GPU Computing

Felix Wellschmied

UC3M

How CPUs and GPUs work

How CPUs and GPUs work

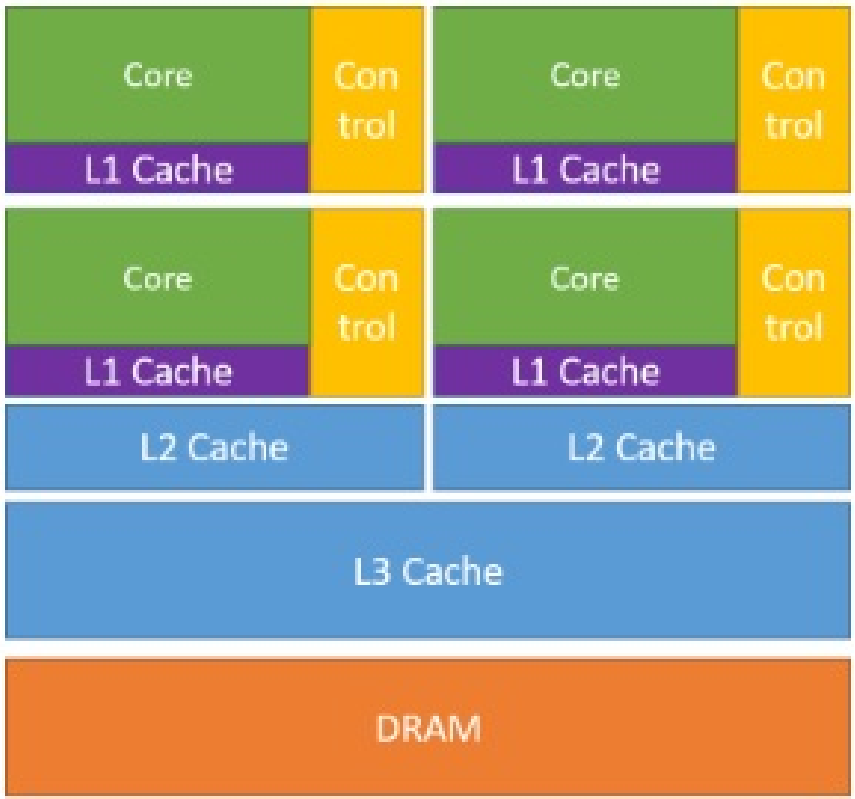
I begin by providing an idea how GPUs work relative to CPUs to

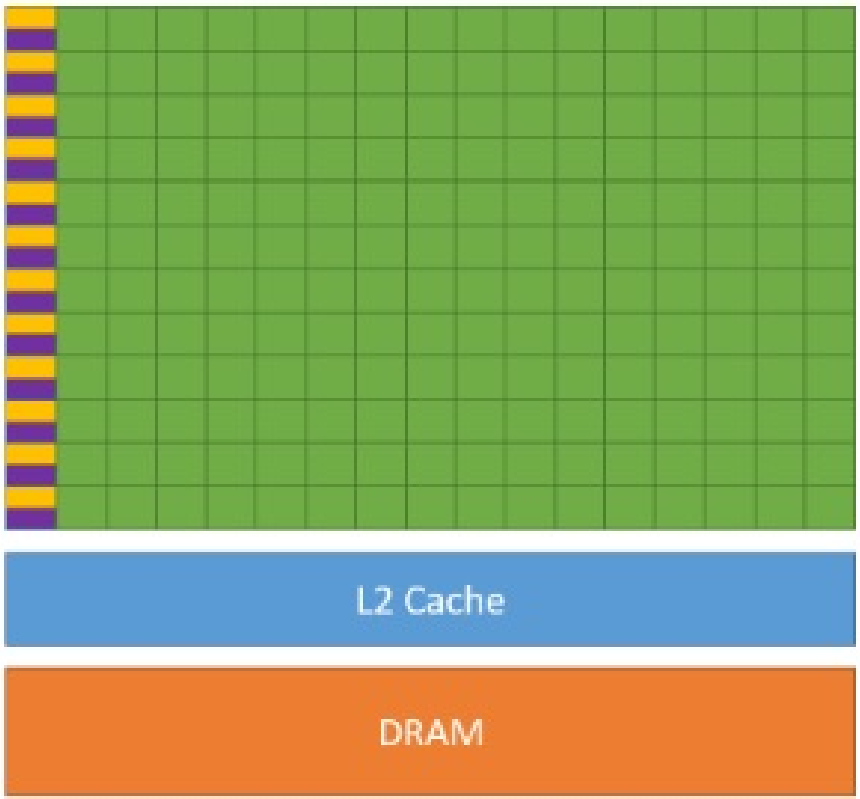
- facilitate the decision which GPU to buy.
- understand which types of problems can be better solved by GPUs.
- understand how to program these problems.

The structures of CPUs and GPUs

- CPUs and GPUs share similar features: they have cores that access data from cache and from a global memory.
- CPUs are designed to run a series of tasks quickly and many transistors are used for cache management as cache bandwidth is much more favorable than reading data from the global memory.
- GPUs are designed to run many (limited) tasks in parallel and most transistors are used for arithmetic operations leading to a large number of computing cores.
- The market provides different GPUs. NVIDIA has the big advantage that it provides an easily implementable programming language, CUDA, which I will be using.

