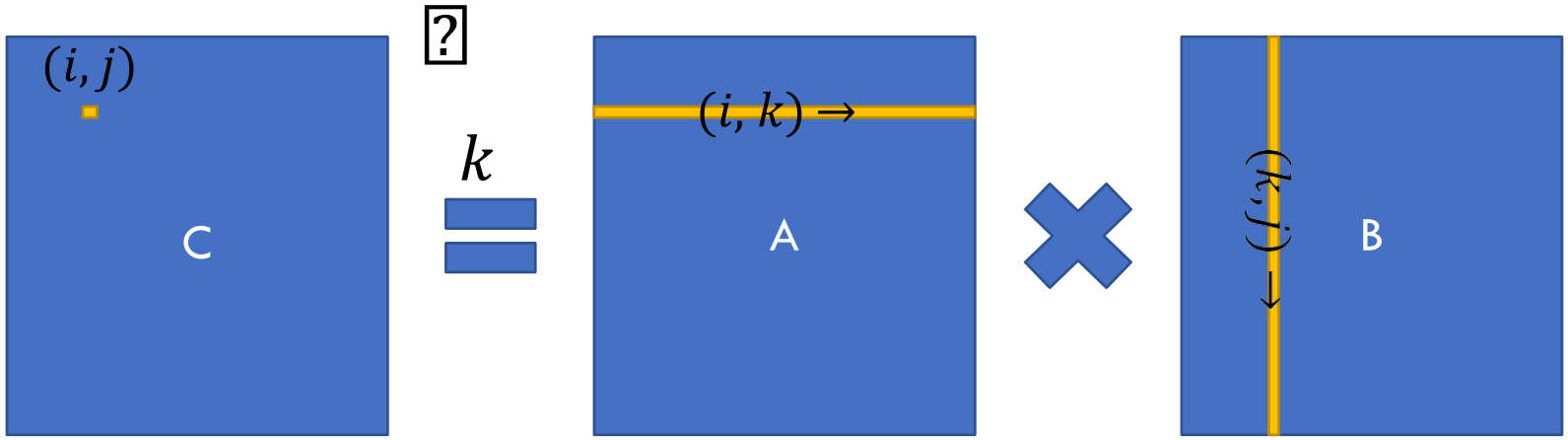
Matrix multiplication



Matrix multiplication (matmul)

- Simple C++ implementation:

```
/* Find element based on row-major ordering */
#define RM(r, c, width) ((r) * (width) + (c))

// Standard multiplication
void multMatrixSimple(int N, float *matA, float *matB, float *matC) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            float sum = 0.0;
            for (int k = 0; k < N; k++)
                sum += matA[RM(i,k,N)] * matB[RM(k,j,N)];
            matC[RM(i,j,N)] = sum;
        }
}
```

Translating matmul to CUDA

- SPMD (single program, multiple data) parallelism
  - "Map this function to all of this data": $\mathrm{map}(f, data)$
  - Similar to SIMD, but doesn't require lockstep execution

- What this means: You write the "inner loop", compiler + GPU execute it in parallel

Translating matmul to CUDA

- Simple CUDA implementation:

```
/* Find element based on row-major ordering */
#define RM(r, c, width) ((r) * (width) + (c))

// Standard multiplication
void multMatrixSimple(int N, float *matA, float *matB, float *matC) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            float sum = 0.0;
            for (int k = 0; k < N; k++)
                sum += matA[RM(i,k,N)] * matB[RM(k,j,N)];
            matC[RM(i,j,N)] = sum;
        }
}
```

1. Find the inner loop

Translating matmul to CUDA

- Simple CUDA implementation:

```
__global__void
cudaSimpleOldKernel(int N, float*  dmatA, float* dmatB, float * dmatC) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i >= N || j >= N)
        return;
    float sum = 0.0;
    for (int k = 0; k < N; k++) {
        sum += dmatA[RM(i,k,N)] * dmatB[RM(k,j,N)];
    }
    dmatC[RM(i,j,N)] = sum;
}
```

2. Write it as a separate function

Translating matmul to CUDA

- Simple CUDA implementation:

```
__global__void
cudaSimpleOldKernel(int N, float*  dmatA, float* dmatB, float * dmatC) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i >= N || j >= N)
        return;
    float sum = 0.0;
    for (int k = 0; k < N; k++) {
        sum += dmatA[RM(i,k,N)] * dmatB[RM(k,j,N)];
    }
    dmatC[RM(i,j,N)] = sum;
}
```

3. Compute loop index + test bound (no outer loop)

Invoking CUDA matmul

- Setup memory (from CPU to GPU)
- Invoke CUDA with special syntax
- Get results (from GPU to CPU)

