Languages for concurrency and distribution

Concurrency in CUDA

Leonardo Monchieri

Table of content

What is CUDA?

CUDA (Compute Unified Device Architecture)

- Parallel computing model
- GPU based
- Developed by Nvidia (proprietary)
- Available for many languages (CUDA C/C++, pyCUDA, CUDA Fortran, CUDA Matlab...)

CUDA C/C++

- Based on C/C++ standard
- NVCC compiler
- Small set of extensions to allow heterogeneous computing

Heterogeneous Computing

CPU vs GPU

CPUs: Latency Oriented

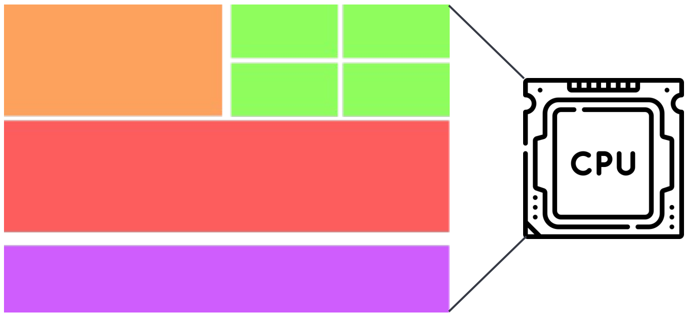
- Powerful ALU
- Large caches
- Sophisticated control:
  - Branch prediction
  - Data forwarding

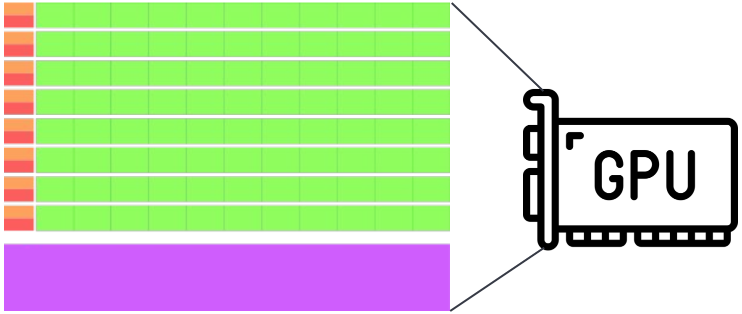
Faster on sequential code

GPUs : Throughput Oriented

- Small caches
- Simple control
- Energy efficient ALU
- Require massive number of threads to tolerate latency

Faster on parallel code



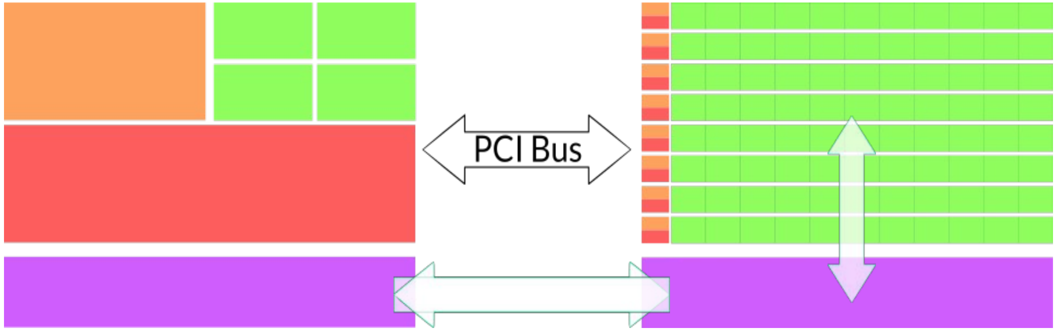


ALU
Control
Cache
DRAM

Host: CPU and its memory(host memory)

Device: GPU and its memory(device memory)

Processing workFlow



1) Copy input data from CPU to GPU memory
2) Load GPU code and execute it, caching data on chip for performance
3) Copy results from GPU memory to CPU memory