CPUs and GPUs





Source: NVIDIA programming guide

An example to get excited

My desktop computer has an Intel i9-13000 processor and a NVIDIA GeForce RTX 4090. In 2023, this was the upper end for a commercial (gaming) computer.

- The CPU has 24 cores providing 845 GFLOPS (845 billion operations per second) of computing power with single precision.
- The GPU has 16384 cores providing 82.6 TFLOPS of computing power!
- One may think that this ends any discussion, however, unfortunately, FLOPS do not really matter any longer.

The issue of memory bandwidth (CPU)

Loading data from the central memory is slow. My CPU has a maximum memory bandwidth of 89.6 GB/second of data. A single-precision piece of data is 4 bytes leading to 22.4 GB of data observations. To keep the CPU busy, you need to do

$$\text{Required compute intensity} = \frac{FLOPs}{\text{Data rate}} = 37.7 \tag{1}$$

operations to each data point.

The issue of memory bandwidth (GPU)

Things are worse with the GPU. My GPU has a maximum memory bandwidth of 1008(!) GB/second of data leading to 252 GB data observations. To keep the GPU busy, you need to do

$$\text{Required compute intensity} = \frac{FLOPs}{\text{Data rate}} = 329 \tag{2}$$

operations to each data point.

The issue of memory latency

- The memory bandwidth is the maximum amount of data that can be loaded from the central memory.
- Most steps of an algorithm require much less than the maximum bandwidth.
- As memory needs to physically travel on the chip, there is a minimum of time even a small memory transfers takes.
- The time the processor is waiting for the data to arrive before it can start working is called memory latency.

An example of memory latency

Assume that the memory latency is 300ns and part of your algorithm is to add a single-precision vector to another vector:

$$\begin{aligned}&\text{for (int i} = 1; i < N; \text{ i++)}\{\\ &\quad x[i] = x[i] + y[i];\\ &\}\end{aligned}$$

To perform each arithmetic, the processor reads 8 bytes in 300ns.

My memory bandwidth allows me to transfer 302400 bytes in 300ns.

I am using only 0.003% of my memory bandwidth at each stage of the loop.

Two ways to address the issue

- CPU: have low latency and be very efficient in memory processing.
- GPU: run a lot of arithmetic operations in parallel. In the before example, running 38500 computations in parallel allows us to use the entire memory bandwidth.

What the CPU does

- When x and y are sufficiently short, their size is small enough to copy them at the beginning of the loop to the processor cache (I have 36 MB) from where latency is much shorter.

- Moreover, your compiler is smart and performs so called loop unrolling for you. Loop unrolling loads several elements of the loop at once (if they can be handled independently) by altering their register entries, then performs for those elements the calculations, and finally stores the results. However, the CPU can only keep track of a few concurrent calculations (it has a small register).

$$\begin{aligned}&\text{for (int i} = 1; i < 5;\ \text{i++)}\{\\ &\quad x[i] = x[i] + y[i];\\ &\}\end{aligned}$$

is very efficient on the CPU.

What the GPU does

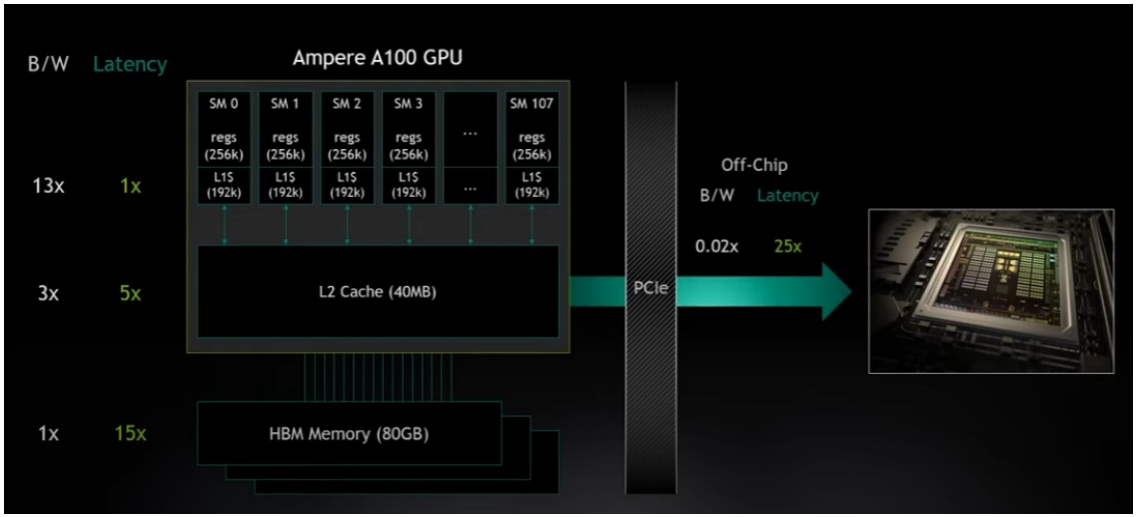
- The GPU schedules a large number of parallel threads that each do the computation **for one element** of the vector independently.
- A thread is scheduled and requests data, a second thread is scheduled and requests data, ...
- My GPU has 128 streaming multiprocessor units (SMU) each with 256kb of register, giving me 32.8 MB of outstanding load data.

