EURO
LUXEMBOURG

Introduction to GPU programming

Dr. Ezhilmathi Krishnasamy
Parallel Computing and Optimization Group (PCOG), University of Luxembourg (UL), Luxembourg

EURO
LUXEMBOURG

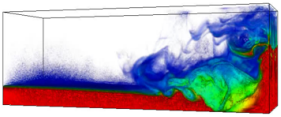
What we are going to discuss

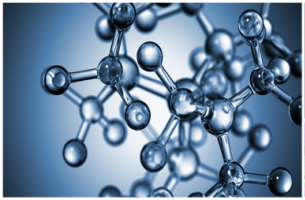
- GPU and CPU architecture overview and comparison
  - streaming multiprocessors, memory hierarchy, threads blocks, etc,.
- CUDA programming model
  - programming structure, thread hierarchy, device call, etc,.
- Memory management
  - unified memory, explicit memory copy, etc,.
- Examples in numerical linear algebra
  - vector multiplication, vector addition, etc,.
- A quick demo session with some examples

Motivation: Why we need supercomputers

- Powerful computers will help to unlock the secrets in science and engineering

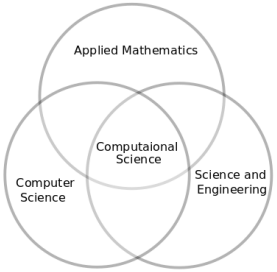
- Astrophysics
- CFD:turbulence
- Bioinformatics
- Material science

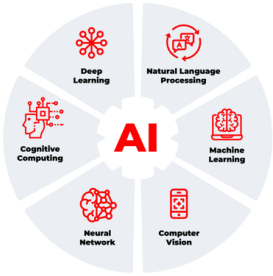




Motivation: Why we need supercomputers

- We need to do lots of arithmetic computation in science & engineering and artificial intelligence
- For example, in science and engineering, problems are defined by partial differential equations (PDEs)
- PDEs are converted into a system of equations by using numerical methods (e.g., finite difference and finite element methods), where we need to find the values for the unknown variables
- Similarly, in artificial intelligence, we end up solving matrices and vectors

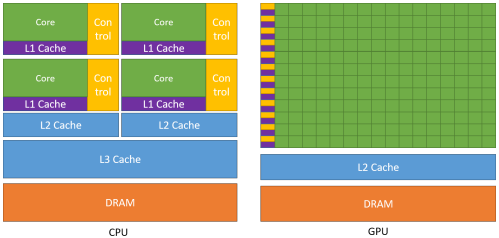




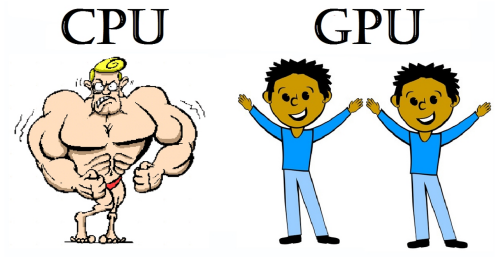
$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Important differences between CPU and GPU

- GPU has many cores compared to CPU
- But on the other hand, the CPU's frequency is higher than the GPU. That makes the CPU faster in computing compared to GPU
  - Intel® Core™ i7-10700K Processor base frequency is 3.80 GHz, whereas, Nvidia Ampere has 0.765 GHz
- However, GPU can handle many threads in parallel, which can process many data in parallel
- In the GPU, cores are grouped into GPU Processing Clusters (GPCs), and each GPCs has its own Streaming Multiprocessors (SMs) and Texture Processor Clusters (TPCs)
- Nvidia (microarchitecture): Tesla (2006), Fermi (2010), Kepler (2012), Maxwell (2014), Pascal (2016), Volta (2017), Turing (2018), and Ampere (2020)
- Video Link: Mythbusters Demo GPU versus CPU