Device Code

Host Code

```
#include <iostream>
#include <cuda.h>

// CUDA kernel (Device code) that runs on the GPU
__global__ void square(float *d_array, int size) {

    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < size) {
        d_array[idx] = d_array[idx] * d_array[idx];
    }
}

int main() {
    /*---------1---------*/
    const int arraySize = 10;   // Array size
    float *d array, *h array;
    h_array = (float*)malloc(arraySize*sizeof(float));   // Host array

    // Allocate device memory
    cudaMalloc(&d_array, arraySize * sizeof(float));    // Device array

    // Initialize array on host
    for (int i = 0; i < arraySize; i++) {
        h_array[i] = i + 1; // Array of {1, 2, 3, ..., 10}
    }

    //Copy the host data into device memory
    cudaMemcpy(d_array, h_array, arraySize * sizeof(float), cudaMemcpyHostToDevice);

    /*---------2---------*/
    //Launch kernel on the device with 1 block and 10 threads(one per array element)
    square<<<1, arraySize>>>(d_array, arraySize);

    /*---------3---------*/
    // Copy the result back to the host
    cudaMemcpy(h array, d array, arraySize * sizeof(float), cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();

    // Free device memory
    cudaFree(d array);
    free(h_array);
}
```

In the code

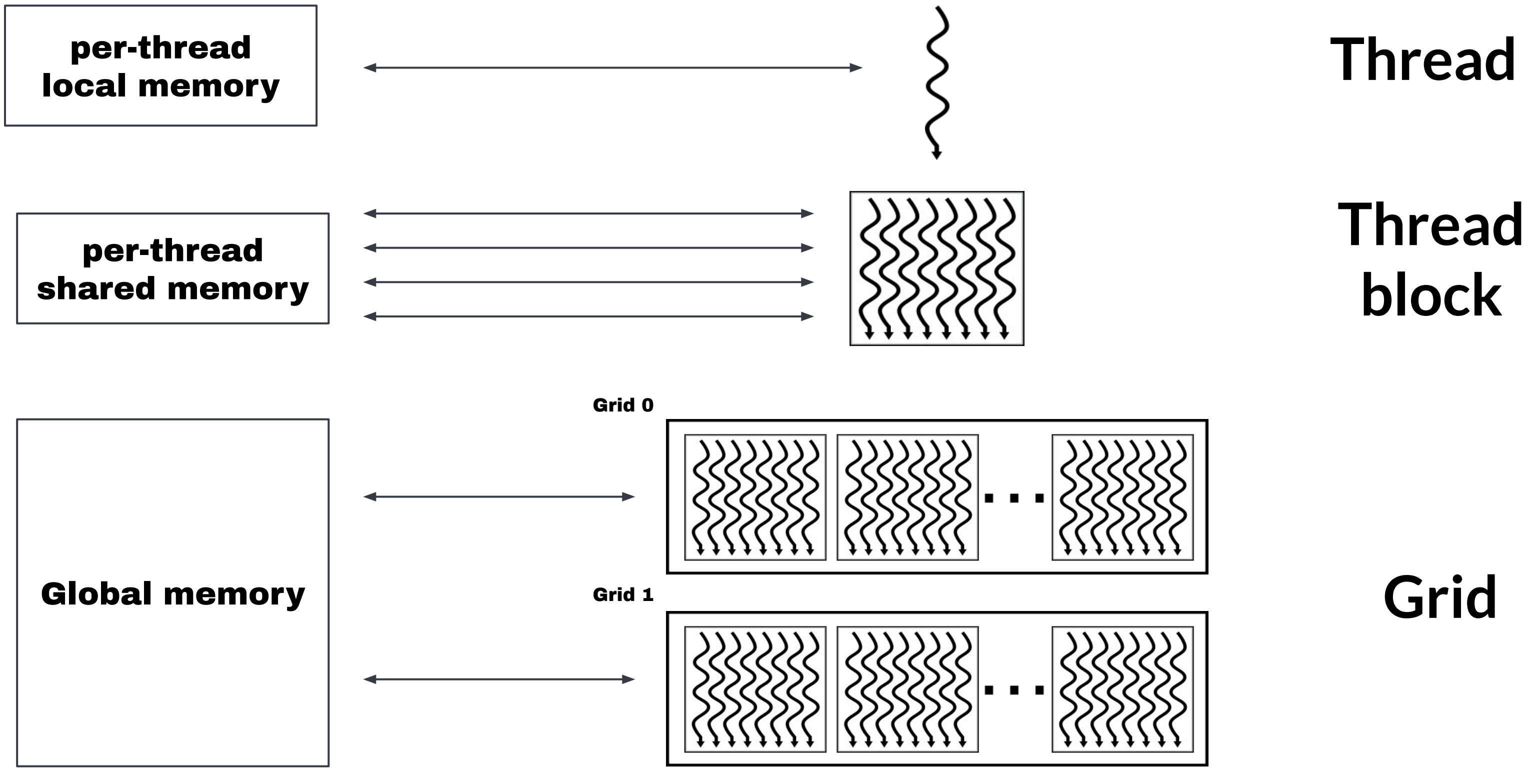
➔ ***CudaMalloc()*** : Allocate memory on the device.