$$
\begin{aligned}
&\text{for (int i} = 1; i < 10^7; \text{ i++)\{} \\
&\quad x[i] = x[i] + y[i]; \\
&\quad \text{for (int i2} = 1; i2 < 30; \text{ i2++)\{} \\
&\quad\quad x[i] = x[i] + 5.0; \\
&\quad \} \\
&\}
\end{aligned} \tag{3}
$$

Threads and SMUs

- By design, a SMU always runs threads in blocks of 32 (in case of NVIDIA) called a warp. The maximum number of threads per SMU is for most GPUs 1024, i.e., 32 warps.
- Threads within an SMU have access to some shared cache memory (L1) which the user can manage.
- There is further L2 cache available across SMUs as well as the global memory.

Where to get the data from

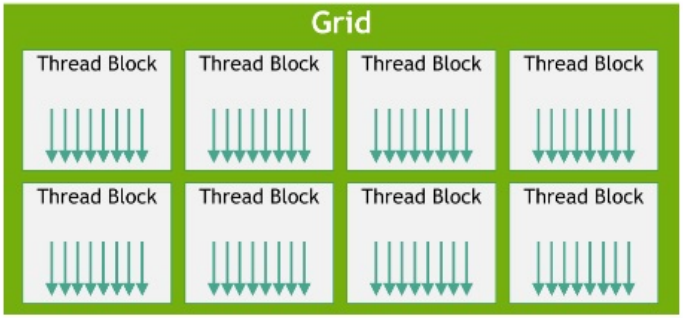


Source: How GPU computing works

- Obviously, using data in the cache is most efficient. As the fastest cache is only shared within SMUs, it will matters how you set up your threads.

- Transferring data from the CPU memory to the GPU memory is extremely slow! You want to run your entire code (segment, e.g., value function iteration) on the GPU. My GPU has 24GB of memory.

Threads in CUDA



Source: NVIDIA programming guide

- In CUDA, you control how many SMUs are working and how many threads they run.
- Usually, the number of threads is defined in multiples of 32.
- A block contains the threads and resides on a SMU, e.g., if you define fewer blocks than you have SMUs, you waste resources.
- Together, threads and blocks make up what is called a grid. Your total number of threads is $N_{threads} * N_{blocks}$.

Getting started with GPU computing

GPU programming in high-level languages

- I will discuss how to use C++ to implement GPU computing.
- However, higher-level languages have also introduced GPU computing without the need to write C++ code.
- I will briefly discuss the use of Matlab and Julia.

GPU computing implemented in Matlab

- Matlab allows you to do GPU computing without writing CUDA code.
- You have to write the code such that a particular operation is done to each element in an array. Matlab than solves the code for each array element in parallel.
- For me, there is not enough flexibility. As explained, you only gain speed if an entire code section runs on the GPU, and I find it impossible to write complex code in the way required by Matlab, e.g., you have to use a lot of meshgrids.
- Also, this is somewhat a black box. E.g., I don't know how Matlab decides on the thread and block structure.
- The VFI toolkit provides a standardized toolkit for some applications of value function iteration.

GPU computing implemented in Julia

- Julia appears to provide an easily implementable way to use the GPU and provides significantly more flexibility than Matlab without writing CUDA/C++ code.
- As I understand it, you can specify the thread and block structure as with C++.
- As I understand it, you can even use shared memory.
- Here you can find an example for value function iteration.
- Hence, if you already use Julia, you may prefer to forget about CUDA. However, to me, the syntax does not appear significantly simpler than the CUDA/C++ syntax. Moreover, you are restricted to float instead of double precision.

Using Matlab as a wrapper

- Personally, I like using Matlab as a wrapper. To me, having an interpreted language helps me in debugging and looking at my results. Hence, I will first explain how to imbed CUDA code into Matlab and afterward how to write a self-contained CUDA/C++ program.

- First, type *gpuDevice* into your Matlab command window. This should list your available GPU(s).

Requisites

- Make sure you have the latest driver installed for your GPU.
- All my descriptions use Windows as operating system, and you need to have Visual studio installed. When installing, make sure to install C++ for the desktop environment.
- Next, install the latest CUDA environment that is suitable for your GPU's microarchitecture. For example, my GPU uses the *Ada Lovelance* architecture.
- CUDA is also available for *Fortran* instead of C++ but only on Linux.

A brief note on indexing

- You may be accustomed to use multi-dimensional arrays in your codes.
- If you think about the computer memory, this is just a very long line of blocks, each with an index.
- Basic C++ only knows one-dimensional arrays (there exist libraries for multi-dimensional arrays) and linear indexing.
- Matlab uses 1-indexing, meaning $A(1)$ given you the first entry in an array.
- C++ uses 0-indexing meaning $A[0]$ given you the first entry in an array.

