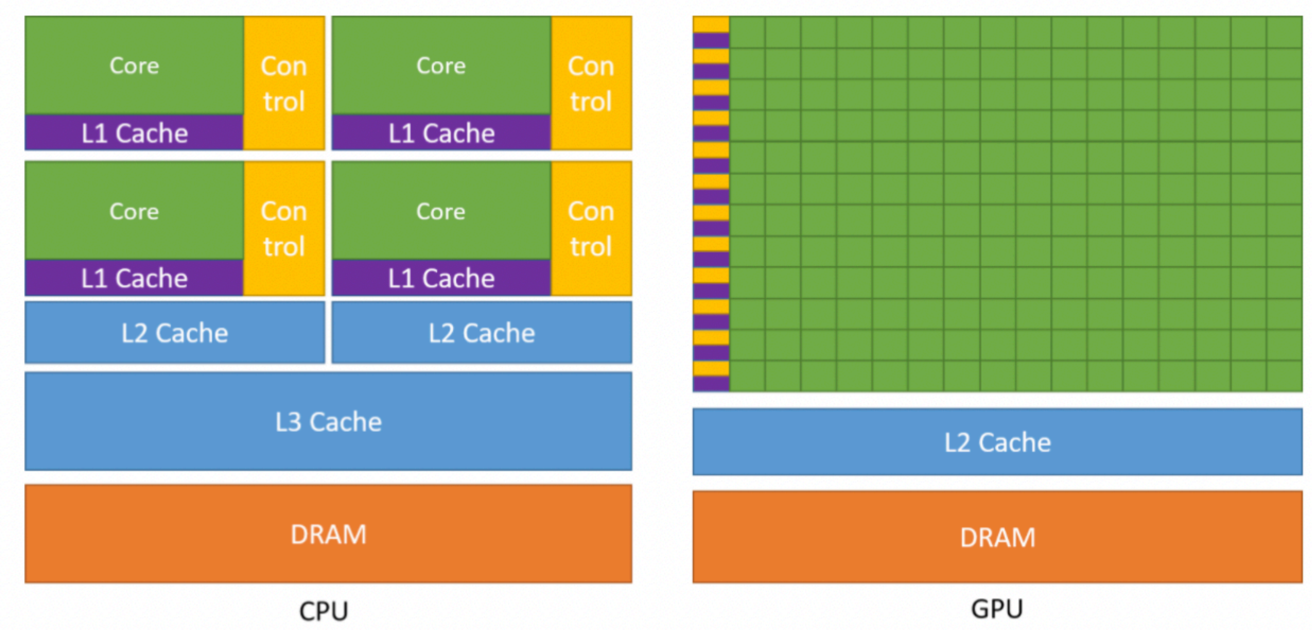
GPU Acceleration

GPU Programming

GPU Introduction

CPU 强调通用型，每个核都有相应的控制单元和 cache, 不同核可以独立并行地完成各自的任务，甚至在单核内可以通过上下文切换执行不同的程序。

GPU 强调相同或相似操作的并行性，不需要过高的自由度，因此 GPU 的核数比 CPU 多得多，且每个核执行的任务相对单一，控制单元的控制能力较弱，通常用于控制若干个核的行为。



GPU Programming Mode: SIMT

SIMT: Single Instruction Multiple Threads

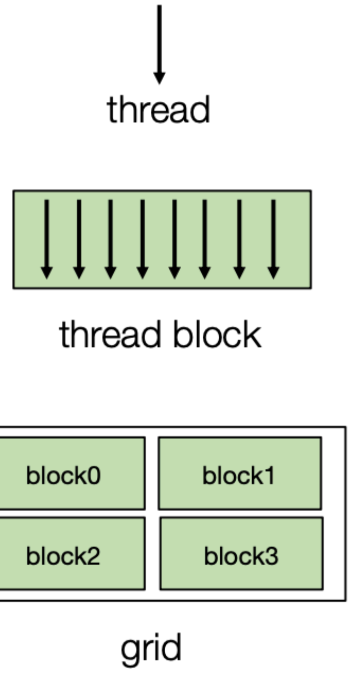
即多线程执行相同的代码，但数据路径可能不同

GPU 的三层架构 thread - thread block - launch grid：

- 若干个线程组成 thread block, 相同 thread block 内的线程有共享内存
- Thread blocks 组成 launch grid

一个 GPU 核对应一个 launch grid

> 此为 CUDA 的术语，不同 GPU 编程模型 (opencl, sycl, metal) 的术语有对应关系



Example: Vector Add

i (global offset)	0	1	2	3	4	5	6	7
threadIdx.x	0	1	2	3	0	1	2	3
blockIdx.x	0				1			

Suppose each block includes 4 threads: blockDim.x = 4

```
__global__ void VecAddKernel(float* A, float *B, float* C, int n) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < n) {
    C[i] = A[i] + B[i];
  }
}
```