Invoking CUDA matmul

- Setup memory (from CPU to GPU)

These addresses are only valid on GPU

```
cudaMalloc((void **) &aDevData, N*N * sizeof(float));
cudaMalloc((void **) &bDevData, N*N * sizeof(float));
cudaMalloc((void **) &cDevData, N*N * sizeof(float));
cudaMemcpy(aDevData, aData, N*N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(bDevData, bData, N*N * sizeof(float), cudaMemcpyHostToDevice);
```

Need to move data manually (separate address spaces)

- Invoke CUDA with special syntax
- Get results (from GPU to CPU)

Invoking CUDA matmul

- Setup memory (from CPU to GPU)
- Invoke CUDA with special syntax

```
#define N 1024
#define LBLK 32
dim3 threadsPerBlock(LBLK, LBLK);
dim3 blocks(updiv(N, LBLK), updiv(N, LBLK)); // updiv() divides + rounds up
cudaSimpleKernelOld<<<blocks, threadsPerBlock>>>(N, aDevData, bDevData, cDevData);
```

These addresses are only valid on GPU

- Get results (from GPU to CPU)

Invoking CUDA matmul

- Setup memory (from CPU to GPU)
- Invoke CUDA with special syntax
- Get results (from GPU to CPU)

Need to move data manually (separate address spaces)

```
tHostData = (float *) calloc(N*N, sizeof(float));
cudaMemcpy(tHostData, tDevData, N*N*sizeof(float), cudaMemcpyDeviceToHost);
cudaFree(aDevData); cudaFree(bDevData); cudaFree(cDevData);
```