A brief note on variable definitions

- Matlab allows you to be "quick and dirty" with variable definitions, e.g., a variable can starts as double precision and become an integer:

  ```
  V = 5;
  V = V * int32(1);
  ```

- In C++ you have to be more specific and define your variables.
  - int V = 5;
  - float V = 5.0;
  - double V = 5.0;

A brief note on pointers

- When you pass a variable in Matlab to another function, you "pass it by value", e.g.,
  $V = 5;\ f = myFun(V)$
  tells your function the value of V. Within the function, Matlab will make a copy of V on a free memory block. Hence, you can change the value of V:
  $V = V + 1;$
  and, as long a you do not explicitly pass it back, once the function has run, V has still its original value 5.
- When we will pass variables to our CUDA code, we will usually "pass them by pointer" meaning we just pass an index on the memory where the variable begins. When our CUDA code changes the value on that memory block, the value will change permanently.

Setting up your first function

- In Visual studio, open a new text file (no project needed) and call it *matmul.cu*.
- Usually, you want to include some math library. You do so by having on top a: $\#include < math.h >$.
- Next, you want to write your main code on the GPU (the device) which is called a kernel:

```
__global__ void myFun(Some inputs){
    Some code
}
```

Threads and blocks

- Let us say we want to compute $C = A * B$.
- The idea is to have for each element in C, c_{ij}, a thread that multiplies the *ith* row of A with the *jth* column of B.
- For that, you need to define the thread that does that operation. To identify threads, CUDA uses 3 dimensional vectors taking the form

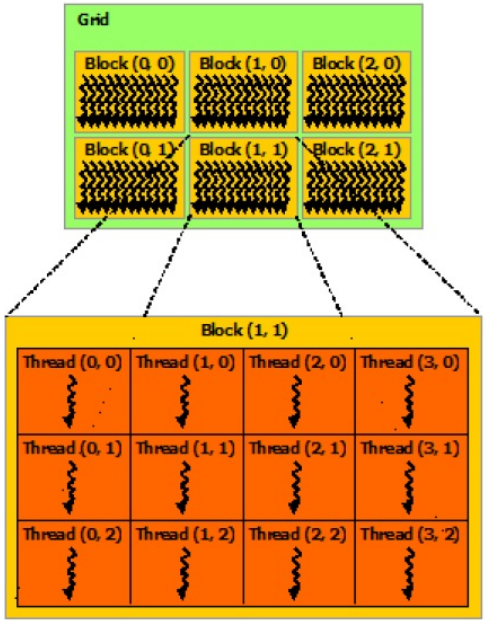
```
dim3 numBlocks(xdim, ydim, zdim);
dim3 threadsPerBlock(xdim2, ydim2, zdim2);
```

- You can think of your threads and blocks as 3-dimensional objects. For example,

```
dim3 numBlocks(3, 3, 3);
dim3 threadsPerBlock(2, 2, 2);
```

generates 8 threads per block and 27 blocks.

Threads and blocks II



Source: NVIDIA programming guide

- The picture shows a 2-dim example with blockdim $[3, 2]$ and threaddim $[4, 3]$ (remember, 0-indexing).
- Each block, has always the same thread structure.

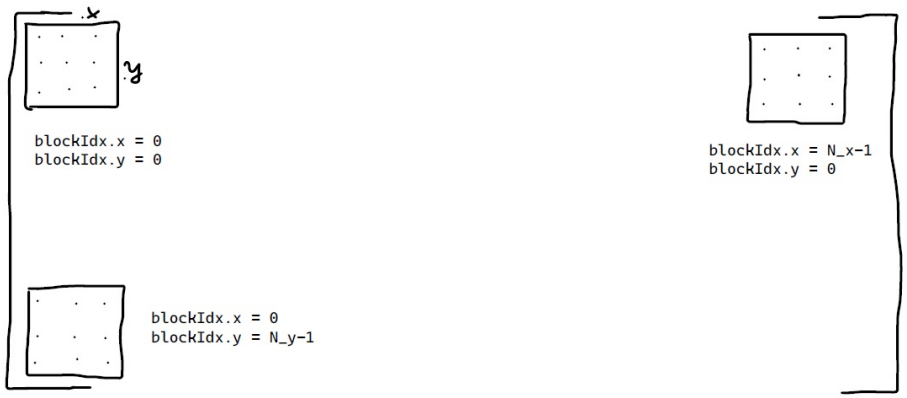
Threads and blocks III

This way of organizing things facilitates working with up to 3-dimensional arrays (if you have more dimensions, you need to stack them). To identify the threads, CUDA uses the build-in *blockIdx*, *blockDim* and *threadIdx* variables and translates them to the linear memory grid with 0-indexing:

$$\text{int COL} = \mathit{blockIdx.x} * \mathit{blockDim.x} + \mathit{threadIdx.x};$$
$$\text{int ROW} = \mathit{blockIdx.y} * \mathit{blockDim.y} + \mathit{threadIdx.y};$$
$$\text{int PAGE} = \mathit{blockIdx.z} * \mathit{blockDim.z} + \mathit{threadIdx.z};$$

Careful, in Matlab, the first dimension is the ROW dimension, and Matlab vectorizes your arrays going down column by column.