除了执行部分，host 端还需要完成 GPU 上的内存分配、数据拷贝等操作。

Host side 的代码为：

```
__global__ void VecAddKernel(float* A, float *B, float* C, int n) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < n) {
    C[i] = A[i] + B[i];
  }
}
void VecAddCUDA(float* Acpu, float *Bcpu, float* Ccpu, int n) {
  float *dA, *dB, *dC;
  cudaMalloc(&dA, n * sizeof(float)); // cudaMalloc 分配在 Global memory
  cudaMalloc(&dB, n * sizeof(float));
```

```
    cudaMalloc(&dC, n * sizeof(float));
    cudaMemcpy(dA, Acpu, n * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dB,  Bcpu, n * sizeof(float), cudaMemcpyHostToDevice);
    int threads_per_block = 512;
    int nblocks = (n + threads_per_block - 1) / threads_per_block;
    VecAddKernel<<<nblocks, thread_per_block>>>(dA, dB, dC, n);
    cudaMemcpy(Ccpu, dC,  n * sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(dA); cudaFree(dB); cudaFree(dC);
}
```

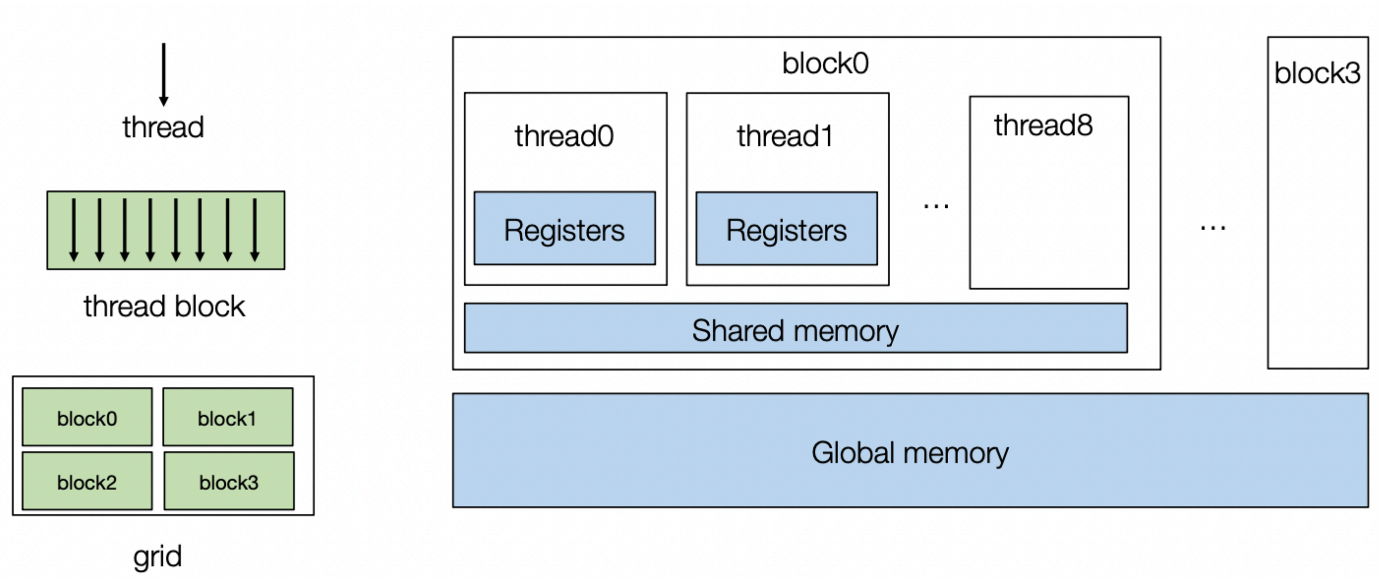
主要瓶颈在 CPU 与 GPU 之间的内存拷贝 (PCI-e)，因此实际应用会尽可能让数据放在 GPU 中。

> 使用 numpy 会将数据传回 CPU, 因此在 PyTorch 库中通常不使用 numpy.

