Introduction to Parallel and Distributed Processing

CUDA Matrix Product

Jaroslaw ‘Jaric’ Zola
http://www.jzola.org/

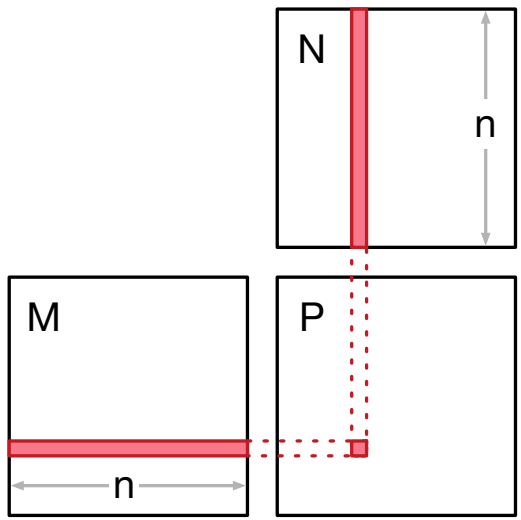
Back To Basics

- Two matrices M and N, we want to compute $P = M \times N$
- Reminder:

```
1  for (int i = 0; i < n; ++i) {
2    for (int j = 0; j < n; ++j) {
3      float S = 0.0;
4      for (int k = 0; k < n; ++k) {
5        S += M[i * n + k] * N[k * n + j];
6      }
7      P[i * n + j] = S;
8    }
9  }
```

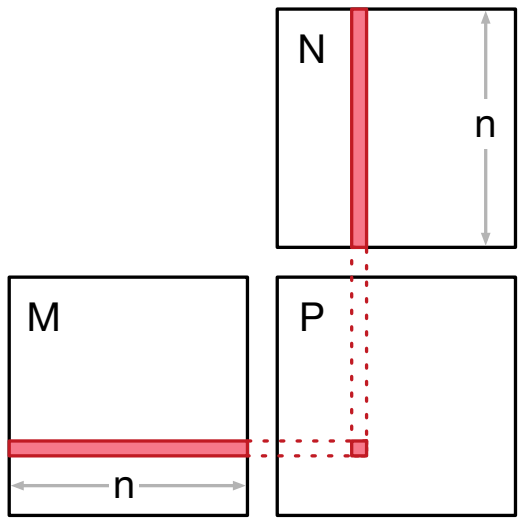
Obvious Approach

- One thread computes one element of P:



Obvious Approach

- One thread computes one element of P:

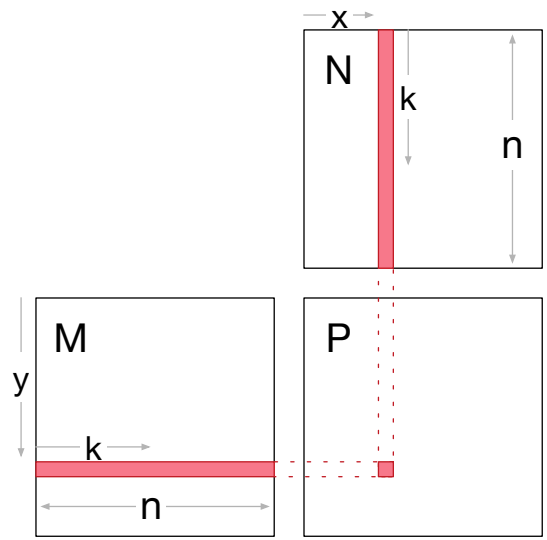


- M and N will be loaded too many times...
- Limited size of the problem we can run...