Threads and blocks IV



- Here, I have 9 threads in each block (3x3).
- I have for the x dimension of the block N_x and for the y dimension N_y.
- This is only for better understanding. Remember, everything will be on a linear line.

Three code examples

I will now discuss three code examples:

1. Matrix multiplication which has some nice features.
2. Value function iteration with on-grid-search.
3. Value function iteration with Golden section search.

I will always display only some code highlights. You can find the complete codes on my homepage.

Matrix multiplication CUDA code

```
#include <math.h>

//matrix multiplication
__global__ void matmul(double* C, const double* A, const double* B, int* index)
{
    //get indexes
    int COL = blockIdx.x * blockDim.x + threadIdx.x;
    int ROW = blockIdx.y * blockDim.y + threadIdx.y;


    //read out size of matrices
    int col_a = index[0];
    int row_a = index[1];
    int col_b = index[2];
    int row_b = index[3];
```

Reading in data:

- *const* tells the program that the value cannot be altered.
- $*$ means the variable is passed by pointer and not by value.
- As we work with matrices, I use a 2-dimensional kernel.

Matrix multiplication CUDA code II

```
    //make sure not to leave the grid
    if (ROW < row_a && COL < col_b) {

        double val = 0.0; //initialize
        for (int i1 = 0; i1 < col_a; i1++) { //loop over each entry in A/B
            val += A[i1 * row_a + ROW] * B[i1 + COL * row_b];
        }
        C[COL * row_a + ROW] = val;

    }

}
```

- We could launch more threads than points in the state space. The first "if" condition makes sure that those threads remain idle.
- $COL * row\_a + ROW$ is the entry in the C vector. Again, pay attention to the Matlab indexing. E.g., entry 2 in the vector (index 1) has to be column 1, row 2.

The Matlab code

```
clear all
close all
clc

%% Matrix multiplication


A = [1 2 3; 4 5 6; 7 8 9];
B = [1 2 3 4; 5 6 7 8; 9 10 11 12];
C = zeros(3,4);
C_CPU = A*B;
A = A(:)';
B = B(:)';
C = C(:)';

%matrix size
col_a = int32(3);
row_a = int32(3);
col_b = int32(4);
row_b = int32(3);
```

- Note, I convert the matrices into vectors.
- For the indexes denoting matrix sizes, I have to explicitly tell Matlab that these are integers, Otherwise, it initializes these as doubles.

The Matlab code II

```
%copy variables to the GPU memory
index_d = gpuArray([col_a row_a col_b row_b]);
A = gpuArray(A);
B = gpuArray(B);
C = gpuArray(C);

k = parallel.gpu.CUDAKernel('matmul.ptx','matmul.cu');
k.ThreadBlockSize = [4 3 1]; %number of threads
k.GridSize = [1 1 1]; %number of blocks
[C] = feval(k,C,A,B,index_d);
%copy result back to CPU memory
C_GPU = gather(C);
C_GPU = reshape(C_GPU,[row_a,col_b]);
```

- The command *gpuArray* copies the data from the CPU memory to the GPU main memory.
- We have to specify three things for our kernel:
  - The name of the CUDA file: *matmul.cu*.
  - The number of threads per block in each dimension: [4 3 1].
  - The number of blocks in each dimension: [1 1 1].
- Matlab wants to know which variable(s) to read back: [C].

Compiling your code

- We still need to use Visual studio to compile our CUDA code.
- For this, go to *Tools* $\rightarrow$ *Command line* $\rightarrow$ *Developer PowerShell*.
- Using *cd myDirectory*, direct to the folder where you saved your *.cu* file.
- CUDA provides you with a compiler, nvcc. You have to locate it in your Visual Studio. For VS22, I have it at
  C:\ [...] \VC\Tools\MSVC\14.35.32215\bin\Hostx64\x64

Compiling your code II

```
PS D:\GPU\matmul> nvcc -ptx -ccbin "C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.35.32215\bi
n\Hostx64\x64" -o matmul.ptx matmul.cu
```

- Matlab requires the parallel thread execution file *.ptx*, so you have to tell your compiler to create it.
- Finally, you copy your *.cu* and *.ptx* file into the folder with your Matlab code.

The C++ code

- You can also write a self-contained C++ code.
- In VS, open a new project and select a CUDA project. This will generate a *kernel.cu* file which will be your main file.
- In C++, you need to define all sub-programs first.
- Then follows the main code which you can start with int main().