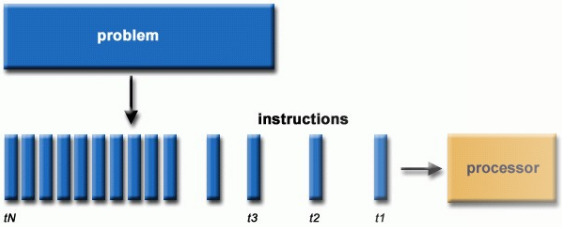
Source:Nvidia: CUDA programming



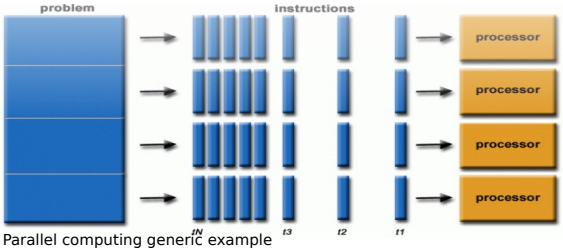
Serial programming vs. parallel programming

- Serial programming:
  - An entire problem can be divided in to discrete series of instructions
  - All the instructions are executed one by one
  - Executed by single thread or processor
  - Only one instruction can be executed at the same time
- Parallel programming
  - An entire problem can be divided into discrete parts such way that it can be solved concurrently
  - Each part may have set of instructions
  - Each parts instructions are executed on different thread/processor
  - Since it is a parallel execution, a target problem needs to be controlled/coordinated
- CPU, GPU, and other parallel processor can perform the parallel computing

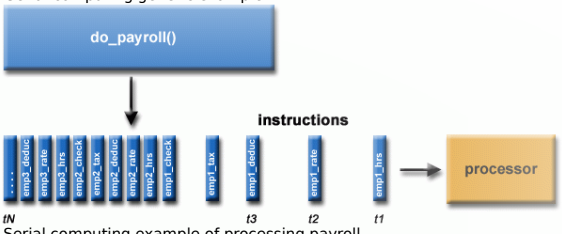
Serial programming vs. parallel programming



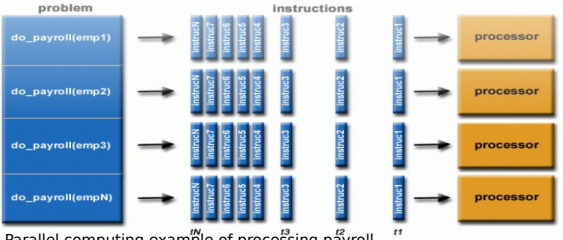
Serial computing generic example



Parallel computing generic example



Serial computing example of processing payroll



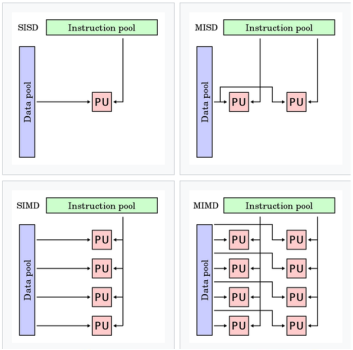
Parallel computing example of processing payroll

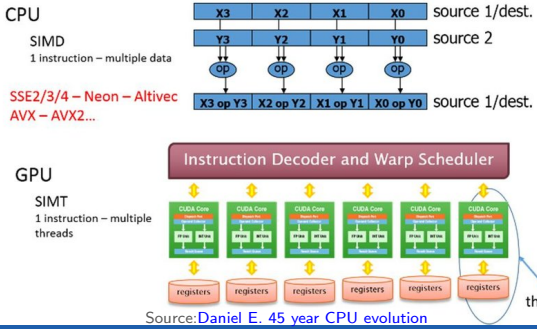
Source: HPC LLNL

GPU architecture

- Computer architecture is characterized by 4 according to Flynn's taxonomy
  - Single instruction stream, single data stream (SISD)
  - Single instruction stream, multiple data streams (SIMD)
  - Multiple instruction streams, single data stream (MISD)
  - Multiple instruction streams, multiple data streams (MIMD)

- GPUs are based on Single Instruction Multiple Threads (SIMT)