Obvious Approach

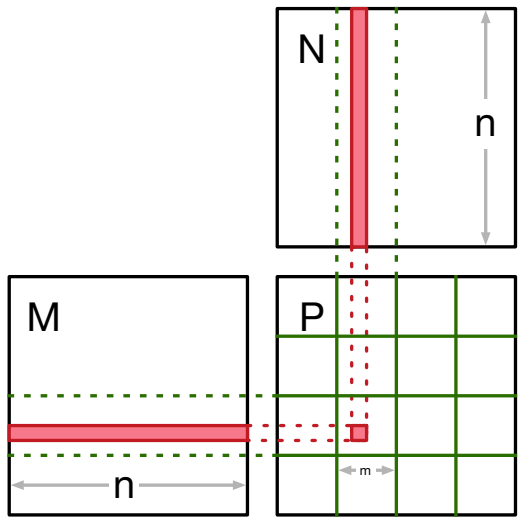
```
1  __global__ void mmul(int n, const float* M, const float* N,
2                       float* P) {
3    float S = 0.0;
4    for (int k = 0; k < n; ++k) {
5      float Mk = M[threadIdx.y * n + k];
6      float Nk = N[n * k + threadIdx.x];
7      S += Mk * Nk;
8    }
9    P[threadIdx.y * n + threadIdx.x] = S;
10 }
```

```
1  dim3 dimGrid(1, 1);
2  dim3 dimBlock(n, n);
3  mmul<<<dimGrid, dimBlock>>>(n, M, N, P);
```



Better Approach

- Use grid with more than one block
- One block of m^2 threads works on a matrix tile of size $m \times m$



Better Approach

```
1  __global__ void mmul(int n, int m, const float* M, const float* N,
2                       float* P) {
3    int row = blockIdx.y * m + threadIdx.y;
4    int col = blockIdx.x * m + threadIdx.x;
5    float S = 0.0;
6    for (int k = 0; k < n; ++k) {
7      float Mk = M[row * n + k];
8      float Nk = N[n * k + col];
9      S += Mk * Nk;
10   }
11   P[row * n + col] = S;
12 }
```

Can We Do Better?

- Each input element is read by n threads
- Let's use shared memory...

Can We Do Better?

- Each input element is read by n threads
- Let's use shared memory...
- We can slide tile-by-tile, and use shared memory to "cache" M and N for reuse

Using Shared Memory

```
1  __global__ void mmul(int n, int m, const float* M, const float* N,
2                       float* P) {
3    extern __shared__ float buf[];
4    float* Ms = buf;
5    float* Ns = buf + m * m;
6
7    int tx = threadIdx.x;
8    int ty = threadIdx.y;
9    int row = blockIdx.y * m + ty;
10   int col = blockIdx.x * m + tx;
11
12   float S = 0.0;
13
14   for (int i = 0; i < n / m; ++i) {
15     Ms[ty * m + tx] = M[row * n + (i * m) + tx];
16     Ns[ty * m + tx] = N[(i * m + ty) * n + col];
17     __syncthreads();
18
19     for (int k = 0; k < m; ++k) {
20       S += Ms[ty * m + k] * Ns[k * m + tx];
21     }
22     __syncthreads();
23   }
24   P[row * n + col] = S;
25 }
```

Using Shared Memory

```
1  __global__ void mmul(int n, int m, const float* M, const float* N,
2                       float* P) {
3    extern __shared__ float buf[];
4    float* Ms = buf;
5    float* Ns = buf + m * m;
6
7    int tx = threadIdx.x;
8    int ty = threadIdx.y;
9    int row = blockIdx.y * m + ty;
10   int col = blockIdx.x * m + tx;
11
12   float S = 0.0;
13
14   for (int i = 0; i < n / m; ++i) {
15     Ms[ty * m + tx] = M[row * n + (i * m) + tx];
16     Ns[ty * m + tx] = N[(i * m + ty) * n + col];
17     __syncthreads();
18
19     for (int k = 0; k < m; ++k) {
20       S += Ms[ty * m + k] * Ns[k * m + tx];
21     }
22     __syncthreads();
23   }
24   P[row * n + col] = S;
25 }
```

```
1  dim3 dimGrid(m / n, m / n);
2  dim3 dimBlock(m, m);
3  mmul<<<dimGrid, dimBlock, 2 * m * m * sizeof(m)>>>(n, m, M, N, P);
```

For Fun

- Implement and profile the codes provided in this lecture.