The C++ code II

```
// Host code
int main()
{


    //matrices
    double A[9] = { 1.0, 4.0, 7.0, 2.0, 5.0, 8.0, 3.0, 6.0, 9.0 }; //A matrix
    double B[12] = { 1.0, 5.0, 9.0, 2.0, 6.0, 10.0, 3.0, 7.0, 11.0, 4.0, 8.0, 12.0 }; //B matrix
    double C[12];
    int index[4] = {3, 3, 4, 3};

    // Allocate vectors in device memory
    size_t size_A = 9 * sizeof(double);
    size_t size_B = 12 * sizeof(double);
    size_t size_index = 4 * sizeof(int);
    double* A_d;
    cudaMalloc(&A_d, size_A);
    double* B_d;
    cudaMalloc(&B_d, size_B);
    double* C_d;
    cudaMalloc(&C_d, size_B);
    int* index_d;
    cudaMalloc(&index_d, size_index);

    // Copy vectors from host memory to device memory
    cudaMemcpy(A_d, A, size_A, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size_B, cudaMemcpyHostToDevice);
    cudaMemcpy(C_d, C, size_B, cudaMemcpyHostToDevice);
    cudaMemcpy(index_d, index, size_index, cudaMemcpyHostToDevice);
```

- We first need to allocate memory on the GPU. For that we need to determine the size of the matrices.
- Next, we use *cudaMemcpy* to copy the matrix from the CPU to the GPU memory.

The C++ code III

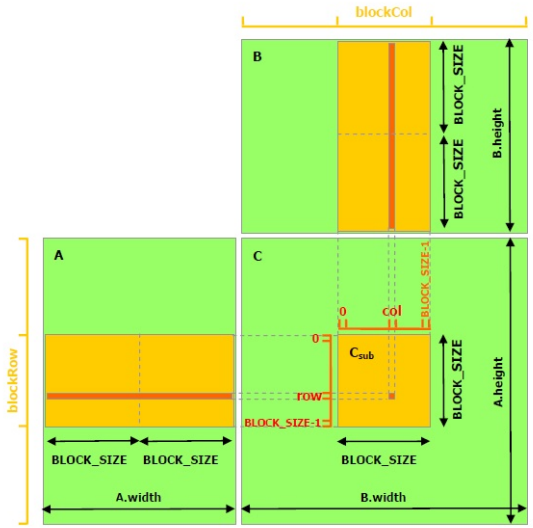
```
// Set up kernels
dim3 threadsPerBlock(4, 3);
dim3 numBlocks(1, 1);

// run the kernel
matmul<<<numBlocks, threadsPerBlock>>>(C_d,A_d,B_d,index_d);

// copy result to CPU memory
double* result = (double*)malloc(size_B);
cudaMemcpy(result, C_d, size_B, cudaMemcpyDeviceToHost);


// Free device memory
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);

// print result to screen
for (int i1 = 0; i1 < 12; i1++) {
    printf("%f\n", result[i1]);
}
```

- Next we define the size of threads and blocks. Here, I use again a two-dimensional grid.
- Running the code uses <<< ... >>> to define the kernel size.
- Finally, we copy the results back to the CPU memory and print them to the screen.

Programing with shared memory

- Note, different threads read the same elements of A and B to calculate C. E.g., to calculate $C(2,2)$ and $C(2,3)$, both threads read the second row of A.
- Loading that row from the global memory once to the L1 cache would be much faster than reading it each time from the global memory.
- In our case, all threads reside on the same SMU, and A and B are small, hence, the compiler might well have copied those to the cache automatically.
- However, we can also explicitly ensure this. Moreover, with larger matrices, we will need more than one block, and it will matter which parts of C which block computes.

Programing with shared memory II



- Assume I set up a $16x16$ thread block to compute a block of C, e.g., C_{sub}.
- As A has more rows than 16, I will have to partition the relevant parts of A which I will call tiles.
- The idea will be to loop over these tiles, and within the loop each thread reads one piece of data from A and B to the shared memory. Finally, each thread does the vector multiplication for that tile.

Simple matrix multiplication III

Understanding the benefits of shared memory:

- Each thread in a tile will read two data items from the global memory to the shared memory. This happens basically in parallel.
- It will then multiply two vectors of size 16. This is key, we have increased the number of arithmetic operations relative to the number of loads from the central memory.
- Finally, it will store the result in the global memory.

Matlab code

```
%matrix size
col_a2 = int32(512);
row_a2 = int32(512);
col_b2 = int32(768);
row_b2 = int32(512);
length_threadx = int32(16);
length_thready = int32(16);
tile_N = row_a2/length_thready; %number of tiles (separations)

%copy variables to the GPU memory
index_d = gpuArray([col_a2 row_a2 col_b2 row_b2 tile_N length_threadx length_thready]);
A2 = gpuArray(A2);
B2 = gpuArray(B2);
C2 = gpuArray(C2);

%512*768>1028, i.e., I have to (and should) use multiple blocks
k = parallel.gpu.CUDAKernel('matmul_share.ptx','matmul_share.cu');
k.ThreadBlockSize = [16 16 1]; %number of threads 16X16
k.GridSize = [768/16 512/16 1]; %make sure I have sufficient number of blocks
[C2] = feval(k,C2,A2,B2,index_d);
```

- A has dimension 512x512, and B has dimension 512x768. This requires more threads than 1028.
- I will use 16x16 thread blocks, meaning I petition A into $\frac{512}{16} = 32$ tiles.
- To cover the entire problem, I need 1536 blocks which I also organize in the x–y dimensions.