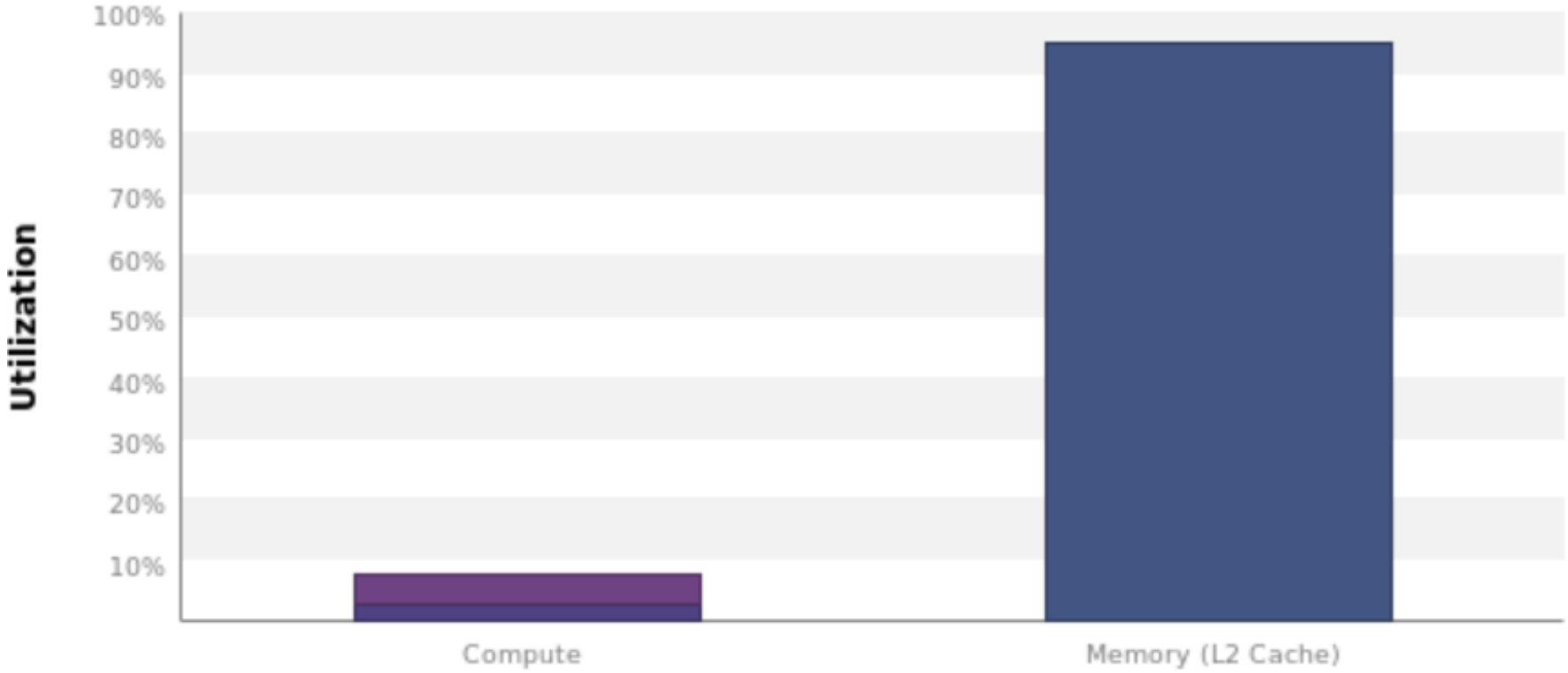
Compiling + running CUDA

- CUDA code is in separate *.cu file (cudaMatrix.cu)
  - Compiled like:
    `nvcc cudaMatrix.cu -O3 -c -o cudaMatrix.o`
  - (See assignment 6 for $PATH, etc)

- Linked with gcc + flags, e.g.:
  - `g++ -O3 -L/path/to/cuda -lcudart -o matrix *.o`

- Run like a normal program, e.g.:
  - `./matrix`

Profiling Results



matmul is memory bound!

Improving matmul memory usage

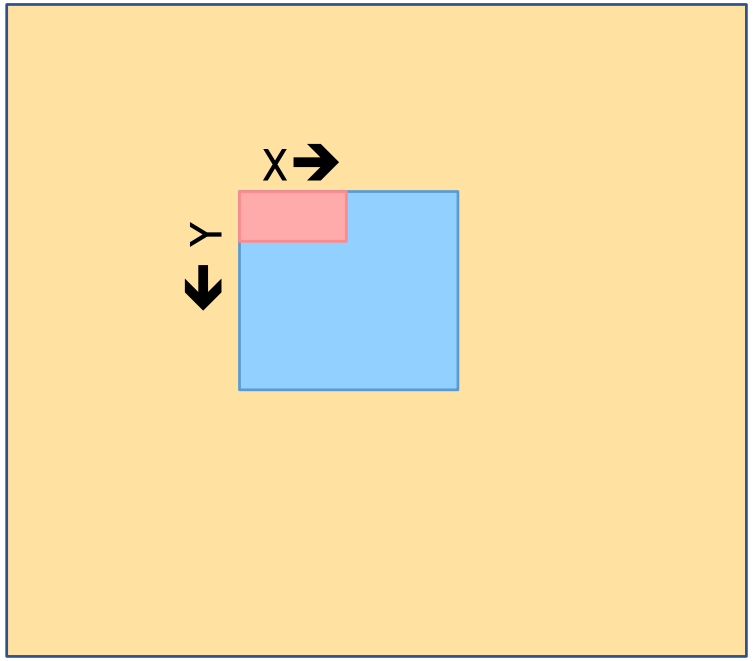
- Why is matmul accessing memory so much?

```
__global__void
cudaSimpleOldKernel(int N, float* dmatA,
                    float* dmatB, float * dmatC) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i >= N || j >= N)
        return;
    float sum = 0.0;
    for (int k = 0; k < N; k++) {
        sum += dmatA[RM(i,k,N)] * dmatB[RM(k,j,N)];
    }
    dmatC[RM(i,j,N)] = sum;
}
```

Improving matmul memory usage: Peeking under the hood

- Need to think about how threads *within a warp* access memory…
  - (This is bad – warps aren't part of programming model)

- CUDA maps threads -> warps *row-major:* Same y values, consecutive x values
  - Warp 0:
    (0,0) (1,0) (2,0) … (31,0)



Improving matmul memory usage: Warp memory access pattern

- What memory locations does warp 0 access?

```
int i = blockIdx.x * blockDim.x + threadIdx.x;  int j =
blockIdx.y * blockDim.y + threadIdx.y;
```

- Access: `dmatA[RM(i,k,N)]`, `dmatB[RM(k,j,N)]`, `dmatC[RM(i,j,N)]` where `RM(i,j,N) = i*N + j`

- Threads have same y + consecutive x ➔
- Threads accesses the same col + consecutive i ➔
- Threads access memory at stride of N floats ➔
- 1 reads + 1 writes <u>per thread</u>

Improving matmul memory usage: Better spatial locality

- What if we flipped it around?

```
int i = blockIdx.y * blockDim.y + threadIdx.y;
int j = blockIdx.x * blockDim.x + threadIdx.x;
```

- Threads have same y + consecutive x ➔
- Threads access the same i + consecutive j ➔
- Threads access memory at stride of 1 ➔
- GPU *coalesces* reads + writes to memory block ➔
- 1 read + 1 write per warp (if large memory blocks)