➔ ***CudaMemCpy()***: copy values from host to device(and viceversa).

➔ ***CudaFree()***: Frees memory that was previously allocated on the device.

CUDA Paradigm

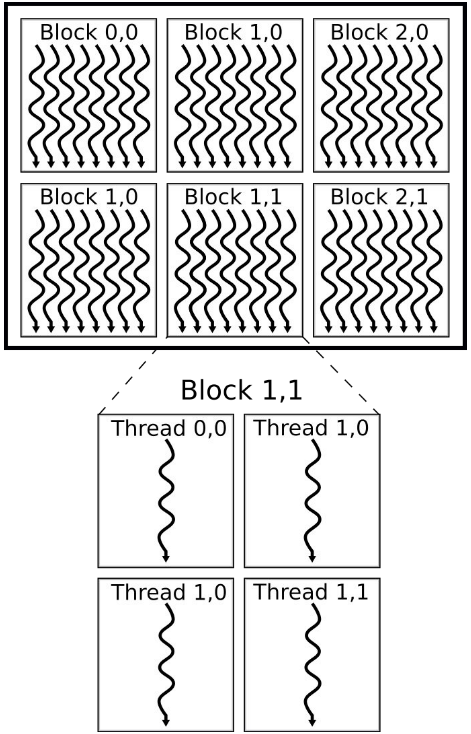
Threads Hierarchy



Kernel

```
__global__ void kernelEx(params){...}
kernelEx<<<(3,2),(2,2)>>>(params);
```

- Functions or full programs.
- Executed in parallel.
- Define hierarchy of grids of thread blocks.
- Support a synchronization mechanism.
- Treads are identified by their thread index (*threadIdx*), block index (blockIdx).



Kernel in the code

➔ Keyword ***__global__*** in the function header.

➔ `kName<<<gridDim, blockDim>>>(params)` to launch the kernel from the CPU.

➔ ***cudaDeviceSynchronize()*** blocks the host until all previously launched CUDA tasks on the device have completed execution

