CUDA PROGRAMMING

The Complexity of the Problem is the Simplicity of the Solution

SHARED MEMORY AND SYNCHRONIZATION IN CUDA PROGRAMMING

POSTED BY NITIN GUPTA AT 07:58 | 3 COMMENTS

This article lets u know what is shared memory and synchronization with detail and complete working example.

Let's start our discussion. We start with this question;
What is Shared Memory and Synchronization in CUDA Programming?

Motivation

The motivation for splitting blocks into threads was simply one of working around hardware limitations to the number of blocks we can have in flight. This is fairly weak motivation, because this could easily be done behind the scenes by the CUDA runtime. Fortunately, there are other reasons one might want to split a block into threads.

Shared Memory in CUDA

CUDA C makes available a region of memory that we call *shared memory*. This region of memory brings along with it another extension to the C language akin **to __device__ and __global__.** As a programmer, we can modify our variable declarations with the CUDA C keyword **__shared__** to make this variable resident in shared memory. **But what's the point?**

TAGS

CUDA BASICS

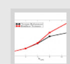
CUDA PROGRAMMING CONCEPT

RELATED POSTS

- How to Query Device Properties and Handle Errors in CUDA C/C++
- How to Implement Performance Metrics in CUDA C/C++ | Optimization in CUDA
- How to specify architecture while compiling CUDA program in Visual profiler How to change command line flag in CUDA
- Further Optimization in histogram CUDA code | Fast implementation of histogram in CUDA
- CUDA complete | Complete reference on CUDA

SHARE THIS

RECENT POPULAR RANDOM


TEXTURE OBJECT IN CUDA | Bindless Texture in CUDA
01/04/2013 - 0 comments


How to specify architecture while compiling CUDA program in Visual profiler How to change command line flag in CUDA
24/03/2013 - 0 comments

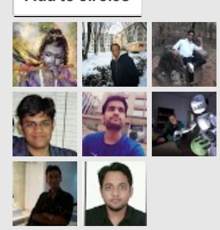
Using Shared Memory in CUDA C/C++
11/03/2013 - 0 comments

Optimization in histogram CUDA code: When number of Bins not equal to max threads in a block
08/03/2013 - 9 comments