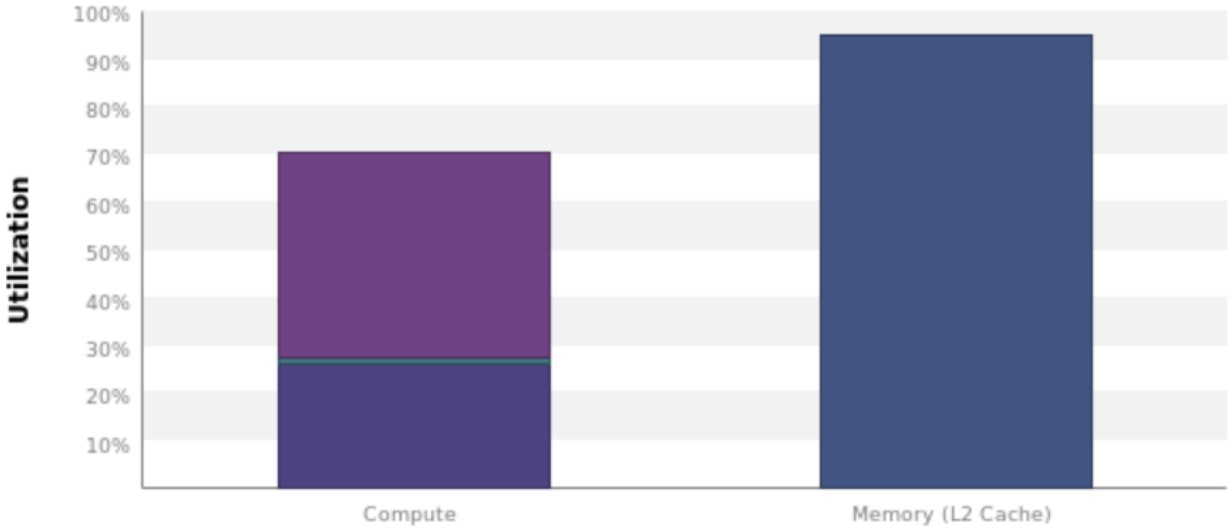
Benchmarking improved simple CUDA matmul

- ./matrix -n 1024 -N 1024 -m csimple

- Simple C++: 1300 ms, 1.6 Gflops
- Simple CUDA: 33 ms, 65 Gflops
- Simple++ CUDA: 2.4 ms, 900 Gflops

Profiling improved simple CUDA matmul ***

- `nvprof --analysis-metrics -f -o csimple.nvprof ./matrix -n 1024 -N 1024 -m csimple`
- `nvvp csimple.nvprof`
- Doing better!



- …Still memory bound, though

*** - Using deprecated profiling tools

CUDA disassembly + its limits

- You can look at PTX assembly:
  `cuobjdump --dump-ptx matrix`

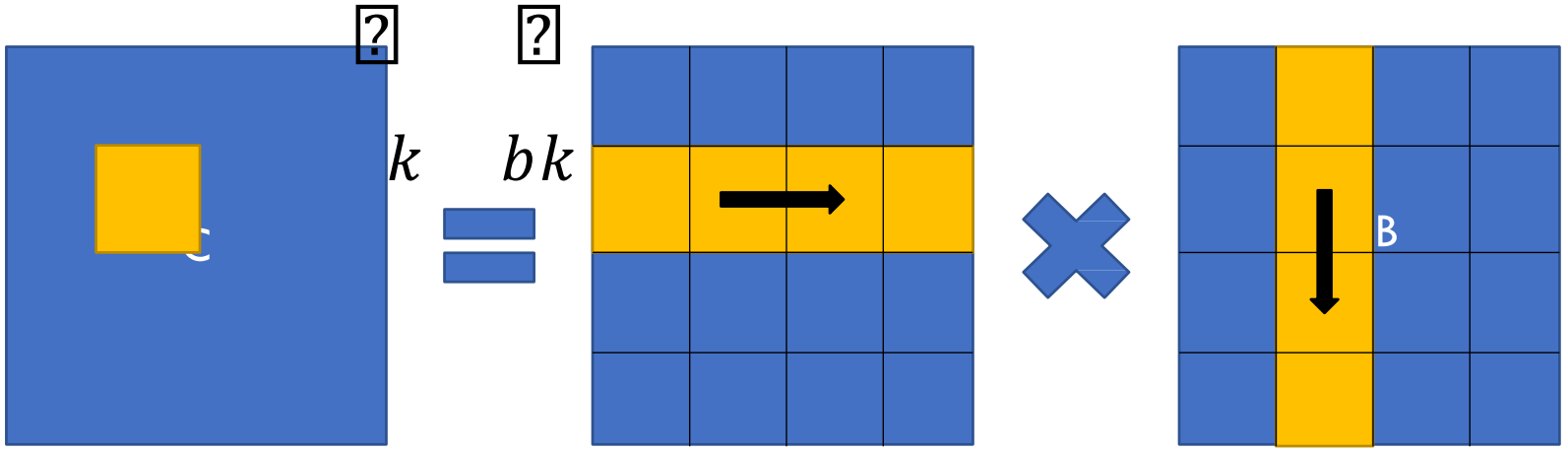
- …But you will not see difference in this case
  (Coalescing done by hardware, not compiler)

```
.visible .entry _Z19cudaSimpleKernelOldiPfS_S_(
...
ld.global.f32 %f6, [%rd9];
ld.global.f32 %f7, [%rd7];
...
st.global.f32 [%rd12], %f9;
...
```