Example : array addition

```
#include <iostream>
#include <cuda_runtime.h>

#define N 1024  // Total number of elements in the array
#define THREADS_PER_BLOCK 256  // Number of threads per
block
#define BLOCKS 4  // Number of threads per block

// DEVICE code
// CUDA Kernel for vector addition
__global__ void add(int *a, int *b, int *c, int n) {
   // Calculate the global index for each thread
   int index = threadIdx.x + blockIdx.x * blockDim.x;

   // Ensure that we don't go out of bounds
   if (index < n) {
       c[index] = a[index] + b[index];
   }
}
```

```
// HOST code
int main() {
   // Host arrays
   int h_a[N], h_b[N], h_c[N];

   // Initialize the host arrays with some values
   for (int i = 0; i < N; i++) {
       h_a[i] = i;
       h_b[i] = i * 2;
   }

   // Device arrays
   int *d_a, *d_b, *d_c;

   // Allocate memory on the GPU for the device arrays
   cudaMalloc((void **)&d_a, N * sizeof(int));
   cudaMalloc((void **)&d_b, N * sizeof(int));
   cudaMalloc((void **)&d_c, N * sizeof(int));

   // Copy data from the host arrays to the device arrays
   cudaMemcpy(d_a, h_a, N * sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy(d_b, h_b, N * sizeof(int), cudaMemcpyHostToDevice);

   // Launch the kernel with multiple blocks and threads
   add<<<BLOCKS, THREADS_PER_BLOCK>>>(d_a, d_b, d_c, N);

   // Wait for the GPU to finish executing the kernel
   cudaDeviceSynchronize();

   // Copy the result from the device array d_c to the host array h_c
   cudaMemcpy(h_c, d_c, N * sizeof(int), cudaMemcpyDeviceToHost);

   // Print the result
   std::cout << "Result of vector addition:\n";
   for (int i = 0; i < N; i++) {
       std::cout << h_a[i] << " + " << h_b[i] << " = " << h_c[i] << "\n";
   }

   // Free the device memory
   cudaFree(d_a);
   cudaFree(d_b);
   cudaFree(d_c);

   return 0;
}
```

Threads communication

Communication and memory

- *Shared Memory:* small amount of data
  - Only from threads of the same block
  - Use Streaming Multiprocessor(SM)
  - Space splitted equally among threads blocks ➞ limited amount of memory
- *Global Memory:* for bigger amount data
  - Any thread
  - Slower
  - Synchronization can be an issue

Kernel Structure is <u>Crucial</u>!

Communication in the code

➔ ***`__shared__`*** to declare a variable in the shared memory.
➔ **`__device__`** to declare a variable in the global memory.
➔ ***`__synchtrhreads()`*** to sync the threads in the block.
➔ ***`__threadfeence()`*** ensure that a writing operation is concluded before subsequent reads or writes from other threads can occur.

Atomic Operations

- Prevent race condition
- Can be performed on shared and global memory
- Hardware acceleration
- No synchronization needed
- Performance trade-of

Atomic operations in the code

CUDA provides many atomic operations:

➔ ***`T atomicAdd(T* address, T val)(or atomicSubs)`***: adds a value from a memory location and returns the old value.

➔ ***`T atomicExch(T* address, T val)`***: replaces the value at a memory location with a new value, and returns the old value.

➔ ***`T atomicCAS(T* address, T compare, T val)`***: compares the value at a memory location with an "expected" value and, if they are equal, swaps it with a new value. It returns the old value regardless of whether the swap occurred.

For the complete list of CUDA atomic operations: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

Example : Philosopher's dinner

```
const int NUM_PHILOSOPHERS= 5;

//Global variables
__device__ int d_forks[NUM_PHILOSOPHERS]; // Array of forks (0 = free, 1 = taken)
__device__ int d_philosophers[NUM_PHILOSOPHERS]; // Philosopher states (0 = thinking, 1 = eating)

//DEVICE code
__global__ void philosopher_dinner (int num_philosophers ) { // Kernel function for philosopher
behavior
    int id = threadIdx .x;

    // Each philosopher alternates between thinking and trying to eat
    while (true) {
        // Thinking
        d_philosophers [id] = 0; // Set philosopher state to thinking
        printf ("Philosopher %d is thinking. \n", id);

        __syncthreads ();
        __threadfence ();

        // Try to pick up the two forks (left and right)
        int left_fork = id;
        int right_fork = (id + 1) % num_philosophers ;

        // Pick up forks only if both are available (use atomic operations to prevent race
conditions)
        if (atomicCAS (&d_forks [left_fork ], 0, 1) == 0 && atomicCAS (&d_forks [right_fork ], 0, 1)  ==
0)
            {
            // Now the philosopher can eat
            d_philosophers [id] = 1; // Set philosopher state to eating
            printf ("Philosopher %d is eating. \n", id);

            // Simulate eating
            for (int i = 0; i < 1000000; ++i); // Busy-waiting

            // Put down the forks after eating
            atomicExch (&d_forks [left_fork ], 0);
            atomicExch (&d_forks [right_fork ], 0);
        }
        else{
            atomicExch (&d_forks [left_fork ], 0);
            atomicExch (&d_forks [right_fork ], 0);
        }

        __syncthreads ();
        __threadfence ();
    }
```