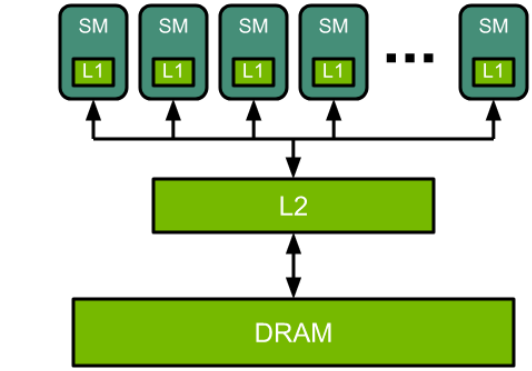




Source:Daniel E. 45 year CPU evolution

GPU architecture

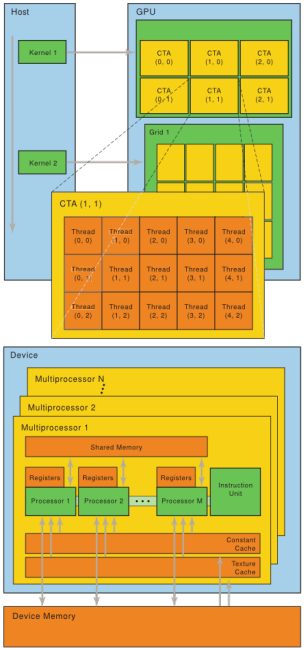
- Ampere GPU had seven GPCs, 42 TPCs, and 84 SMs.
- Volta GPU has six GPCs, each GPC has a seven TPCs (each including two SMs), and 14 SMs.
- Each SMs has L1 cache (up to 128 KB) and L2 (up to 6144 KB) cache is shared between the GPCs.
- RT (Ray Tracing) cores dedicated to do the ray-tracing rendering math computation.
- Tensor Cores: provides the speedups for AI neural network training computation.
- Programmable Shading Cores, which has a CUDA cores.



Source: Nvidia: deep learning

GPU architecture

- SIMT enables programmers to achieve thread-level parallelism in streaming multiprocessors (SMs)
- The multiprocessor *occupancy* is the ratio of active warps to the maximum number of warps supported on the GPU's multiprocessor
- SMs in the GPU are based on the scalable array multi-thread, which allows grid and thread blocks of 1D, 2D, and 3D data
- Programmers can write the grid and block size to create a thread when executing the device kernel; this thread block is typically called a cooperative thread array (CTA)
- A parallel execution is happening in the SMs via *warps* and one warp contains 32 threads



Source: Nvidia: Parallel Thread Execution

EURO
LUXEMBOURG

Usage of compute capabilities in different Nvidia GPU architecture

Compute capability (flag)	**Architecture support**
sm_35, and sm_37	Basic features + Kepler support + Unified memory programming + Dynamic parallelism support
sm_50, sm_52 and sm_53	+ Maxwell support
sm_60, sm_61, and sm_62	+ Pascal support
sm_70 and sm_72	+ Volta support
sm_75	+ Turing support
sm_80, sm_86 and sm_87	+ NVIDIA Ampere GPU architecture support

Thread organization

- Threads are organized within a Grids and Blocks. These Grids and Blocks can be in 1D, 2D or 3D. And these are declared as *dim3*
- Example: 2D grid and thread block

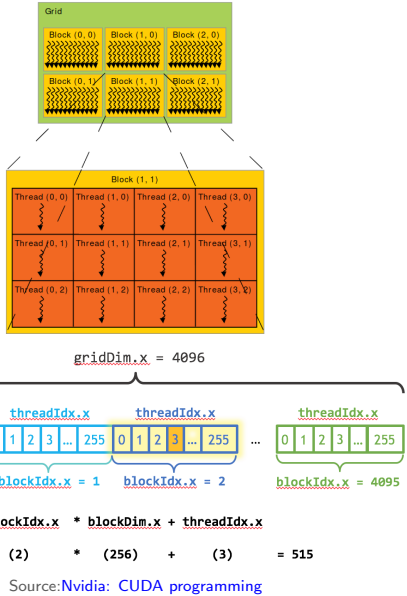
```
dim3 Grid(3, 2, 1); # two dimentional grid
dim3 Block(4, 3, 1); # two dimentinal thread block
```