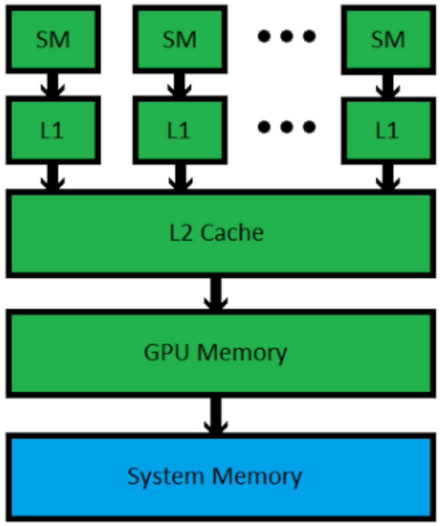
GOOGLE+ FOLLOWERS

Nitin Gupta

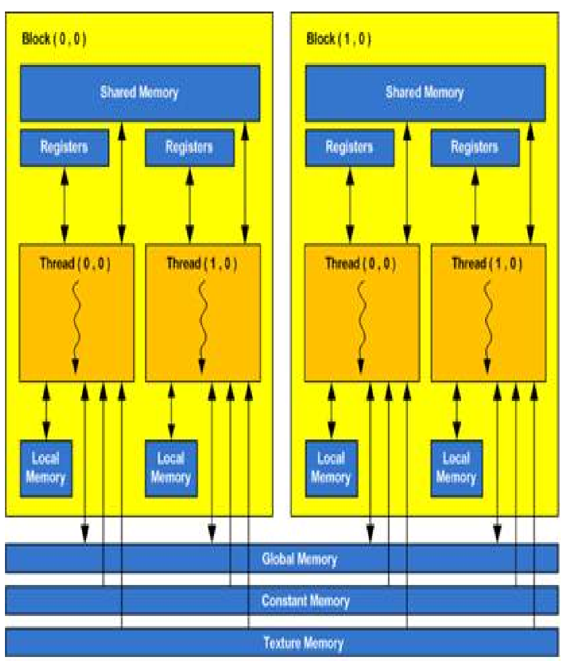
Add to circles



8 have me in circles View all



We're glad you asked. The CUDA C compiler treats variables in shared memory differently than typical variables**. It creates a copy of the variable for each block that you launch on the GPU. Every thread in that block shares the memory, but threads cannot see or modify the copy of this variable that is seen within other blocks. This provides an excellent means by which threads within a block can communicate and collaborate on computations. Furthermore, shared memory buffers reside physically on the GPU as opposed to residing in off-chip DRAM**. Because of this, the latency to access shared memory tends to be far lower than typical buffers, making shared memory effective as a per-block, software managed cache or scratchpad. Fig shows you the memory hierarchy diagram in CUDA Arch. With Shared Memory



Motivation of Synchronization

Race Condition

The prospect of communication between threads should excite you.

It excites me, too. But nothing in life is free, and interthread communication is no exception. **If we expect to communicate between threads, we also need a mechanism for synchronizing between threads**. For example, if thread A writes a value to shared memory and we want thread B to do something with this value, we can't have thread B start its work until we know the write from thread A is complete**. Without synchronization, we have created a race condition where the correctness of the execution results depends on the nondeterministic details of the hardware**.

Shared Memory and Global Memory

Shared memory

Threads within the same block have two main ways to communicate data with each other. The fastest way would be to use shared memory. When a block of threads starts executing, it runs on an SM, a multiprocessor unit inside the GPU. **Each SM has a fairly small amount of shared memory associated with it, usually 16KB of memory**. To make matters more difficult, often times, multiple thread blocks can run simultaneously on the same SM. **For example,** if each SM has 16KB of shared memory and there are 4 thread blocks running simultaneously on an SM, then the maximum amount of shared memory available to each thread block would be 16KB/4, or 4KB. So as you can see, if you only need the threads to share a small amount of data at any given time, using shared memory is by far the fastest and most convenient way to do it.

Global memory

However, if your program is using too much shared memory to store data, or your threads simply need to share too much data at once, then it is possible that the shared memory is not big enough to accommodate all the data that needs to be shared among the threads. In such a situation, threads always have the option of writing to and reading from global memory. Global memory is much slower than accessing shared memory; however, **global memory is much larger**. For most video cards sold today, there is at least 128MB of memory the GPU can access.

Looking for the example?

Declaring shared arrays

For CUDA kernels, there is a special keyword**, __shared__,** which places a variable into shared memory for each respective thread block. The **__shared__** keyword works on any type of variable or array. In the case for this tutorial, we will be declaring three arrays in shared memory.

```
// Declare arrays to be in shared memory.
// 256 elements * (4 bytes / element) * 3 = 3KB.
__shared__ float min[256];

__shared__ float max[256];
__shared__ float avg[256];
```

If you are not clear with idea of thread and block architecture and how to decide. please go through this link

For descriptive example; Vector Dot Product
For Simple example with more description; Simple and explained

Summary of the Article

In summing up this article, it is possible, and many times necessary, for threads within the same block to communicate with each other through either shared memory, or global memory. Shared memory is by far the fastest way, however due to it's size limitations, some problems will be forced to use global memory for thread communication. Using **__syncthreads** is sometimes necessary to ensure that all data from all threads is valid before threads read from shared memory which is written to by other threads. Below is a graph of execution time it took my CPU against the amount of time it took my graphics card. the CPU is a 2.66 Core

2 Duo, while the graphics card is a GTX 280, slightly **underclocked.** As you can see, the GPU is faster when there are at least a million elements, and the spread between the GPU and CPU continues to widen with more elements. However, main system memory may be a significant bottleneck which is preventing the GPU from achieving more than 1.5x the processor performance.

Got Questions?
Feel free to ask me any question because I'd be happy to walk you through step by step!