CUDA code

```
#include <math.h>

//matrix multiplication with shared memory. Threads need to be squared threads
__global__ void matmul(double* C, const double* A, const double* B, int* index)
{
    //get indexes
    int COL = blockIdx.x * blockDim.x + threadIdx.x; //COL in C
    int ROW = blockIdx.y * blockDim.y + threadIdx.y; //ROW in C

    int blockx = blockIdx.x;
    int xdim = blockDim.x;
    int threadx = threadIdx.x;
    int blocky = blockIdx.y;
    int ydim = blockDim.y;
    int thready = threadIdx.y;


    //read out size of matrices
    int col_a = index[0];
    int row_a = index[1];
    int col_b = index[2];
    int row_b = index[3];
    int tile_N = index[4]; //number of tiles: row_a/N_thready
    int length_threadx = index[5]; //number of threads in each tilde (and block) in x dimension
    int length_thready = index[6]; //number of threads in each tilde (and block) in y dimension


    //make sure not to leave the grid
    if (ROW < row_a && COL < col_b) {

        //determine the memory size of shared memory and preallocate matrices
        //size_t size_share = (tile_N * tile_N) * sizeof(double);
        __shared__ double a[(16 * 16) * sizeof(double)];
        __shared__ double b[(16 * 16) * sizeof(double)];
```

- I begin by reading in indexes.
- I have to allocate memory space in the cache. The amount of memory needed is 16^2 times the size of a double variable.
- Unfortunately, I cannot figure out to pass in C++ an index (e.g., 16) to a function that uses this index to initialize the size of an array.

CUDA code II

```
    double val = 0.0; //initialize the value of C(ROW,COL)
    for (int i1 = 0; i1 < tile_N; i1++) { //loop over partitions

        //save the corresponding matrices to shared memory
        a[threadx * length_thready + thready] = A[i1 * length_threadx * length_thready * tile_N + threadx * row_a + ROW];
        b[threadx * length_thready + thready] = B[COL * row_a + i1 * length_thready + thready]; //move in row dimension:

        //make sure all threads have loaded the data
        __syncthreads();

        //add the multiplication of the i1 partition
        for (int i2 = 0; i2 < 16; i2++) { //loop over entries in block
            val += a[i2 * length_thready + thready] * b[i2 + threadx * length_thready];
        }

        //make sure all threads are done
        __syncthreads();

    }
    C[COL * row_a + ROW] = val;


}

}
```

- $i1$ is the index for the number of tiles I have to loop over.
- Within each loop, I load to a and b the relevant entries from A and B.
- I have to make sure that all threads have loaded the data before using it. *__syncthreads*(); assures this.
- Then I do matrix multiplication for tile $i1$.

Value function iteration

Consider a simple household problem, where the household chooses assets given an exogenous interest rate r:

$$V(a,\epsilon) = \max_{c,a'} \left\{ \ln(c) + \beta \mathbb{E} V(a',\epsilon') \right\}$$

$$c + a' = \epsilon + a(1+r)$$

$$a' \geq \underline{a}$$

$$\pi_{jk}(\epsilon' = \epsilon^j | \epsilon = \epsilon^k)$$

Value function iteration II

- I solve the problem using on grid optimization.
- I use only 330 asset states and 3 productivity states. This way, I can use shared memory very efficiently.
  - I can have $abs(Vnew - Vold)$ in shared memory, thus, the loop can be inside the kernel.
  - I have $Vold$ in shared memory making updating fast.
- This serves expositional purposes to give an example of shared memory. 330 asset states with on grid search are obviously very few. With more grid points, I will have to use the global memory.

CUDA code

```
__global__ void VFI(double *Vnew, int *policy, double *Vold, const double *P, const double* util, const int Nz, const int Na, const double *par)
{

    int COL = blockIdx.x * blockDim.x + threadIdx.x;
    int ROW = blockIdx.y * blockDim.y + threadIdx.y;

    double beta = par[0];

    //initialization
    double EV;
    double val = 0;
    double val2 = 0;
    int N = Na * Nz;
    int i1;
    int pol;

    //declare shared variables
    __shared__ double V[990];
    __shared__ double distance[990];

    V[ROW + COL * Na] = Vold[ROW + COL * Na];
    //synchronize the local threads writing to the local memory cache
    __syncthreads();
```

- Note, I declare *distance* and V as shared memory. Each thread writes one element of the initial guess ($Vold$) into V.
- To make sure that V is filled up before proceeding, I use *__syncthreads*();.

CUDA code II

```
double crit = 10;
while (crit > 0.0001) {


    if (ROW < Na && COL < Nz) {
        val = -10000000;
        pol = 1;
        int i = 0;
        while (i < Na) { //possible capital choices

            //expected value function of
            EV = 0;
            for (int i2 = 0; i2 < Nz; i2++) {
                EV += beta * P[COL * Nz + i2] * V[i + i2 * Na];
            }
            val2 = util[i * N + ROW + COL * Na] + EV;
            if (val2 > val) {
                val = val2;
                pol = i;
                i = i + 1;
            }
            else {
                //break;
                i = Na;
            }
        }
    }
    Vnew[ROW + COL * Na] = val;
    policy[ROW + COL * Na] = pol;
    distance[ROW + COL * Na] = abs(Vnew[ROW + COL * Na] - V[ROW + COL * Na]);
```

- Now I simply compute for each thread one update of the value function as well as the distance to the old value function.
- Note, the utility of each choice (3rd dimension) given the states (*util*) was already computed in Matlab.

