\_\_ void cacheMatmul(float \*a, float \*b, float \*c, int n)

{

int i = blockIdx.x \* blockDim.x + threadIdx.x;

int j = blockIdx.y \* blockDim.y + threadIdx.y;

float acc = 0;

for(int k1=0;k1<n;k1+=gridDim.x)

{

acc=c[i\*n+j];

for(int k=k1;k<k1+gridDim.x;k++)

{

acc += a[i\*n+k] \* b[k\*n+j];

}

c[i\*n+j] = acc;

}

}

1. **Optimal GPU :** This is the most optimized code which includes the concept of shared memeory , tilting and bank conflicts .

**Code:**

\_\_global\_\_ void sharedMatmul(float \*a, float \*b, float \*c, int n)

{

\_\_shared\_\_ float A\_tile[32][32];

\_\_shared\_\_ float B\_tile[32][32];

int width = gridDim.x\*blockDim.x;

float acc = 0;

int i = blockIdx.x\*32 + threadIdx.x;

int j = blockIdx.y\*32 + threadIdx.y;

/\* Accumulate C tile by tile. \*/

for (int tileIdx = 0; tileIdx < gridDim.x ; tileIdx+=1)

{

/\* Load one tile of A and one tile of B into shared mem \*/

A\_tile[threadIdx.y][ threadIdx.x] = a[j \* width + tileIdx\*32+threadIdx.x];

B\_tile[threadIdx.y][threadIdx.x] = b[(tileIdx \* 32 + threadIdx.y)\* width+ i ];

\_\_syncthreads();

/\* Accumulate one tile of C from tiles of A and B in shared mem \*/

for (int k = 0 ;k < 32; k++)

{

acc += A\_tile[threadIdx.y][k] \* B\_tile[k][threadIdx.x];

}

\_\_syncthreads();

}

c[j \* width + i ] = acc;

}

Below is the result after optimizing and running the program in **opuntia.cacds.uh.edu (mcebolsu)** cluster .

**[mcebolsu@compute-0-40 project3]$ ./matmul**

Size 512 naive CPU: 99.915520 ms

Size 512 naive GPU: 35.404350 ms

Size 512 cache GPU: 27.381216 ms

Size 512 shared GPU: 0.754976 ms

Size 1024 naive CPU: 792.968079 ms

Size 1024 naive GPU: 241.626434 ms

Size 1024 cache GPU: 258.099579 ms

Size 1024 shared GPU: 11.601920 ms

Size 2048 naive CPU: 6262.144531 ms

Size 2048 naive GPU: 1815.276855 ms

Size 2048 cache GPU: 1827.928955 ms

Size 2048 shared GPU: 48.436161 ms

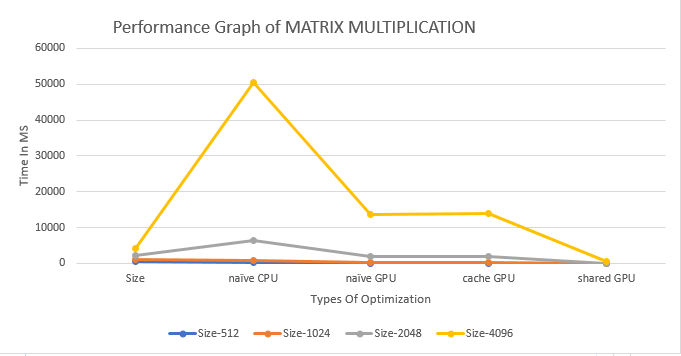
Size 4096 naive CPU: 50552.894531 ms

Size 4096 naive GPU: 13679.469727 ms

Size 4096 cache GPU: 13773.951172 ms

Size 4096 shared GPU: 529.743286 ms

**Below is the timing and performance of different size of matrix and their respective optimization timings:**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Size** | **naïve CPU** | **naïve GPU** | **cache GPU** | **shared GPU** |
| **512** | 99.91552 | 35.40435 | 27.381216 | 0.754976 |
| **1024** | 792.968079 | 241.626434 | 258.099579 | 11.60192 |
| **2048** | 6262.144531 | 1815.276855 | 1827.928955 | 48.436161 |
| **4096** | 50552.89453 | 13679.46973 | 13773.95117 | 529.743286 |
|  |  |  |  |  |

**From the above description we can conclude that except from Shared/Optimal code on GPU others are almost same performance as the NAÏVE GPU . Also the naïve CPU is very low compared to GPU .**

**Matrix Transpose :** This is a method to transpose each block of the naive transpose handles a 64x64 block of the input matrix, with each thread of the block handling a 1x4 section and each warp handling

a 32x4 section. If we split the 64x64 matrix into 32 blocks of shape (32, 4), then we have

a block matrix of shape (2 blocks, 16 blocks). Warp 0 handles block (0, 0), warp 1 handles (1, 0), warp 2 handles (0, 1), warp n handles (n % 2, n / 2).

Below is the result after optimizing and running the program in **opuntia.cacds.uh.edu (mcebolsu)** cluster .