- Example: 1D grid and thread block

```
dim3 Grid(4096, 1, 1); # one dimentional grid
dim3 Block(256, 1, 1); # one dimentinal thread block
```

- Example: calling thread block in the main program

```
Hello_world(); # calling c function
Hello_world<<<Grid, Block>>>(); calling cuda device function
```



Source:Nvidia: CUDA programming

EURO LUXEMBOURG

Thread organization

- Dimension variables:
  - **gridDim** specifies the number of blocks in the grid
  - **blockDim** specifies the number of threads in each block
- Index variables:
  - **blockIdx** gives the index of the block in the grid
  - **threadIdx** gives the index of the thread within the block

CUDA API

- *cudaMalloc()* allocates device memory
- *cudaMemcpy()* transfers data to or from a device
- *cudaFree()* frees device memory that is no longer in use
- *__syncthreads()* synchronizes threads within a block
- *cudaDeviceSynchronize()* effectively synchronizes all threads in a grid
- *cudaMallocManaged()* for allocating unified memory

Major comparison between Turing vs. Ampere

Graphics Card	GeForce RTX 2080 Founders Edition	GeForce RTX 3080 10GB Founders Edition
GPU Codename	TU104	GA102
GPU Architecture	**Nvidia Turing**	**Nvidia Ampere**
GPCs	6	6
TPCs	23	34
SMs	46	68
CUDA Cores / SM	64	128
CUDA Cores / GPU	2944	8704
Tensor Cores / SM	8 (2nd Gen)	4 (3rd Gen)
Tensor Cores / GPU	368	272 (3rd Gen)
RT cores	46 (1st Gen)	68 (2nd Gen)

Source:Nvidia Ampere

Compute capabilities for latest Nvidia GPUs

Data Center GPU	**Nvidia V100**	**Nvidia A100**	**Nvidia H100**
GPU architecture	Nvidia Volta	Nvidia Ampere	Nvidia Hopper
Compute Capability	7	8	9
Thread / Warp	32	32	32
Max Warps / SM	64	64	64
Max Threads / SM	2048	2048	2048
Max Thread Blocks (CTAs) / SM	32	32	32
Max Threads Blocks / Thread Block Clusters	NA	NA	NA
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Thread Block	65536	65536	65536
Max Registers / Thread	255	255	255
Max Thread Block Size (#of threads)	1024	1024	1024
FP32 Cores / SM	64	64	64
Ratio of SM Registers to FP32 Cores	1024	1024	1024
Shared Memory Size / SM	Configurable up to 96 KB	Configurable up to 164 KB	Configurable up to 228 KB

Source:Nvidia H100

CUDA function qualifiers and variable memory space specifiers

Qualifier	Description
__device__	These functions are executed only from the device and callable only from device
__global__	These functions are executed from the device, and it can be callable from the host and device (only for compute capabilities 3.2 or higher)
__host__	These functions are executed from a host, and callable only from the host
__noninline__ *__forceinline__*	Compiler directives instruct the functions to be inline or not inline

Variable	Memory	Scope	Lifetime
__device__	Global	Grid (entire grid of thread blocks)	Application
__constant__	Constant	Grid (entire grid of thread blocks)	Application
__shared__	Shared	Block (within a thread block)	Block

Hello world

- Run a part or entire application on the GPU
- Call *cuda_function* on device
- It should be called using function qualifier *__global__*
- Calling the device function on the main program:
  - C/C++ example, *c_function()*
  - CUDA example, *cuda_function<<<1,1>>>()* (just using 1 thread)
- <<< >>>, specify the threads blocks within the bracket
- Make sure to synchronize the threads
  - __syncthreads(); synchronizes all the threads within a thread block
  - CudaDeviceSynchronize(); synchronizes a kernel call in host
- Most of the CUDA API are synchronized call by default (but sometimes it is good to call explicit synchronized call to avoid error in the computation)

```c
// hello-world.c
#include <stdio.h>
void c_function()
{
  printf("Hello World!\n");
}

int main()
{
  c_function();
  return 0;
}
```

```c
// hello-world.cu
#include <stdio.h>
__global__ void cuda_function()
{
   printf("Hello World from GPU!\n");
    __syncthreads();                // to synchronize all threads
}

int main()
{
    cuda_function<<<1,1>>>();
    cudaDeviceSynchronize();      // to synchronize device call
    return 0;
}
```