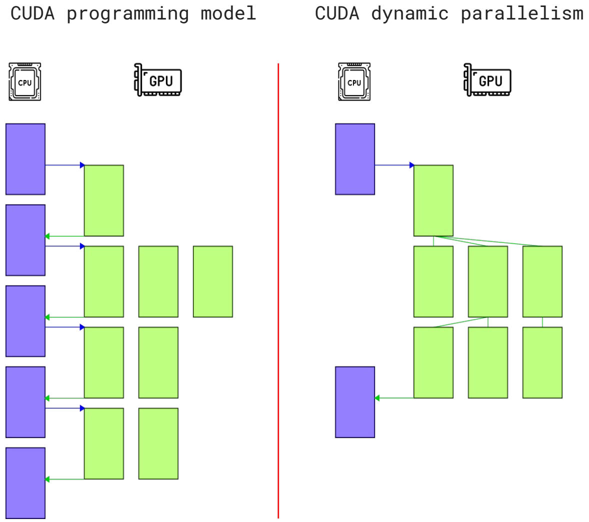
```
//HOST code
int main () {
    // Initialize philosophers and forks arrays
    int h_forks [NUM_PHILOSOPHERS ] = {0}; // 0 = fork is free
    int h_philosophers [NUM_PHILOSOPHERS ] = {0}; // 0 = thinking, 1 = eating

    // Copy data to device
    cudaMemcpyToSymbol (d_forks , h_forks , sizeof (int) * NUM_PHILOSOPHERS );
    cudaMemcpyToSymbol (d_philosophers , h_philosophers ,
          sizeof (int) * NUM_PHILOSOPHERS );

    // Launch kernel: each philosopher runs on its own CUDA thread
    philosopher_dinner <<<1, NUM_PHILOSOPHERS >>>(NUM_PHILOSOPHERS );

    // Wait for the kernel to complete
    cudaDeviceSynchronize ();

    return 0;
}
```

Dynamic Parallelism

Dynamic Parallelism

Allow Kernel (*parent*) launch other kernels (*childs*)

- Runtime launch
- launched from the GPU
- High scalability
- Recursive algorithm optimisation
- Require <u>explicit</u> synchronization



Dynamic Parallelism in code

➔ Simply call the child kernel inside the parent using **<<<>>>**

```
  global   void parentKernel(){
      childKernel<<<blocks,threads>>>();
}
```

Streams and Event

CUDA streams introduce concurrency between different operations (kernels, memory transfers) on the GPU:

- kernels can be assigned to a specific stream
- Asynchronous execution
- Concurrency between streams

Synchronization can be achieved in different ways:

Explicit:

- Synchronize everything
- Synchronize w.r.t. a specific stream
- Using events

Implicit:

- Device memory allocation
- Non-Async version of memory operations
- Page-locked memory allocation

Streams in code

Streams

➔ ***cudaStreamCreate***(*&stream*)/**cudaStreamDestroy**(*stream*): to manage stream creation and destruction

➔ ***cudaStreamSynchronize(***&stream***)***: blocks host until all CUDA calls in the stream are complete

Events

➔ ***cudaEventCreate***(*&event*)/***cudaEventDestroy***(*event*): to manage event creation and destruction

➔ ***cudaEventRecord***(*event, stream*): to signal completion of the event task

➔ ***cudaWaitEvent***(*stream2, event*): to wait an event before lunch a new stream