Size 512 naive CPU: 0.002112 ms

Size 512 GPU memcpy: 0.020928 ms

Size 512 naive GPU: 0.041888 ms

Size 512 shmem GPU: 0.019808 ms

Size 512 optimal GPU: 0.020032 ms

Size 1024 naive CPU: 4.344704 ms

Size 1024 GPU memcpy: 0.054112 ms

Size 1024 naive GPU: 0.144480 ms

Size 1024 shmem GPU: 0.054848 ms

Size 1024 optimal GPU: 0.057152 ms

Size 2048 naive CPU: 13.179680 ms

Size 2048 GPU memcpy: 0.217952 ms

Size 2048 naive GPU: 0.520736 ms

Size 2048 shmem GPU: 0.194144 ms

Size 2048 optimal GPU: 0.207840 ms

Size 4096 naive CPU: 196.310791 ms

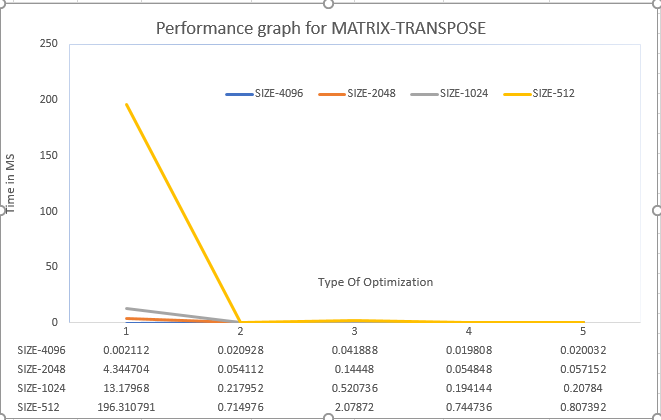
Size 4096 GPU memcpy: 0.714976 ms

Size 4096 naive GPU: 2.078720 ms

Size 4096 shmem GPU: 0.744736 ms

Size 4096 optimal GPU: 0.807392 ms

**Below is the timing and performance of different size of matrix and their respective optimization timings:**



**Type of Transpose Optimization:**

1. **Naïve CPU Transpose :**

**Code:**

const int i = threadIdx.x + 64 \* blockIdx.x;

int j = 4 \* threadIdx.y + 64 \* blockIdx.y;

const int end\_j = j + 4;

for (; j < end\_j; j++)

output[j + n \* i] = input[i + n \* j];

1. **GPU Memory copy**
2. **Naïve GPU**
3. **Share Memory GPU**

**Code:**

\_\_global\_\_

void shmemTransposeKernel(const float \*input, float \*output, int n) {

// TODO: Modify transpose kernel to use shared memory. All global memory

// reads and writes should be coalesced. Minimize the number of shared

// memory bank conflicts (0 bank conflicts should be possible using

// padding). Again, comment on all sub-optimal accesses.

\_\_shared\_\_ float block[64][65];

int i = blockIdx.x \* 64 + threadIdx.x;

int j = blockIdx.y \* 64 + threadIdx.y;

int width = gridDim.x \* 64;

for (int k = 0; k < 64; k+=16)

{

block[threadIdx.y+k][threadIdx.x] = input[(j+k)\*width + i];

}

\_\_syncthreads();

i = blockIdx.y \* 64 + threadIdx.x; // transpose block offset

j = blockIdx.x \* 64 + threadIdx.y;

for (int k = 0; k < 64; k+=16)

{

output[(j+k) \* width + i ] = block[threadIdx.x][threadIdx.y+k];

}

}

1. **Optimal GPU**

**Code:**

\_\_global\_\_ void optimalTransposeKernel(const float \*input, float \*output, int n)

{

\_\_shared\_\_ float tile[64][65];

int x = blockIdx.x \* 64 + threadIdx.x;

int y = blockIdx.y \* 64 + threadIdx.y;

const int width = gridDim.x \* 64;

const int height = gridDim.y \* 64;

if (x < width && y < height)

{ tile[threadIdx.y][threadIdx.x] = input[y\*width + x];

tile[threadIdx.y+16][threadIdx.x] = input[(y+16)\*width +x];

tile[threadIdx.y+32][threadIdx.x] = input[(y+32)\*width +x];

tile[threadIdx.y+48][threadIdx.x] = input[(y+48)\*width +x];

}

\_\_syncthreads();

x = blockIdx.y \* 64 + threadIdx.x; // transpose block offset

y = blockIdx.x \* 64 + threadIdx.y;

if (y < width && x < height)

{ output[y\*height + x] = tile[threadIdx.x][threadIdx.y];

output[(y+16)\*height +x] = tile[threadIdx.x][threadIdx.y+16];

output[(y+32)\*height +x] = tile[threadIdx.x][threadIdx.y+32];

output[(y+48)\*height +x] = tile[threadIdx.x][threadIdx.y+48];

}

}

In case of TRNASPOSE also we can clearly that the optimal solution is derived from LOOP UNROLLING and the shared memory has been derived using padding and has a tremendous improvement in performance compared to the NIAVE versions and CPU versions.

**Conclusion:**

• It is easy enough to get something to run on a GPU

• But it is difficult to get it to run fast – Things to consider

• Which algorithm to use; some algorithms are better suited for GPUs than others

• Understand if the kernel is compute-bound or memory I/O bound and optimize it accordingly