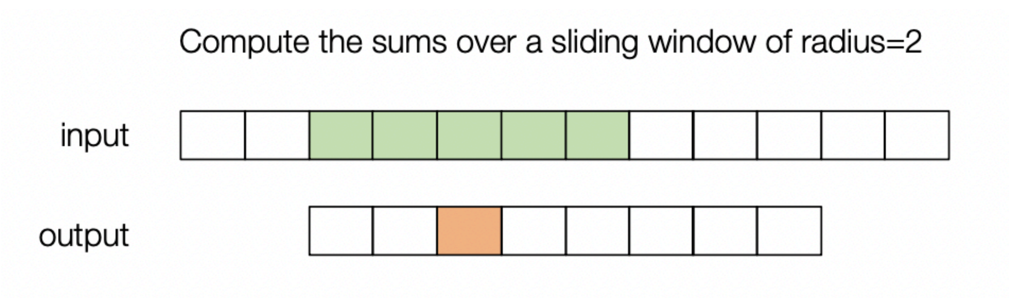
GPU Memory Hierarchy

Thread block 内的每个线程之间有共享内存，每个线程内部有自己的寄存器

Grid 中所有 thread block 之间有共享的全局内存



Example: Window Sum



```
#define RADIUS 2
__global__ void WindowSumSimpleKernel(float* A, float *B, int n) {
  int out_idx = blockDim.x * blockIdx.x + threadIdx.x;
  if (out_idx < n) {
    float sum = 0;
    for (int dx = -RADIUS; dx <= RADIUS; ++dx) {
      sum += A[dx + out_idx + RADIUS];
    }
    B[out_idx] = sum;
}
```

这样做的效率不高，因为相邻输出之间所用到的输入数据有重复，数据加载次数多

Takeaway: 同一个 block 内的线程之间协同将共用数据取到共享内存中，以提高复用

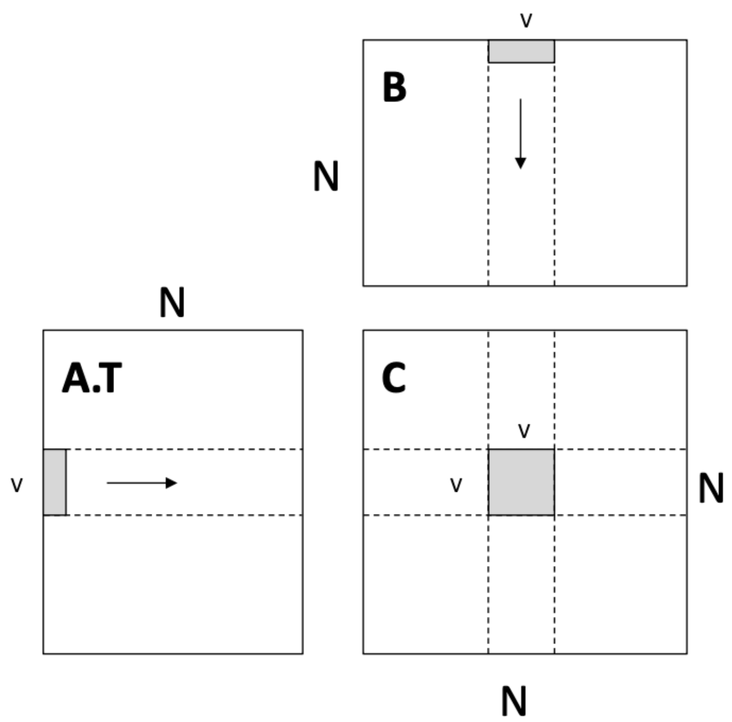
改进：大小为 4 的 thread block 协同将数据取到共享内存中，每个线程加载 2 次数据

```
#define RADIUS 2
__global__ void WindowSumSharedKernel(float* A, float *B, int n) {
  __shared__ float temp[THREADS_PER_BLOCK + 2 * RADIUS];
  int base = blockDim.x * blockIdx.x;
  int out_idx = base + threadIdx.x;
  if (base + threadIdx.x < n) {
    temp[threadIdx.x] = A[base + threadIdx.x];
  }
  if (threadIdx.x < 2 * RADIUS && base + THREADS_PER_BLOCK + threadIdx.x < n) {
    temp[threadIdx.x + THREADS_PER_BLOCK] = A[base + THREADS_PER_BLOCK + threadIdx.x];
  }
  __syncthreads();
  if (out_idx < n) {
      float sum = 0;
      for (int dx = -RADIUS; dx <= RADIUS; ++dx) {
      sum += temp[threadIdx.x + dx + RADIUS];
    }
    B[out_idx] = sum;
}
```