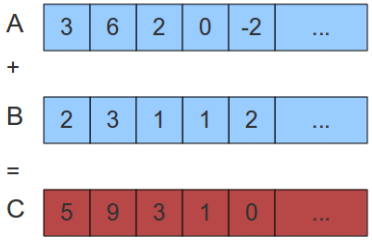
Vector addition

- Memory allocation on both CPU and GPU

```
// Initialize the memory on the host
float *a, *b, *out;

// Allocate host memory
a   = (float*)malloc(sizeof(float) * N);
b   = (float*)malloc(sizeof(float) * N);
out = (float*)malloc(sizeof(float) * N);

// Initialize the memory on the device
float *d_a, *d_b, *d_out;

// Allocate device memory
cudaMalloc((void**)&d_a, sizeof(float) * N);
cudaMalloc((void**)&d_b, sizeof(float) * N);
cudaMalloc((void**)&d_out, sizeof(float) * N);
```

- Fill values for host vectors a and b

```
// Initialize host arrays
for(int i = 0; i < N; i++)
  {
    a[i] = 1.0f;
    b[i] = 2.0f;
  }
```



Vector addition

- Transfer initialized value from CPU to GPU

```
// Transfer data from host to device memory
cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);
```

- Creating a 2D thread block

```
// Thread organization
dim3 dimGrid(1, 1, 1);
dim3 dimBlock(8, 8, 1);
```

- Calling the kernel function

```
// execute the CUDA kernel function
  vector_add<<<dimGrid, dimBlock>>>(d_a, d_b, d_out, N);
```

- Copy back computed value from GPU to CPU

```
// Transfer data back to host memory
cudaMemcpy(out, d_out, sizeof(float) * N, cudaMemcpyDeviceToHost);
```

Vector addition

- Vector addition function call

```
// GPU function that adds two vectors
__global__ void vector_add(float *a, float *b,
           float *out, int n)
{

  int i = blockIdx.x * blockDim.x * blockDim.y +
    threadIdx.y * blockDim.x + threadIdx.x;
  // Allow the   threads only within the size of N
  if(i < n)
    {
      out[i] = a[i] + b[i];
    }

  // Synchronice all the threads
  __syncthreads();
}
```

- Release the host and device memory

```
// Deallocate device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_out);

// Deallocate host memory
free(a);
free(b);
free(out);
```

- Source:Vector-Addition.cu

Matrix multiplication

Matrix multiplication function in C/C++

```
float * matrix_mul(float *h_a, float *h_b, float *h_c, int width)
{
  for(int row = 0; row < width ; ++row)
    {
      for(int col = 0; col < width ; ++col)
        {
          float single_entry = 0;
          for(int i = 0; i < width ; ++i)
            {
              single_entry += h_a[row*width+i] * h_b[i*width+col];
            }
          h_c[row*width+col] = single_entry;
        }
    }
  return h_c;
}
```

Source: Matrix-Multiplication.cu

Matrix multiplication function in CUDA

```
__global__ void matrix_mul(float* d_a, float* d_b,
                           float* d_c, int width)
{

  int row = blockIdx.x * blockDim.x + threadIdx.x;
  int col = blockIdx.y * blockDim.y + threadIdx.y;

  if ((row < width) && (col < width))
    {
      float single_entry = 0;
      // each thread computes one
      // element of the block sub-matrix
      for (int i = 0; i < width; ++i)
        {
          single_entry += d_a[row*width+i]*d_b[i*width+col];
        }
      d_c[row*width+col] = single_entry;
    }
}
```