References and External Links
Wikipedia
hyperphysic
algebralab
CUDA C Programming Guide
CUDA; Nvidia

For Contact us….. Click on Contact us Tab

3 comments:

Anonymous 12 March 2013 09:33

guys the program that i'm working on is for matrix multiplication using non shared and shared memory, and it shows the same calculation time, if possible could you have a look at the code below thanks.

```
#include
#include
#include

# define TILE_WIDTH 32


__global__ void gpu_matrixmult (int *Md, int *Nd, int *Pd, int Width)
{
__shared__ int Mds[TILE_WIDTH][TILE_WIDTH];
__shared__ int Nds[TILE_WIDTH][TILE_WIDTH];

int bx = blockIdx.x; // tile (block) indices
int by = blockIdx.y;
int tx = threadIdx.x; //thread indices
int ty = threadIdx.y;
int Row = by * TILE_WIDTH + ty; // global indices
int Col = bx * TILE_WIDTH + tx;


int Pvalue = 0;
for (int m = 0; m < Width/TILE_WIDTH; m++)
{
Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)]; // load Md, Nd tiles into sh. mem
Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
__syncthreads();

for ( int k = 0; k < TILE_WIDTH; k++)
Pvalue += Mds[ty][k] * Nds[k][tx];
}
Pd[Row * Width + Col] = Pvalue;
}


int main ()
{
int *A, *B, *C;
int N=10;
```

```
int i,j; //loop counters
int size;
char key;
int* Ad;
int* Bd;
int* Cd;
cudaEvent_t start, stop; // using cuda events to measure time
float elapsed_time_ms; // which is applicable for asynchronous code also

//keyboard input

do
{
printf("Enter size of array in one dimension (square array), currently %d\n",N);
scanf("%d",&N);

dim3 Block(TILE_WIDTH, TILE_WIDTH);
dim3 Grid(N / TILE_WIDTH, N / TILE_WIDTH);

size = N* N * sizeof(int);

A = (int*) malloc(size);
B = (int*) malloc(size);
C = (int*) malloc(size);
for(i=0;i < N;i++) { // load arrays with some numbers
for(j=0;j < N;j++) {
A[i * N + j] = i;
B[i * N + j] = i;
}
}

cudaMalloc((void**)&Ad, size);
cudaMalloc((void**)&Bd, size);
cudaMalloc((void**)&Cd, size);

cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
cudaMemcpy(Cd, C, size, cudaMemcpyDeviceToHost);

cudaEventCreate(&start); // instrument code to measure start time
cudaEventCreate(&stop);
cudaEventRecord(start, 0); // here start time, after memcpy


// Launch the device computation
gpu_matrixmult<<>>(Ad, Bd, Cd, N);

// Read C from the device
cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

cudaEventRecord(stop, 0); // measuse end time
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsed_time_ms, start, stop );
printf("Time to calculate results on GPU: %f ms.\n", elapsed_time_ms);

printf("\nEnter c to repeat, return to terminate\n");
scanf("%c",&key);
scanf("%c",&key);

} while (key == 'c');



// Free device memory
cudaFree(Ad);
cudaFree(Bd);
cudaFree(Cd);
cudaEventDestroy(start);
cudaEventDestroy(stop);


system ("pause");
}
```

Reply

Replies

Nitin Gupta 12 March 2013 10:16

Have you walk through this implementation "http://cuda-programming.blogspot.in/2013/01/cuda-c-program-for-matrix-addition-and.html"

Nitin Gupta 12 March 2013 10:18

And try to reduce you Tile width to some smaller number like ;
16,8,4 or 2

Reply

Enter your comment...

Comment as: Google Account

Publish Preview

Help us to improve our quality and become contributor to our blog

Links to this post

Create a Link

Newer Post Home Older Post

Become a contributor to this blog. Click on contact us tab

LABELS

- Books on CUDA (9)
- C program (2)
- Compilation (3)
- CUDA Advance (25)
- CUDA Basics (31)
- CUDA Function (1)
- CUDA Programming Concept (41)
- CUDA programs Level 1.1 (10)
- CUDA programs Level 1.2 (4)
- CUDA programs Level 2.1 (3)
- Debugging (2)
- Images Processing (6)
- Installation (2)
- Kepler Features (1)
- Matlab Coding (3)
- Optimization in CUDA (17)

LIKE US

CLOUD

ADMIN

Nitin Gupta

facebook

Name:
Nitin Gupta
Email:
nitinguptaiit@gmail.com
Status:
Very impressive, Team India!! The Champion of...

08:30:06 am