Case study: Matrix Multiplication on GPU

Thread-level: Register Tiling

计算 `C = dot(A.T, B)`

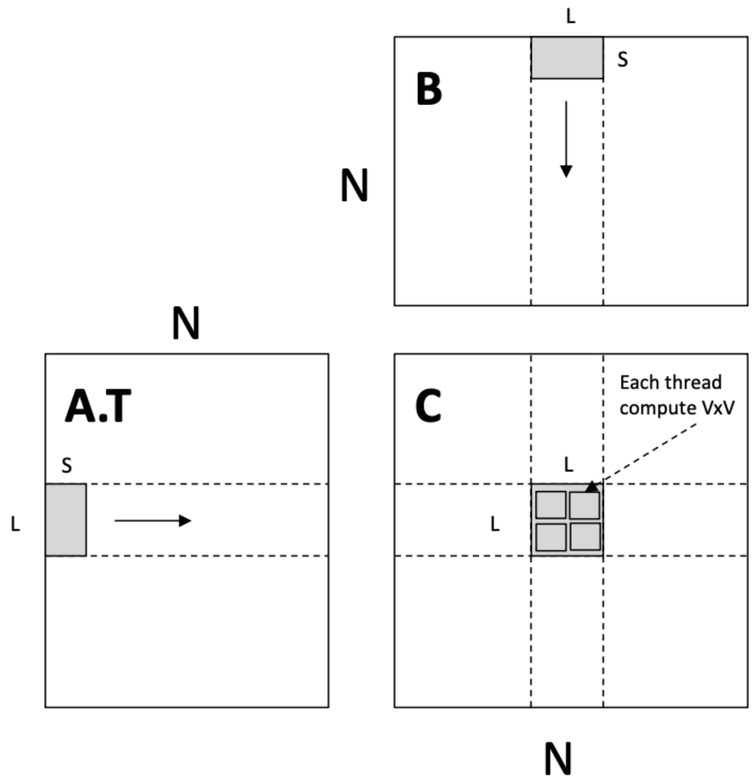



```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  int ybase = blockIdx.y * blockDim.y + threadIdx.y;
  int xbase = blockIdx.x * blockDim.x + threadIdx.x;

  float c[V][V] = {0};
  float a[V], b[V];
  for (int k = 0; k < N; ++k) {
    a[:] = A[k, ybase*V : ybase*V + V];
    b[:] = B[k, xbase*V : xbase*V + V];
    for (int y = 0; y < V; ++y) {
      for (int x = 0; x < V; ++x) {
        c[y][x] += a[y] * b[x];
      }
    }
  }
  C[ybase * V : ybase*V + V, xbase*V : xbase*V + V] = c[:];
}
```

Block-level: Shared Memory Tiling

每个 thread block 计算一个 L * L 的矩阵，每个线程计算一个 V * V 的矩阵




```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  __shared__ float sA[S][L], sB[S][L];
  float c[V][V] = {0};
  float a[V], b[V];
  int yblock = blockIdx.y;
  int xblock = blockIdx.x;

  for (int ko = 0; ko < N; ko += S) {
    __syncthreads();
    // needs to be implemented by thread cooperative fetching
    sA[:, :] = A[ko : ko + S, yblock * L : yblock * L + L];
    sB[:, :] = B[ko : ko + S, xblock * L : xblock * L + L];
    __syncthreads();
    for (int ki = 0; ki < S; ++ ki) {
      a[:] = sA[ki, threadIdx.y * V : threadIdx.y * V + V];
      b[:] = sB[ki, threadIdx.x * V : threadIdx.x * V + V];
      for (int y = 0; y < V; ++y) {
        for (int x = 0; x < V; ++x) {
          c[y][x] += a[y] * b[x];
        }
      }
    }
  }
  int ybase = blockIdx.y * blockDim.y + threadIdx.y;
  int xbase = blockIdx.x * blockDim.x + threadIdx.x;
  C[ybase * V : ybase * V + V, xbase * V : xbase * V + V] = c[:];
}
```

协同获取部分：

```
sA[:, :] = A[k : k + S, yblock * L : yblock * L + L];
```

```
int nthreads = blockDim.y * blockDim.x;
int tid = threadIdx.y * blockDim.x + threadIdx.x;

for(int j = 0; j < L * S / nthreads; ++j) {
  int y = (j * nthreads + tid) / L;
  int x = (j * nthreads + tid) % L;
  s[y, x] = A[k + y, yblock * L + x];
}
```

```
global->shared copy: 2 * N^3 / L
shared->register:    2 * N^3 / V
```

由于从全局内存传输到共享内存的速度比较慢，GPU 中其他空闲的线程可以在该线程等待传输的同时处理计算部分（上下文切换），此时需要有足够多的线程。

由于一个 GPU 核内的寄存器总数是固定值，所以 L 和 V 的选择存在线程数量与寄存器数量的 tradeoff: 要么更多线程但每个线程只有少量寄存器，或者少量线程但每个线程有很多寄存器。

在共享内存方面也存在 tradeoff.

More GPU Optimization Techniques

- Global memory continuous read
- Shared memory bank conflict
- Software pipelining
- Warp level optimizations
- Tensor Core