CUDA code III

```
        //synchronize the local threads writing to the local memory cache
        __syncthreads();

        //find maximum distance
        crit = distance[0];
        i1 = 1;
        while (i1 < N) {
            if (distance[i1] > crit) {
                crit = distance[i1];
            }
            i1++;
        }
        V[ROW + COL * Na] = Vnew[ROW + COL * Na];
        //synchronize the local threads writing to the local memory cache
        __syncthreads();

    }

}
```

- After each thread finishes an update, to know whether to stop, each thread has to evaluate whether the overall value function has converged.
- This may seem inefficient as all threads compute the same thing. As it uses shared memory, it is not particularly costly.

Golden Section Search

Another example where shared memory can be useful is Golden Section Search:

- The asset (capital) grid usually < 1024 making shared memory feasible.
- The algorithm accesses frequently the expected value function and the asset grid.

Hence, the following preparatory steps are natural:

1. Use as number of threads the length of the asset grid. Use blocks to fill out the other dimensions (my example has productivity).
2. Each thread computes the expected value function and reads it into shared memory.
3. Each thread reads the asset grid into shared memory.

CUDA code

```
if (ROW < Na && COL < Nz) {

    //calculate expected value function in shared memory for each asset state
    __shared__ double EV[128 * sizeof(double)]; //preallocate memory
    __shared__ double agrid_s[128 * sizeof(double)];
    double EVE = 0.0;
    for (int i2 = 0; i2 < Nz; i2++) {
        EVE += P[COL * Nz + i2] * V[ROW + i2 * Na];
    }
    EV[ROW] = EVE;
    agrid_s[ROW] = agrid[ROW];
    __syncthreads();
```

- My asset grid has 128 points, so I have to preallocate memory for this.
- Each thread (here the ROW), computes the expected value function. COL is given by the block. Moreover, it simply copies the agrid into shared memory.
- Note, the speed gain does not arise from pre-computing the expected value function instead of having each thread compute it multiple times in the algorithm. The gain comes from having faster memory access.
- I obtain speed gains of about 20%.

Matlab code

```
%% value function itteration GPU shared memory

dims = size(V_old);
k = parallel.gpu.CUDAKernel('golden_share.ptx','golden_share.cu');
% k = parallel.gpu.CUDAKernel('comp_EVE.ptx','comp_EVE.cu');
k.ThreadBlockSize = [1 dims(1) 1]; %number of threads
k.GridSize = [dims(2) 1 1]; %number of blocks
```

- My value function has dimension assets x productivity ($128x30$).
- The number of asset points is in the y–dimension of threads.
- The number of productivity points is in the x–dimension of blocks.

When is VFI fast on the GPU

- In general, a large state space favors the GPU, as long as each states takes about the same computing time, as all states can be solved in parallel.
- Algorithms that increase the arithmetic to memory load favor the GPU. For example, Golden Section Search is much faster on the GPU, particularly with non-linear interpolation.
- Higher order interpolation, for example, when doing off-grid search in multiple dimensions, also favors the GPU. A model with two dynamic states can quickly become non-feasible on the CPU but may remain feasible on the GPU.

Splitting up your problem

- Sometimes, even sequential problems can be split-up into parallel problems. Take as example exploiting the monotonicity of the policy function.
- Assume we have an endogenous state a and an exogenous state X (X could be multi dimensional) and we look for the policy $a' = f(a, X)$.
  - If X is small, probably the best you can do is solve the optimal policy a by solving it for each X separately on a CPU core.
  - If X is large, you can solve for $a'(a(1), X)$ using the GPU for each X and then use the policy $a'(a(1), X)$ to do the same for $a'(a(2), X)$, i.e., loop over a.
- Another obvious example are panel Monte Carlo simulations where for each period, you can use the GPU to simulate individuals.

So, when to use the GPU?

- You have a problem that can be parallelized for many grid points!
- For each grid point, the problem takes about the same time.
- The arithmetic to memory load is relatively high.
- You can avoid transfer between CPU and GPU memory.

Is Matlab/CUDA code particularly tedious?

- In many problems, you can achieve speed gains even without deeply thinking about memory access and, thus, making the C++ code relatively easy to write.
- Still, using Matlab as a wrapper makes debugging the CUDA code somewhat tedious. I usually write an additional pure Matlab code and debug until both give the same result.
- In my experience, all codes I write are somewhat tedious to debug:
  - In Matlab, I have to vectorize, stack, and reshape my arrays all the time leading to errors and tedious debugging.
  - With Fortran (C(++)) *.mex* files, the gateway code between Matlab and the code is annoying and debugging is terrible.
  - When doing everything in Fortran (or C++), I lose the advantages of interpreted code.
  - Julia may give the best mix as vectorization is not needed and just-in-time compiling gives you the advantage of an interpreted language. However, to obtain Fortran (C++) speed, you appear to have to write code that looks very similar in complexity to C++ code.