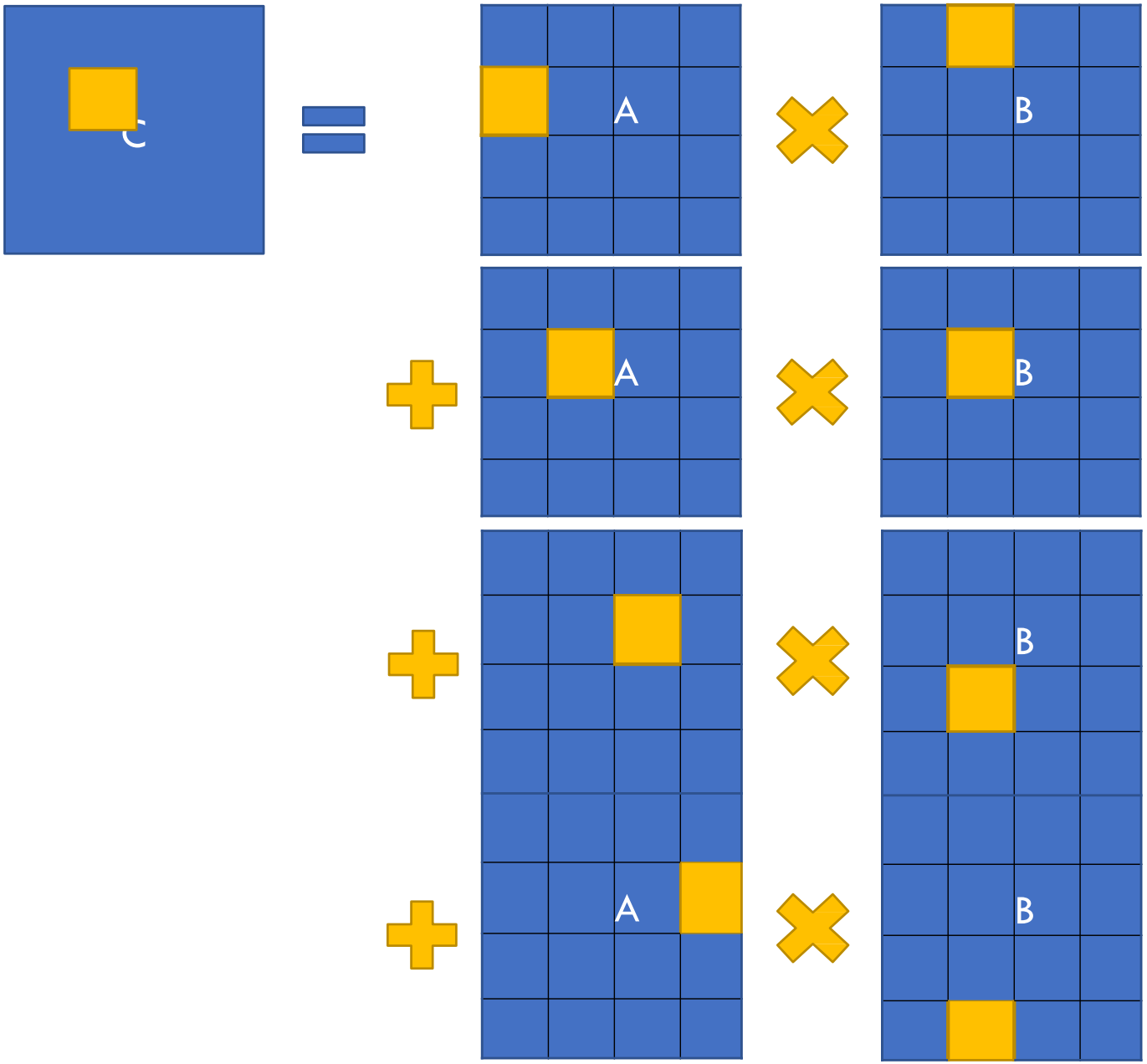
```
.visible .entry _Z19cudaSimpleKerneliPfS_S_(
...
ld.global.f32 %f6, [%rd9];
ld.global.f32 %f7, [%rd7];
...
st.global.f32 [%rd12], %f9;
...
```

