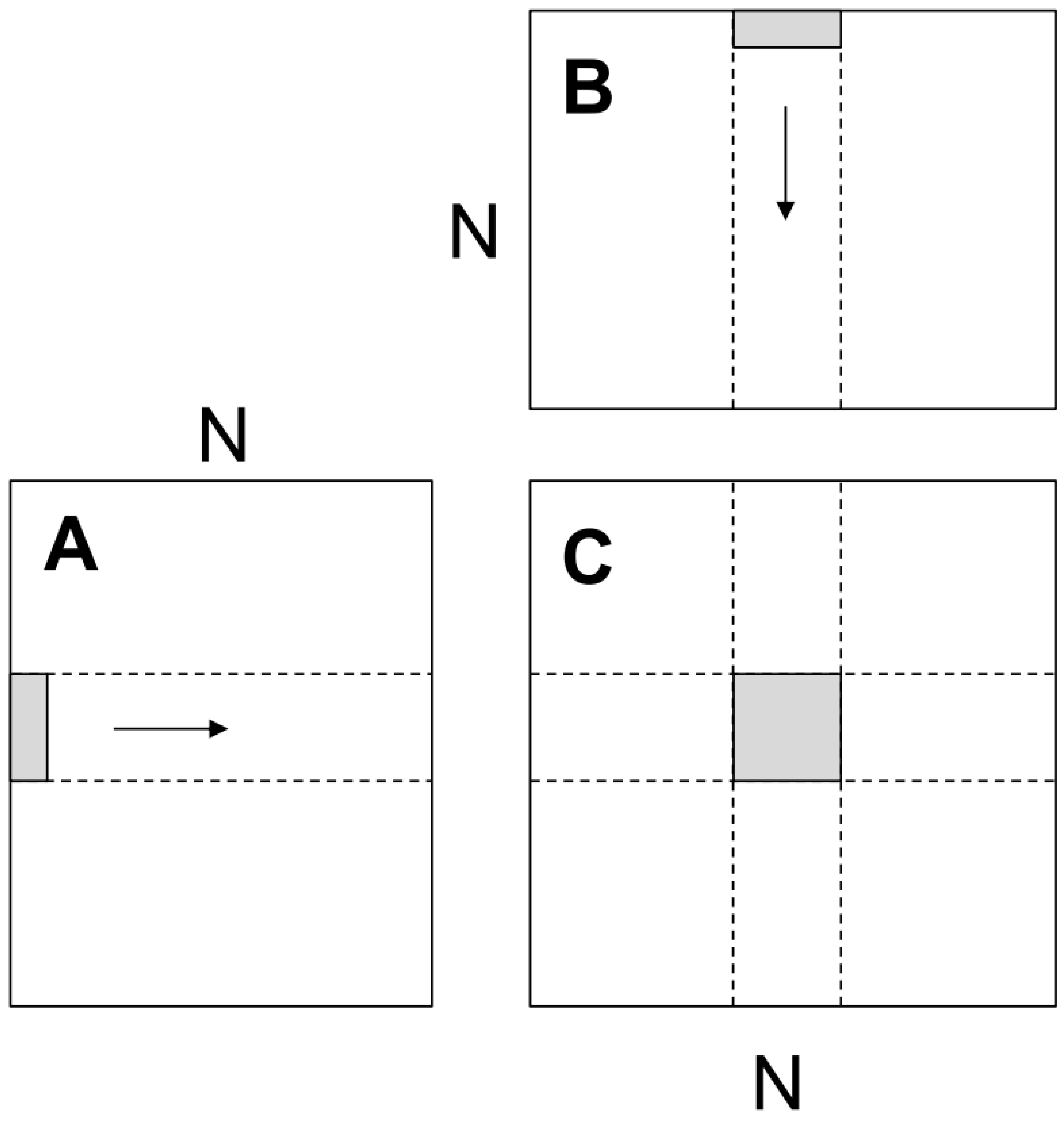
Case study: GPU Matmul v1

- C = A x B
- Q: what's the work that can be parallelizec
- 💡Each thread computes one element!