Matrix multiplication

- Allocating the CPU and GPU memory for A,B, and C matrix

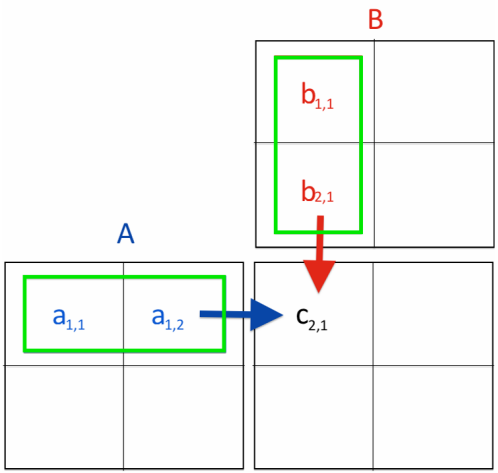
```
// Initialize the memory on the host
float *a, *b, *c;

// Allocate host memory
a = (float*)malloc(sizeof(float) * (N*N));
b = (float*)malloc(sizeof(float) * (N*N));
c = (float*)malloc(sizeof(float) * (N*N));

// Initialize the memory on the device
float *d_a, *d_b, *d_c;

// Allocate device memory
cudaMalloc((void**)&d_a, sizeof(float) * (N*N));
cudaMalloc((void**)&d_b, sizeof(float) * (N*N));
cudaMalloc((void**)&d_c, sizeof(float) * (N*N));
```

- Transfer initialized A and B matrix from CPU to GPU

```
// Transfer data from host to device memory
cudaMemcpy(d_a, a, sizeof(float) * (N*N), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float) * (N*N), cudaMemcpyHostToDevice);
```

- 2D thread block for indexing x and y

```
// Thread organization
int blockSize = 32;
dim3 dimBlock(blockSize,blockSize,1);
dim3 dimGrid(ceil(N/float(blockSize)),ceil(N/float(blockSize)),1);
```



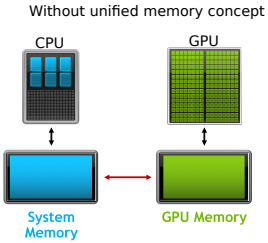
Unified Memory

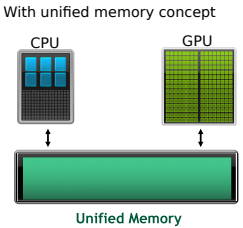
Without unified memory

- Allocate the host memory
- Allocate the device memory
- Initialize the host value
- Transfer the host value to device memory location
- Do the computation using the CUDA kernel
- Transfer the data from the device to host
- Free device memory
- Free host memory

With unified memory

- ~~Allocate~~ ~~the host memory~~
- Allocate the device memory
- Initialize the host value
- ~~Transfer the host value to device memory location~~
- Do the computation using the CUDA kernel
- ~~Transfer the data from the device to host~~
- Free device memory
- ~~Free host memory~~





Source: Vector-Addition-Unified.cu

Unified Memory

Use *cudaMallocManaged()*

```
/*
// Initialize the memory on the host
float *a, *b, *out;

// Allocate host memory
a   = (float*)malloc(sizeof(float) * N);
b   = (float*)malloc(sizeof(float) * N);
out = (float*)malloc(sizeof(float) * N);
*/

// Initialize the memory on the device
float *d_a, *d_b, *d_out;

// Allocate device memory
cudaMallocManaged(&d_a, sizeof(float) * N);
cudaMallocManaged(&d_b, sizeof(float) * N);
cudaMallocManaged(&d_out, sizeof(float) * N);
```

Do not forget to call *cudaDeviceSynchronize()* after a kernel call

EURO
LUXEMBOURG

Thank you

If you any question or research collaboration, please contact
ezhilmathi.krishnasamy@uni.lu

If you interested to learn more about CUDA and OpenACC programming, please refer to PRACE MOOC: GPU Programming for Scientific Computing and Beyond (given by Prof. Pacal Bouvry and Dr. Ezhilmathi Krishnasamy)