Blocked matmul: Even better memory usage

- Problem: Entire matrix doesn't fit in local cache



- Classic solution: *Block* into sub-matrices that do fit in cache, and then multiply and sum sub-matrices
  - (This is just a re-association of the original computation)


C
A
B
A
B
A
B
A
B

Blocked matmul: C++ version

```
void multMatrixBlocked(int N, float *matA, float *matB, float *matC) {
    /* Zero out C */
    memset(matC, 0, N * N * sizeof(float));
    int i, j, k;
    for (i = 0; i <= N-SBLK; i+= SBLK) {
        for (j = 0; j <= N-SBLK; j+= SBLK) {
            for (k = 0; k <= N-SBLK; k+= SBLK) {
                for (int bi = 0; bi < SBLK; bi++)  {
                    for (int bj = 0; bj < SBLK; bj++) {
                        float sum = 0.0;
                        for (int bk =0; bk < SBLK; bk++)
                            sum += matA[RM(i+bi,k+bk,N)]
                                * matB[RM(k+bk,j+bj,N)];
                        matC[RM(i+bi,j+bj,N)] += sum;
}   }   }   }   }   }
```

Outer loops iterate over submatrices in steps of SBLK

Inner bi, bj loops iterate over sub-matrix and accumulate into output matrix

Note: This code assumes SBLK evenly divides N; need extra loops for "leftovers" in general

Benchmarking blocked matmul in C++

- ./matrix -n 1024 -N 1024 -m block

- Simple C++: 1300 ms, 1.6 Gflops
- Simple CUDA: 33 ms, 65 Gflops
- Simple++ CUDA: 2.4 ms, 900 Gflops

- Block C++: 500 ms, 4.3 Gflops

Blocked matmul: CUDA version

1. Find the inner loop

2. Write it as a separate function

3. Compute indices from block/thread id

Blocked matmul: Attempt #1

```
__global__void
cudaBlockKernelCoarse(int N, float *dmatA, float *dmatB, float *dmatC) {
  int i = blockIdx.y * blockDim.y + threadIdx.y; i *= LBLK;
  int j = blockIdx.x * blockDim.x + threadIdx.x; j *= LBLK;

  for (int bi = 0; bi < LBLK; bi++)
    for (int bj = 0; bj < LBLK; bj++)
      dmatC[RM(i+bi,j+bi,N)] = 0;

  for (int k = 0; k <= N-LBLK; k+=LBLK) {
    for (int bi = 0; bi < LBLK; bi++) {
      for (int bj = 0; bj < LBLK; bj++) {
        float sum = 0.0;
        for (int bk = 0; bk < LBLK; bk++) {
          sum += dmatA[RM(i+bi,k+bk,N)]
            * dmatB[RM(k+bk,j+bj,N)];
        }
        dmatC[RM(i+bi,j+bj,N)] += sum;
} } } }
```

Map threads across submatrices

Compute submatrix product

Blocked matmul: Attempt #1 + Local memory

```
__global__void cudaBlockKernelCoarse(int N, float *dmatA, float *dmatB,
float *dmatC) {
  int i = blockIdx.y * blockDim.y + threadIdx.y; i *= LBLK;
  int j = blockIdx.x * blockDim.x + threadIdx.x; j *= LBLK;
  float subA[LBLK * LBLK];
  float subB[LBLK * LBLK];
  float subC[LBLK * LBLK];

  for (int bi = 0; bi < LBLK; bi++) /* Zero out C */
    for (int bj = 0; bj < LBLK; bj++)
      subC[RM(bi,bj,LBLK)] = 0;

  for (int k = 0; k <= N-LBLK; k+=LBLK) {
    for (int bi = 0; bi < LBLK; bi++) {
      for (int bj = 0; bj < LBLK; bj++) {
        subA[RM(bi,bj,LBLK)] = dmatA[RM(i+bi,k+bj,N)];
        subB[RM(bi,bj,LBLK)] = dmatB[RM(k+bi,j+bj,N)];
    } }
    for (int bi = 0; bi < LBLK; bi++) {
      for (int bj = 0; bj < LBLK; bj++) {
        float sum = 0.0;
        for (int bk = 0; bk < LBLK; bk++) {
          sum += subA[RM(bi,bk,LBLK)] * subB[RM(bk,bj,LBLK)];
        }
        subC[RM(bi,bj,LBLK)] += sum;
  } } }
  for (int bi = 0; bi < LBLK; bi++)
    for (int bj = 0; bj < LBLK; bj++)
      dmatC[RM(i+bi,j+bj,N)] = subC[RM(bi,bj,LBLK)];
}
```

Keep a local copy of submatrix

Explicitly read from global to local memory

Only reference local copy in loop

Explicitly write from local to global memory

Benchmarking blocked matmul

- ./matrix -n 1024 -N 1024 -m block

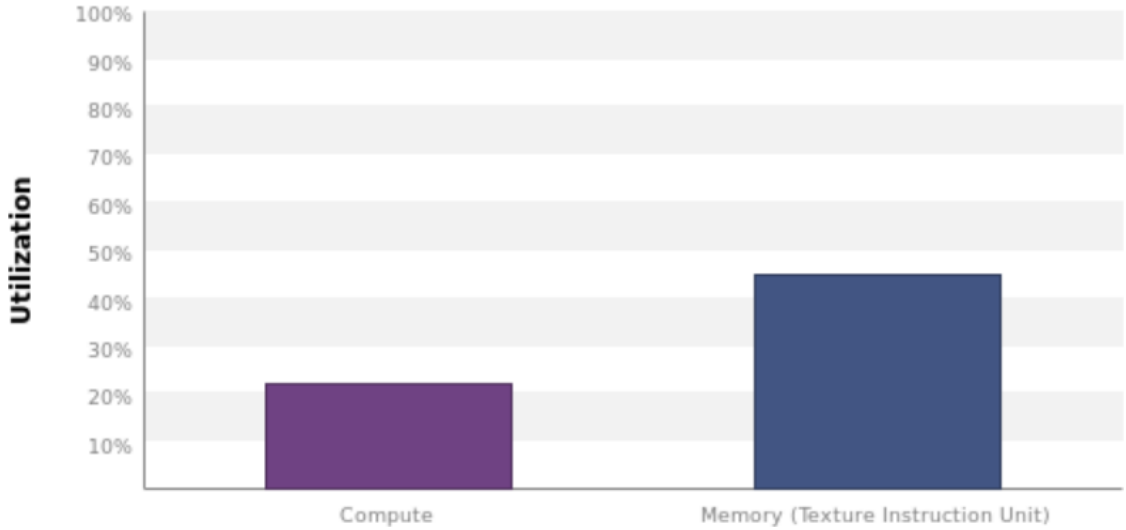
- Simple C++: 1300 ms, 1.6 Gflops
- Simple CUDA: 33 ms, 65 Gflops
- Simple++ CUDA: 2.4 ms, 900 Gflops

- Block C++: 500 ms, 4.4 Gflops
- Block CUDA: 107 ms, 20 Gflops ☹

Profiling blocked matmul ***

- `nvprof --analysis-metrics -f -o ccblock.nvprof ./matrix -n 1024 -N 1024 -m ccblock`
- `nvvp ccblock.nvprof`

- Huh…



*** - Using deprecated profiling tools

Blocked matmul: What went wrong?

- How much parallelism is there in our first attempt?

- Each thread generates 32×32 output elements
- Each thread block is 32×32 threads
- There are 1024×1024 output elements

- ➔ We only spawn one thread block!
- Need to split loops across more threads

Blocked matmul: Attempt #2

- Original matmul had one thread for each output element: 1024×1024 threads
  - 1 thread for each i, j loop iteration in C++ code

- Idea: Unroll the inner bi & bj loops in Attempt #1 across a threads in a block
- ➔ Thread block shares a single copy of submatrix

Blocked matmul: Attempt #2

```
__global__void cudaBlockKernel(int N, float *dmatA, float *dmatB, float *dmatC) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;

  int bi = threadIdx.y;
  int bj = threadIdx.x;

  __shared__float subA[LBLK * LBLK];
  __shared__float subB[LBLK * LBLK];
  float sum = 0;

  for (int k = 0; k < N; k += LBLK) {
    subA[RM(bi,bj,LBLK)] = dmatA[RM(i,k+bj,N)];
    subB[RM(bi,bj,LBLK)] = dmatB[RM(k+bi,j,N)];


    for (int bk = 0; bk < LBLK; bk++) {
      sum += subA[RM(bi,bk,LBLK)] * subB[RM(bk,bj,LBLK)];
    }


  }

  dmatC[RM(i,j,N)] = sum;
}
```

Each thread responsible for one output element (like original CUDA code)

But now mapped within a LBLK × LBLK block

Keep a block-shared copy of submatrix

Explicitly read from global to shared memory

Only reference shared copy in loop

Explicitly write from local to global memory

Is this code correct?

Blocked matmul: Attempt #2

```
__global__void cudaBlockKernel(int N, float *dmatA, float *dmatB, float *dmatC) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;

  int bi = threadIdx.y;
  int bj = threadIdx.x;

  __shared__float subA[LBLK * LBLK];
  __shared__float subB[LBLK * LBLK];
  float sum = 0;

  for (int k = 0; k < N; k += LBLK) {
    subA[RM(bi,bj,LBLK)] = dmatA[RM(i,k+bj,N)];
    subB[RM(bi,bj,LBLK)] = dmatB[RM(k+bi,j,N)];

    __syncthreads();

    for (int bk = 0; bk < LBLK; bk++) {
      sum += subA[RM(bi,bk,LBLK)] * subB[RM(bk,bj,LBLK)];
    }

    __syncthreads();
  }

  dmatC[RM(i,j,N)] = sum;
}
```

Need barriers across thread block to ensure subA/subB are ready to be read/updated

(A block is executed as multiple warps, which can proceed at different rates through the kernel)

Benchmarking improved blocked matmul

- ./matrix -n 1024 -N 1024 -m cblock

- Simple C++: 1300 ms, 1.6 Gflops
- Simple CUDA: 33 ms, 65 Gflops
- Simple++ CUDA: 2.4 ms, 900 Gflops

- Block C++: 500 ms, 4.4 Gflops
- Block CUDA: 100 ms, 20 Gflops
- Block++ CUDA: 1.9ms, 1130 Gflops

Benchmarking at 2048 × 2048 (8 × more work)

- ./matrix -n 2048 -N 2048 -m ...

Big drop-off—data falls out of L3 cache

- Simple C++: 44000 ms, 0.4 Gflops
- Simple CUDA: 208 ms, 82 Gflops
- Simple++ CUDA: 18 ms, 940 Gflops

- Block C++: 5500 ms, 3.2 Gflops
- Block CUDA: 206 ms, 83 Gflops
- Block++ CUDA: 15ms, 1180 Gflops

Big improvement—increased parallelism

Debugging tips and pitfalls

- printf() is available, but will reorder or lose output
  - So be cautious using printf() for debugging!

- Check your error codes

```
#define CHK(ans) gpuAssert((ans),_FILE_,
__LINE__);

void gpuAssert(CUDAError_t code, const char *file,
int line){
    if (code != CUDASuccess)
        fprintf(stderr, "GPUassert: %s %s %s\n",
            CUDAGetErrorString(code), file, line);
}

#define POSTKERNEL CHK(CUDAPeekAtLastError())
```

Debugging tips and pitfalls

- Write reference version on host in C++
- Watch out for out-of-bounds memory errors (all kinds of crazy stuff will happen)
- Don't assume stuff about N (e.g., that it's a multiple of LBLK)
- cuda-gdb lets you step through + inspect code

Debugging tips and pitfalls

- What will happen here?

```
for (int k = 0; k < N; k+= LBLK) {
    if (i >= N || j >= N) continue;
    // Some computation
    __syncthreads();
    // Some more computation
    __syncthreads();
}
```

Optimization advice

- Get the high-level abstraction + implementation first
  - Don't start with low-level optimizations

- Use nvprof to figure out where your bottleneck is
  - Low utilization of compute + memory → no parallelism
  - Low utilization of compute → memory bound
  - Low utilization of memory → compute bound

- Memory is often key
  - E.g., when to use local/shared/global memory

CUDA syntax

- `__shared / global` : Place variable in block-/device-shared memory
- `cudaMalloc/cudaMemcpy/cudaFree`: Manage device memory (flag sets to/from device)
- `__syncthreads()`: Barrier within a thread block
- `kernel<<<blocks,threadsPerBlock>>>()`: Invoke kernel on device
- `blockIdx/threadIdx`: current block/thread idx
- `blockDim/gridDim`: Num threads per block/blocks per grid

CUDA as a vector processor

- NVIDIA has abused architecture terminology badly

CUDA/GPU Terminology	Classic vector terminology
Grid	Vectorizable loop
Thread	Loop iteration
Block	??
Warp	Thread
GPU/Device	Vector multicore
SM (streaming multiprocessor)	Core
Core	(Vector) Lane
Global memory	Memory
Shared memory	Local memory
Local memory	Registers