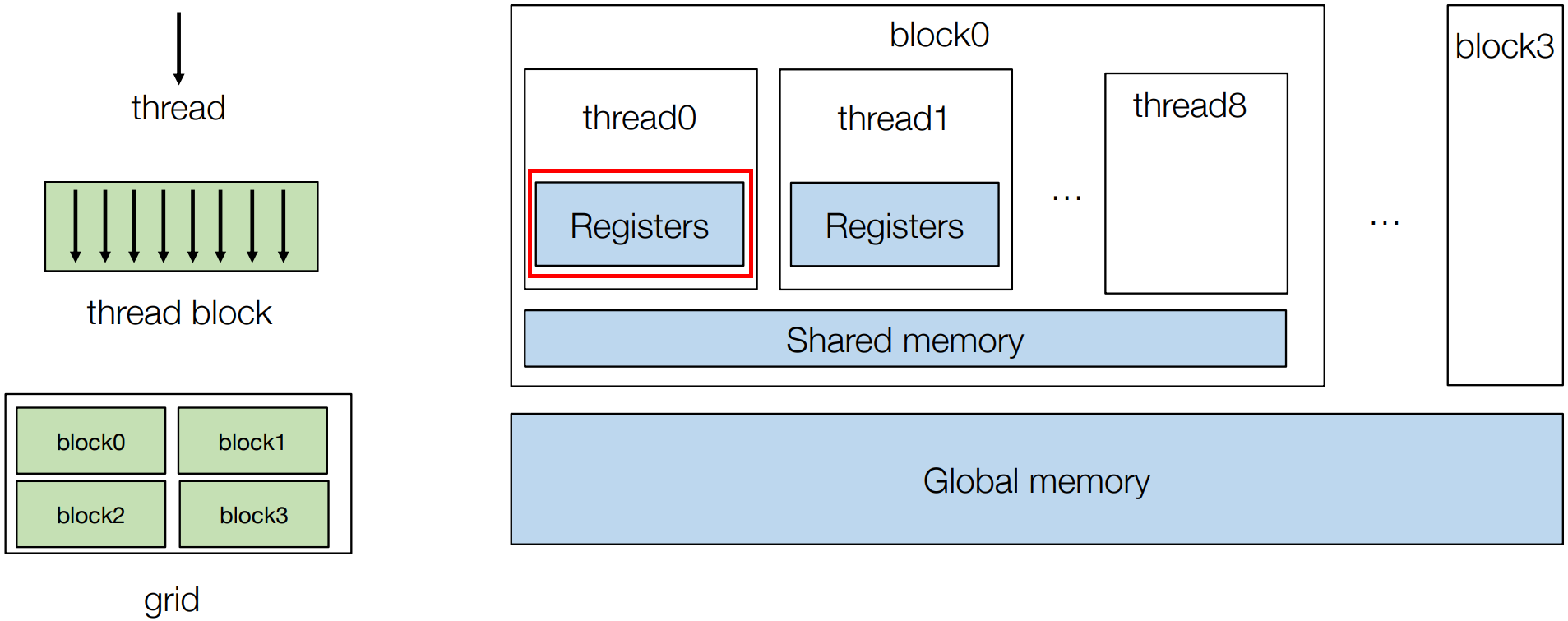



```
int N = 1024;
dim3 threadsPerBlock(32, 32, 1);
dim3 numBlocks(N/32, N/32, 1);

matmul<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

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

  result = 0;
  for (int k = 0; k < N; ++k) {
    result += A[x][k] * B[k][y];
  }
  C[x][y] = result;
}
```

- Global memory read per thread?
  - N + N = 2N
- # threads?
  - N^2
- Total global memory access?
  - N^2 * 2N = 2N^3
- Memory?
  - 1 float per thread

Recall Memory Hierarchy and Register tiling

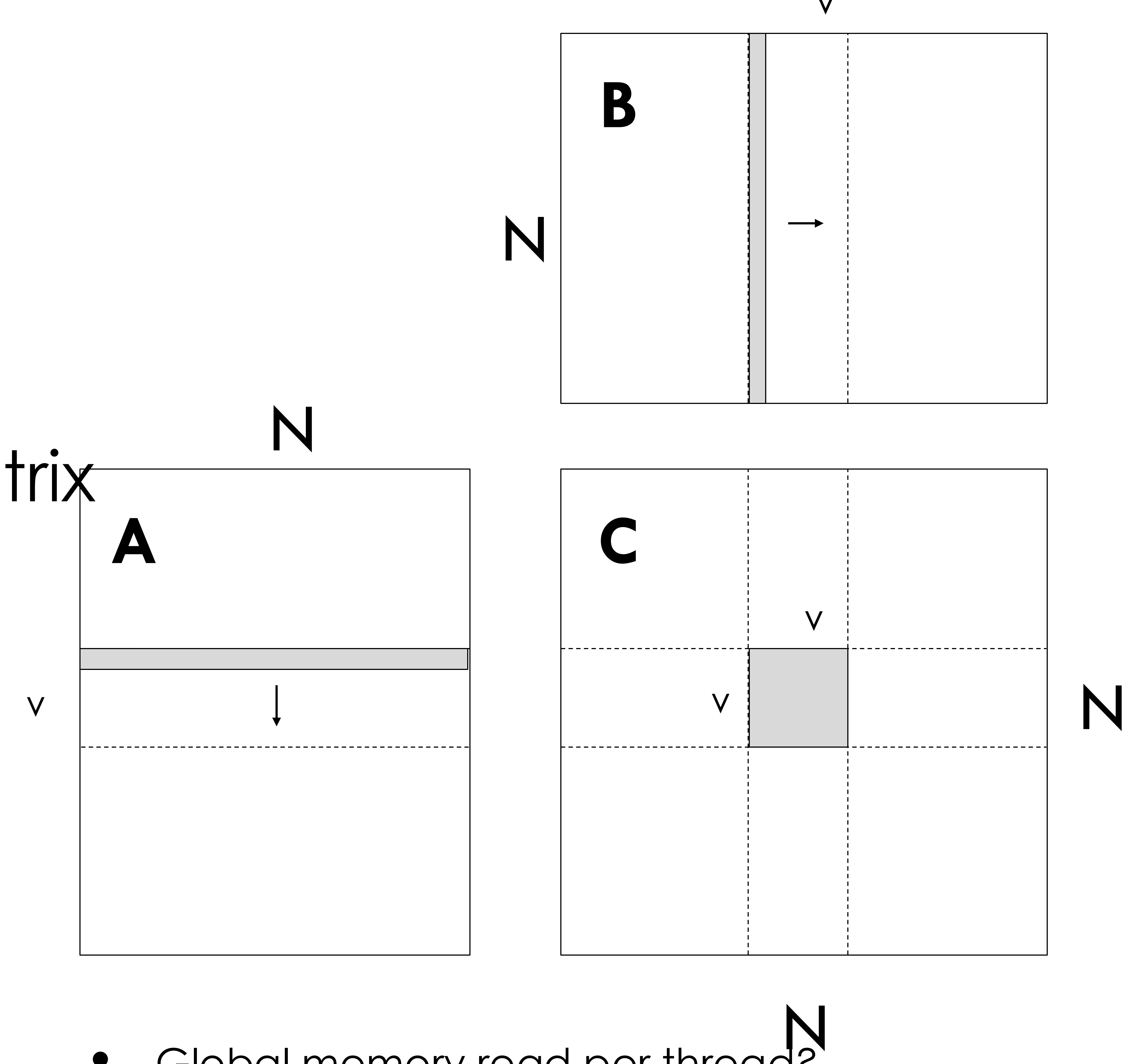


💡Each thread uses more thread-level registers to compute outputs to save I/o

GPU Matmul v1.5: Thread Tiling

- Each thread computes a VxV submatrix

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

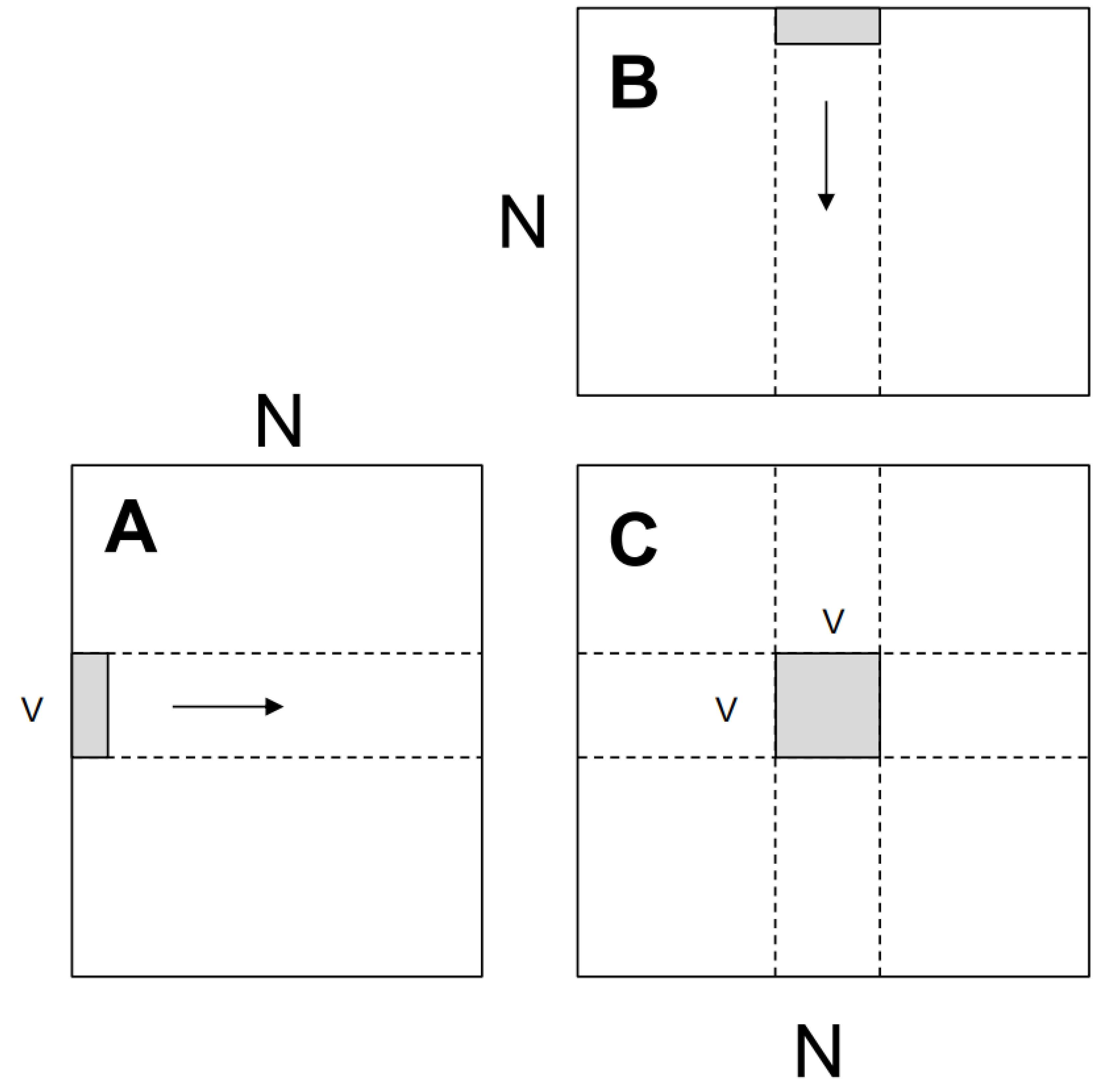


- Global memory read per thread?
  - NV + NV^2
- # threads?
  - N/V * N/V = N^2/V^2
- Total global memory access?
  - N^2 / V^2 * (NV + NV^2) = N^3/V + N^3
- Memory?
  - V^2 + 2N float per thread

GPU Matmul v2: Can we do better?

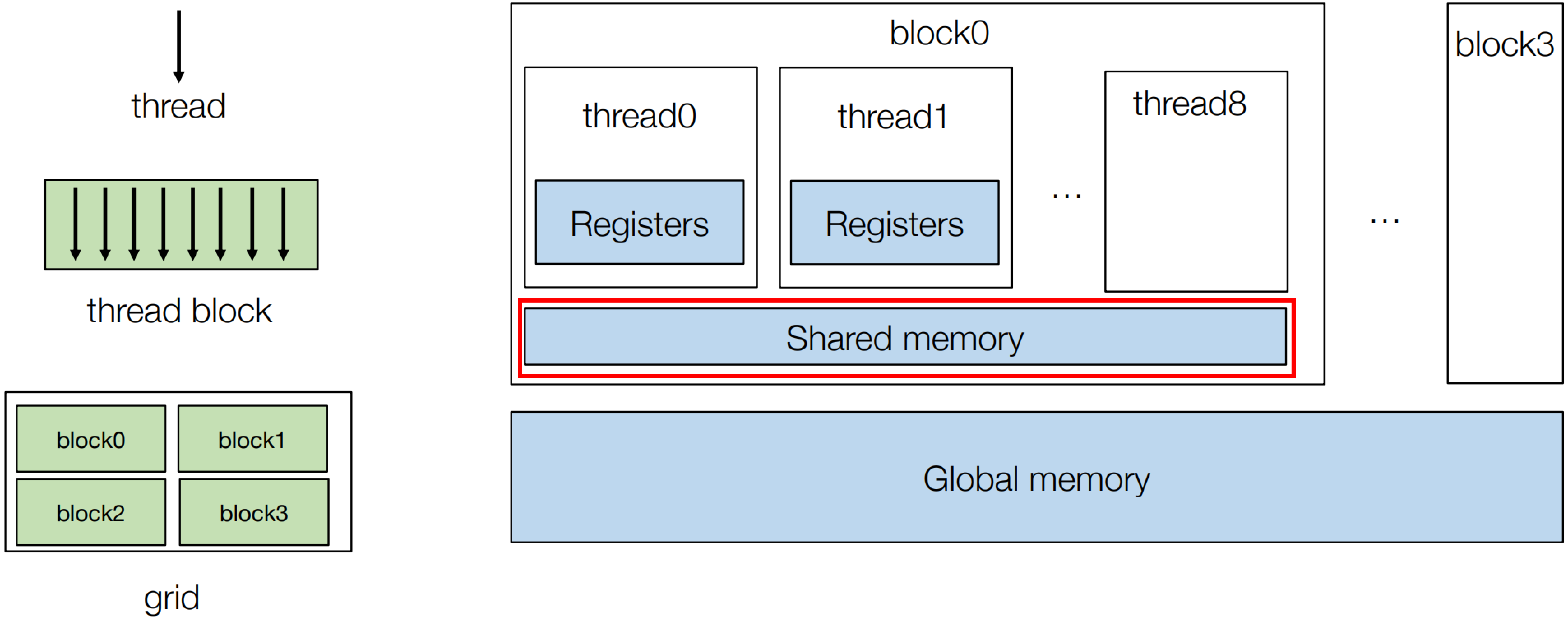
- Each thread computes a VxV submatrix
- 💡compute partial sum: $[X_1, X_2]\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = X_1Y_1 + X_2Y_2$

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



- Global memory read per thread?
  - NV * 2
- # threads?
  - N/V * N/V = N^2/V^2
- Total global memory access?
  - N^2 / V^2 * 2NV = 2N^3/V
- Memory?
  - V^2 + 2V float per thread

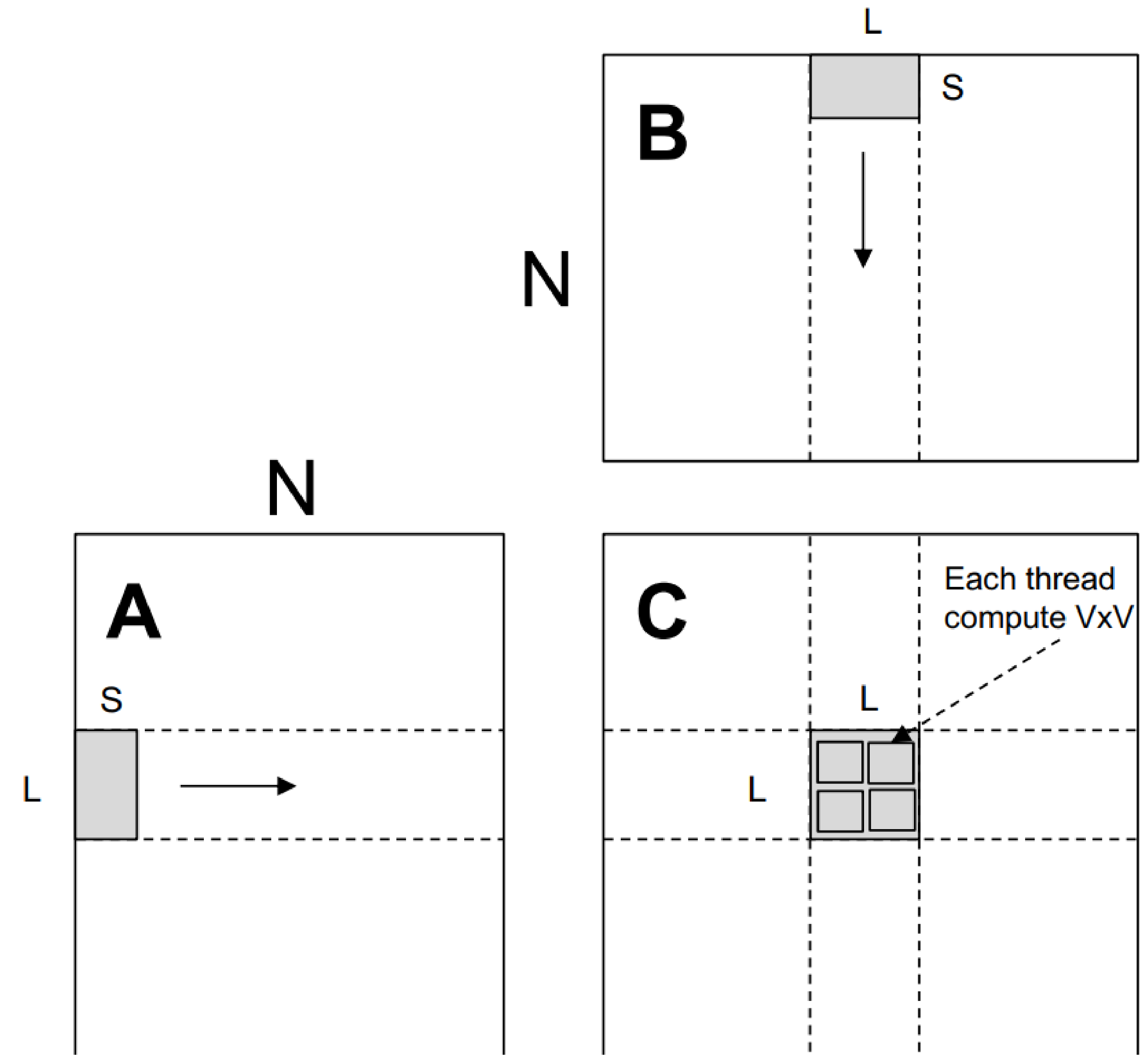
Recall Memory Hierarchy and Cache tiling



💡Try to utilize block-level shared memory (SRAM)

GPU Matmul v3: SRAM Tiling (GPU)

- Use block shared mem
- A block computes a L x L submatrix
- Then a thread computes a V x V submatrix and reuses the matrices in shared block memory




```
__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[:] = sA[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[:];
}
```

Memory overhead?

- Global memory access per threadblock
  - 2LN
- Number of threadblocks:
  - N^2 / L^2
- Total global memory access:
  - 2N^3 / L
- Shared memory access per thread:
  - 2VN
- Number of threads
  - N^2 / V^2
- Total shared memory access:
  - 2N^3 / V

```
__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[:] = sA[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[:];
}
```

Cooperative Fetching

```
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];
}
```