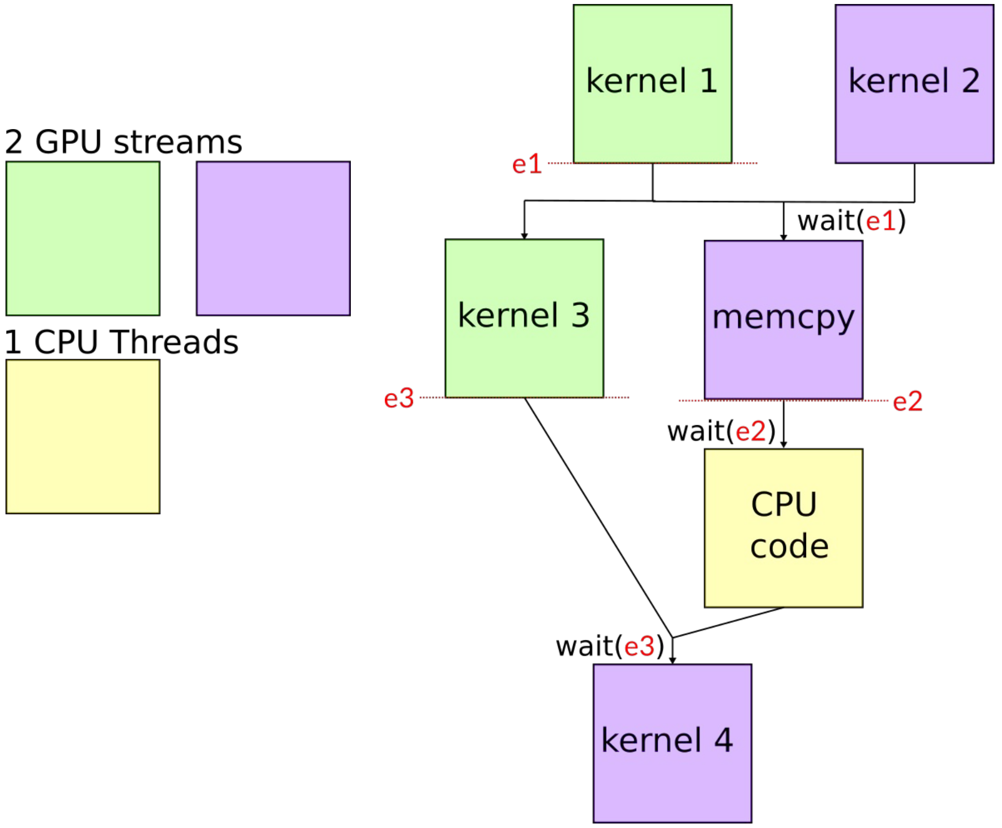
➔ ***cudaEventSynchronize***(*event*): to synchronize computation till a specific event occurs

Pinned Memory

➔ ***cudaMallocHost(****&host, size****)***: lock host memory to prevent paging.

➔ ***cudaMemCpyAsync()***: copy values from host to device(and viceversa) asynchronously.

```
kernel1<<<,,,a>>>();

cudaEventRecord(e1, a);
kernel2<<<,,,b>>>();

cudaStreamWaitEvent(b, e1);
cudaMemcpyAsync(,,,,b);


cudaEventRecord(e2, b);
kernel3<<<,,,a>>>();
cudaEventRecord(e3, a);

cudaEventSynchronize(e2);

   CPU code

cudaStreamWaitEvent(b, e3);
kernel4<<<,,,b>>>();
```



Final Consideration

Final Considerations

- *Memory management* to minimize transfer between CPU(*host*) and GPU(*device*).
- *Thread hierarchy design* is crucial, carefully design grid and block dimensions.
- Correct usage of *Atomic operations* and *synchronization* to avoid race conditions and maintain paralellism.
- Maximize parallelism using *dynamic parallelism* and *streams.*
- Performance profiling: NVIDIA provides tools like *Nsight* to identify bottlenecks and optimize performance.

END

Thanks for your attention

References

- Fang, Jianbin, et al. "Parallel programming models for heterogeneous many-cores: a comprehensive survey." *CCF Transactions on High Performance Computing* 2 (2020): 382-400.

- https://ams148-spring18-01.courses.soe.ucsc.edu/home.html (Lectures 2, 3 and 8)

- Nickolls, John, et al. "Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for?." *Queue* 6.2 (2008): 40-53.

- https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles/