Demonstrating Locality of Reference on Multi-Cores and GPUs

Cosmin E. Oancea

Department of Computer Science (DIKU)
University of Copenhagen

September 2024, PMPH Lecture Slides

Locality of Reference at a Very High Level

Main Goal: demonstrate several "simple" techniques for optimizing locality of reference in the context of two different hardware: multi-core CPUs and GPGPUs.

Q: What is locality of reference?

- small set of addresses accessed at a time, named working set (low miss rate),
- when program transitions there is an abrupt change of working sets (characterized by high miss rate).

Q: What are the two main types of locality?

Locality of Reference at a Very High Level

Main Goal: demonstrate several "simple" techniques for optimizing locality of reference in the context of two different hardware: multi-core CPUs and GPGPUs.

Q: What is locality of reference?

- small set of addresses accessed at a time, named working set (low miss rate),
- when program transitions there is an abrupt change of working sets (characterized by high miss rate).

Q: What are the two main types of locality?

Spatial: items close-by a referenced item are likely to be accessed soon thereafter,

Temporal: a referenced item is likely to be accessed again in the near future,

- Spatial Locality gives raise to temporal locality at higher hwd levels (block/page).

Q: What types of memory are there?

Locality of Reference at a Very High Level

Main Goal: demonstrate several "simple" techniques for optimizing locality of reference in the context of two different hardware: multi-core CPUs and GPGPUs.

Q: What is locality of reference?

- small set of addresses accessed at a time, named working set (low miss rate),
- when program transitions there is an abrupt change of working sets (characterized by high miss rate).

Q: What are the two main types of locality?

Spatial: items close-by a referenced item are likely to be accessed soon thereafter,

Temporal: a referenced item is likely to be accessed again in the near future,

- Spatial Locality gives raise to temporal locality at higher hwd levels (block/page).

Q: What types of memory are there?

- Hard Disk,
- Global Memory (GM),
- Last-Level Cache (LL$) . . . Level-1 Cache (L1$),
- Registers (?)

Structure of the Lecture

(1) Flat representation of multi-dimensional arrays in memory;

(2) CPU vs GPU: Bird's Eye View;

(3) How do we measure/reason about Performance?

(4) Programming models demonstrated on simple examples:
  (4.1) OpenMP for multi-cores (very brief);
  (4.2) Cuda for GPUs;

(5) Case studies:
  (5.1) LL$ threshing: Histogram-like computation.
  (5.2) Spatial Locality: Transposition.
  (5.2) Optimizing Spatial Locality by Transposition.
  (5.3) L1$ and Register: Matrix-Matrix Multiplication.
  (5.4) L1$ and Register: Batch Matrix Multiplication under a Mask.

Teaching Method

The lecture is intended to present:

- the key differences between the CPU and GPU hardware,
- the essence of the programming models, with emphasis on what is needed to implement the four case studies,
- the rationale behind the techniques for optimizing locality for the 5 case studies: reason like a human (pictures) or as a compiler (optimization recipe) + code
- a practical demonstration of the impact of the discussed optimizations.

Theory is put into practice by you solving a set of ”fill-in-the-blanks” exercises that

- require implementing key parts of the code according to instructions;
- demonstrate significant performance gains (while validation still holds);
- allows easy digestion of the OpenMp & Cuda by pattern matching existing code.

We will use C-like notation/pseudo-code, which is hopefully easy to translate to Cuda.

Flat representation of multi-dimensional arrays in memory

CPU vs GPU: Bird's Eye View

How do we measure/reason about Performance?

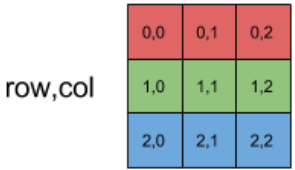
Programming Models Demonstrated on Simple Examples
- OpenMP
- Cuda

Five Case Studies
- LL$ threshing: Histogram-like computation
- Spatial Locality: Matrix Transposition
- Optimizing Spatial Locality by Transposition.
- L1$ and Register: Matrix-Matrix Multiplication
- L1$ and Register: Batch Matrix Multiplication under a Mask

Conclusions

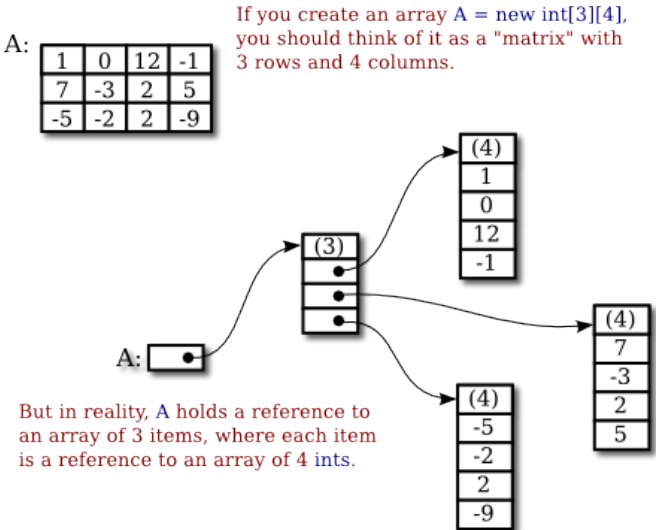
Multi-Dimensional Arrays

In this lecture, whenever we talk of a multi-dimensional array, **we mean this:**



Multi-Dimensional Arrays Disclaimer

In this lecture, whenever we talk of a multi-dimensional array, **we do NOT mean this:**



Multi-Dimensional Arrays in C

In C-like languages one can (statically) declare and use a multidimensional array in this way if and only if the dimension sizes $n_1 \ldots n_k$ are statically-known constants:

```
float arr[n1]...[nk];

for(int i1=0; i1<n1; i1++) {
  ...
    for(int ik=0; ik<nk; ik++) {
      arr[i1]...[ik] = i1 * ... * ik;
} ... }
```

Multi-Dimensional Arrays in Lecture's Notation

For convenience of notation, in the lecture, we will use the same notation even when sizes are not statically-known constants, e.g., are part of the program input. The C translation would be to dynamically allocate and work with a flat one-dimensional array of size $n_1 * \ldots * n_k$:

```
float* arr = (float*)malloc(n1*...*nk*sizeof(float));

for(int i1=0; i1<n1; i1++) {
  ...
    for(int ik=0; ik<nk; ik++) {
      arr[i1*n2*...*nk + ... + ik-1*nk + ik] =
                i1 * ... * ik;
} ... }
...
free(arr);
```

Flat representation of multi-dimensional arrays in memory

CPU vs GPU: Bird's Eye View

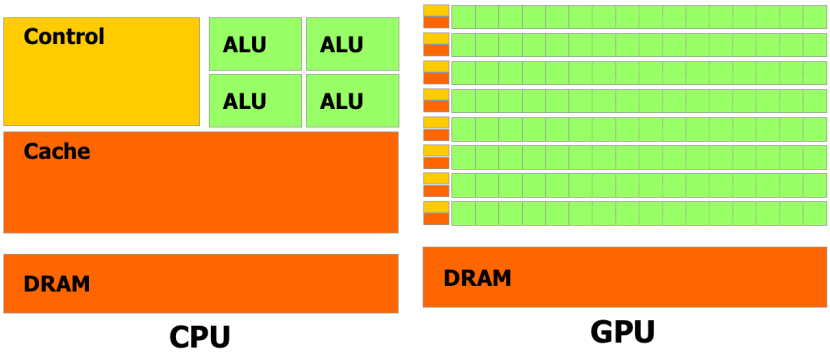
How do we measure/reason about Performance?

Programming Models Demonstrated on Simple Examples
- OpenMP
- Cuda

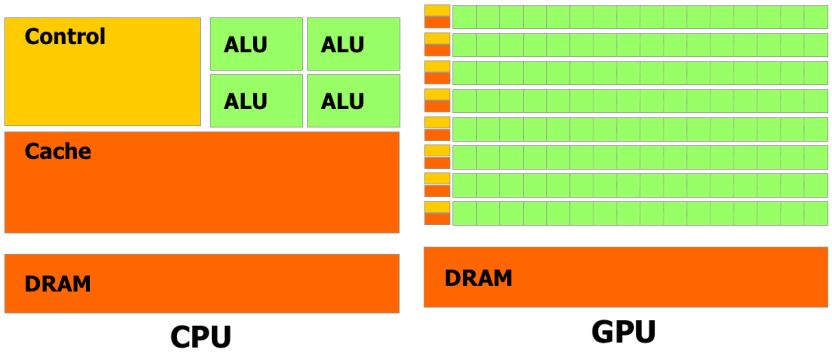
Five Case Studies
- LL$ threshing: Histogram-like computation
- Spatial Locality: Matrix Transposition
- Optimizing Spatial Locality by Transposition.
- L1$ and Register: Matrix-Matrix Multiplication
- L1$ and Register: Batch Matrix Multiplication under a Mask

Conclusions

Key Ideas in GPU Design

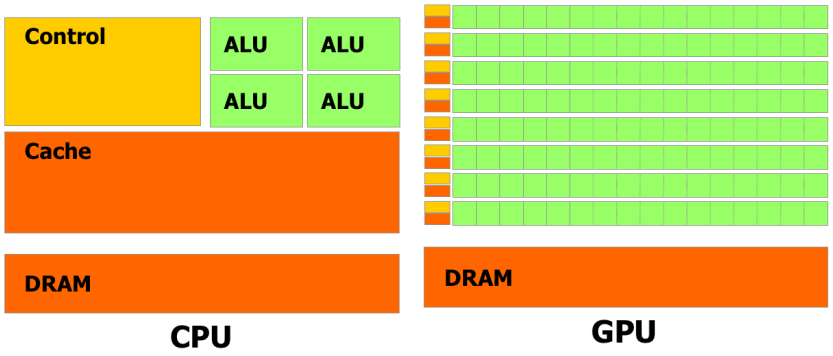


Key Ideas in GPU Design



1. Remove the hardware components that help a single instruction stream run fast,
2. SIMD: amortizes the management of an instruction stream across many ALUs,
3. Aggressively use hardware(-supported) multi-threading to hide latency.

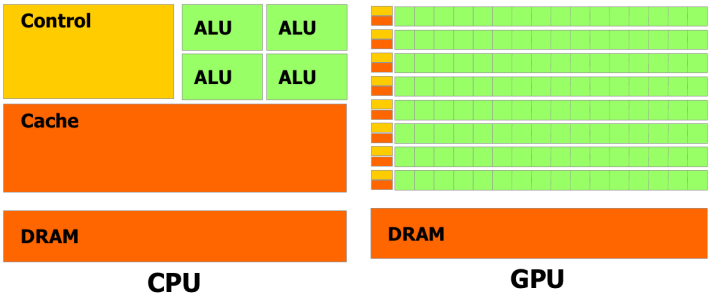
Key Ideas in GPU Design



1. Remove the hardware components that help a single instruction stream run fast,
2. SIMD: amortizes the management of an instruction stream across many ALUs,
3. Aggressively use hardware(-supported) multi-threading to hide latency.

Spatial locality to global memory means "coalesced accesses": threads executing in lock step a load/store SIMD instruction access consecutive memory locations!

CPUs compared to CPUs



- GPUs have *thousands* of simple cores and taking full advantage of their compute power requires *tens/hundred of thousands* of threads.
- GPU threads are very *restricted* in what they can do: no stack, no allocation, limited control flow, etc.
- Potential *very high performance* and *lower power usage* compared to CPUs, but programming them is *hard*.

Flat representation of multi-dimensional arrays in memory

CPU vs GPU: Bird's Eye View

How do we measure/reason about Performance?

Programming Models Demonstrated on Simple Examples
- OpenMP
- Cuda

Five Case Studies
- LL$ threshing: Histogram-like computation
- Spatial Locality: Matrix Transposition
- Optimizing Spatial Locality by Transposition.
- L1$ and Register: Matrix-Matrix Multiplication
- L1$ and Register: Batch Matrix Multiplication under a Mask

Conclusions

What Is Performance? How to Measure it?

(1) What is performance?

Performance measures the degree to which hardware resources are utilized.

(2) How do we measure performance?

2.1 So as to compare the performance of an implementation across datasets?

What Is Performance? How to Measure it?

(1) What is performance?

Performance measures the degree to which hardware resources are utilized.

(2) How do we measure performance?

2.1 So as to compare the performance of an implementation across datasets?

- If program has low arithmetic intensity $\Longrightarrow$ **memory bandwidth/throughput**:

$$\frac{\text{total number of bytes accessed}}{\text{Running time } (\mu s) \cdot 10^3} \quad \text{(GB/sec)}$$

- If program has high arithmetic intensity $\Longrightarrow$ **computational performance**:

$$\frac{\text{total number of float operations}}{\text{Running time } (\mu s) \cdot 10^3} \quad \text{(GFlop/sec)}$$

- If in between $\Longrightarrow$ roofline model.

2.2 How to reason about the degree of hardware utlization?

What Is Performance? How to Measure it?

(1) What is performance?

Performance measures the degree to which hardware resources are utilized.

(2) How do we measure performance?

2.1 So as to compare the performance of an implementation across datasets?

- If program has low arithmetic intensity $\Longrightarrow$ **memory bandwidth/throughput**:

$$\frac{\text{total number of bytes accessed}}{\text{Running time } (\mu s) \cdot 10^3} \quad \text{(GB/sec)}$$

- If program has high arithmetic intensity $\Longrightarrow$ **computational performance**:

$$\frac{\text{total number of float operations}}{\text{Running time } (\mu s) \cdot 10^3} \quad \text{(GFlop/sec)}$$

- If in between $\Longrightarrow$ roofline model.

2.2 How to reason about the degree of hardware utlization?

- compute the percentage achieved by your implementation relative to the peak memory bandwidth or peak flops performance of the hardware.
- if these are not listed, compare your performance with the best-known implementation of your algorithm for a certain hardware type, e.g., Cublas for MMM.

Comparing Performance Across Different Implementations

- . . .

2.3 How to compare performance across datasets & different implementations?

Comparing Performance Across Different Implementations

- ...

2.3 How to compare performance across datasets & different implementations?

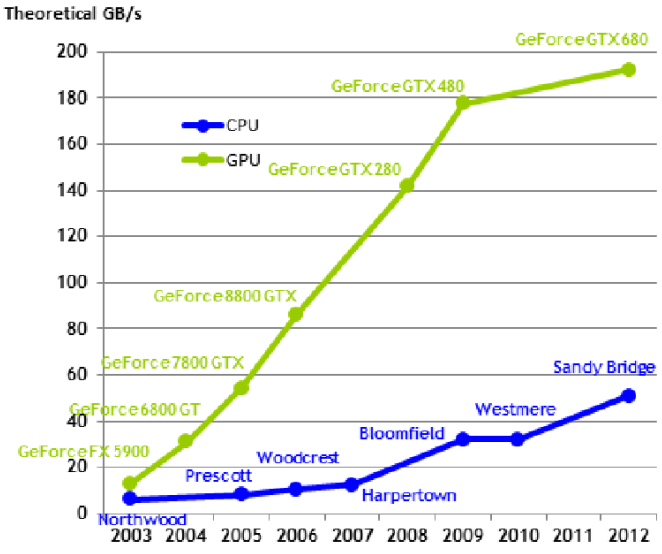
- Use the total number of bytes (or float ops) of the "golden sequential" implem!
- If top hardware performance not listed, sometimes it is useful to compare with simpler algorithms that have the same characteristics and are known to have near-optimal performance.

```
// Inclusive Prefix Sum:
// Input:  A = {a_0, ..., a_{n-1}}
// Result: X = {a_0, a_0 + a_1, ..., Σ_{i=0}^{n-1} a_i}
float acc = 0;
for(int i=0; i<n; i++) {
    acc = acc + A[i];
    X[i] = acc;
}
```

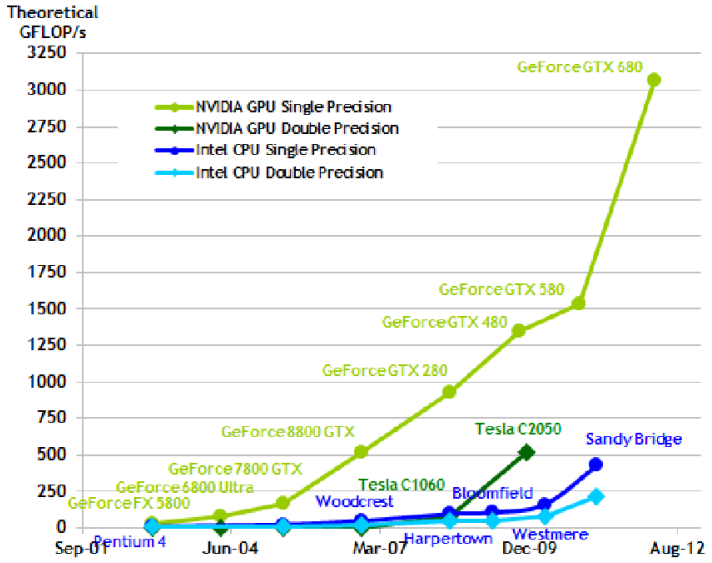
```
// Memcpy
// Input:  A = {a_0, ..., a_{n-1}}
// Result: X = {a_0, ..., a_{n-1}}

for(int i=0; i<n; i++) {
    X[i] = A[i];
}
```

- **Prefix sum** is challenging to implement efficiently for GPU;
- **Memcpy** is trivial and has the same access pattern: *n* reads + *n* writes.
- If your prefix scan reaches 80% of memcpy's parallel performance $\Longrightarrow$ happy!

Peak Memory Performance: GPU vs CPU



Peak Computational Performance: GPU vs CPU



Flat representation of multi-dimensional arrays in memory

CPU vs GPU: Bird's Eye View

How do we measure/reason about Performance?

Programming Models Demonstrated on Simple Examples
- OpenMP
- Cuda

Five Case Studies
- LL$ threshing: Histogram-like computation
- Spatial Locality: Matrix Transposition
- Optimizing Spatial Locality by Transposition.
- L1$ and Register: Matrix-Matrix Multiplication
- L1$ and Register: Batch Matrix Multiplication under a Mask

Conclusions

Multi-Core Programming with OpenMP

Disclaimer: we just discuss simple features of OpenMP that are used in the exercises.

Trivial example: multiplying each element of a matrix by 2:

```
for(int i=0; i<n; i++) {
  for(int j=0; j<n; j++) {
    Y[i][j] = 2 * X[i][j];
} }
```

Multi-Core Programming with OpenMP

Disclaimer: we just discuss simple features of OpenMP that are used in the exercises.

Trivial example: multiplying each element of a matrix by 2:

```
for(int i=0; i<n; i++) {
  for(int j=0; j<n; j++) {
    Y[i][j] = 2 * X[i][j];
} }
```

can be fully parallelized by inserting a simple pragma annotation:

```
#pragma omp parallel for collapse(2) schedule(static)
for(int i=0; i<m; i++) {
  for(int j=0; j<n; j++) {
    Y[i][j] = 2 * X[i][j];
} }
```

- `#pragma omp parallel for`: **"I solemny swear that the following loop is parallel!"**
- `collapse 2`: **"I solemny swear that the following two loops are parallel; please merge/flatten them!"**
- `schedule(static)`: loop iterations are divided into number-of-processor, nearly-equal contiguous chunks; each thread executes its chunk.
- `schedule(dynamic)`: the earliest non-executed iteration is assigned to the first thread that asks for it (i.e., dynamic, first-come, first-served mechanism).

What is suboptimal in this code?

```
#pragma omp parallel for schedule(static)
for(int i=m; i>0; i=i-1) { // parallel
  float tmp = X[i];
  for(int j=0; j<i*i; j++) { // sequential
    tmp = sqrt(tmp) * 2.0;
  }
  Y[i] = tmp;
}
```

What is suboptimal in this code?

```
#pragma omp parallel for schedule(static)
for(int i=m; i>0; i=i-1) { // parallel
  float tmp = X[i];
  for(int j=0; j<i*i; j++) { // sequential
    tmp = sqrt(tmp) * 2.0;
  }
  Y[i] = tmp;
}
```

Iterations are imbalanced, please use `schedule(dynamic)` instead!

- **`schedule(dynamic)`**: the earliest non-executed iteration is assigned to the first thread that asks for it (dynamic, first-come, first-served mechanism).
- **`schedule(dynamic,`** `chunk_size`**)**: like dynamic, but `chunk_size` iterations are assigned to a thread.

Unrelated but useful: the number of utilized threads can be changed by setting the `OMP_NUM_THREADS` environment variable in the terminal you use to run the program:

```
$ export OMP_NUM_THREADS = 8
```

Multi-Core Programming with OpenMP: privatization

This breaks your solemn vow! Why?

```
float x;
#pragma omp parallel for
for(int i=0; i<n; i++) {
    x = X[i];
    Y[i] = 2 * x;
}
```

Multi-Core Programming with OpenMP: privatization

This breaks your solemn vow! Why?

```
float x;
#pragma omp parallel for
for(int i=0; i<n; i++) {
    x = X[i];
    Y[i] = 2 * x;
}
```

Because there are races on x. It can be fixed in two ways:

```
#pragma omp parallel for
for(int i=0; i<n; i++) {
    float x = X[i];
    Y[i] = 2 * x;
}
```

```
float x;
#pragma omp parallel for private(x)
for(int i=0; i<n; i++) {
    x = X[i];
    Y[i] = 2 * x;
}
```

Similarly, the code on the right is not parallel since there are races on `i`:

```
int i;
#pragma omp parallel for
for(i=0; i<n; i++) {
    Y[i] = 2.0 * X[i];
}
```

```
#pragma omp parallel for
for(int i=0; i<n; i++) {
    Y[i] = 2.0 * X[i];
}
```

Flat representation of multi-dimensional arrays in memory

CPU vs GPU: Bird's Eye View

How do we measure/reason about Performance?

Programming Models Demonstrated on Simple Examples

- OpenMP
- Cuda

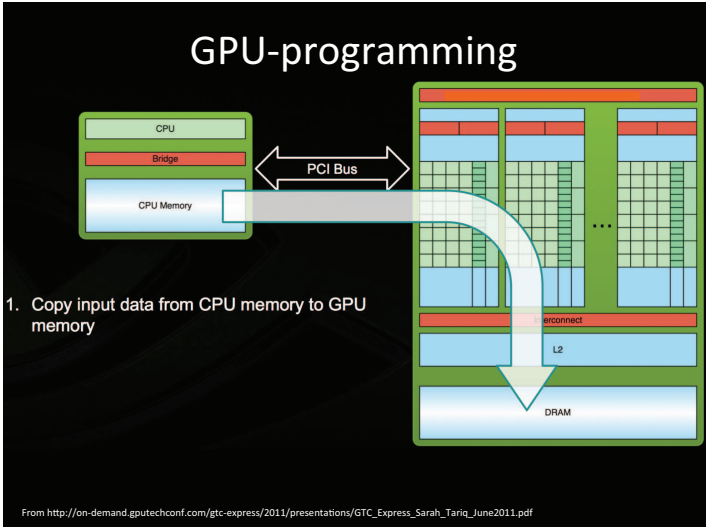
Five Case Studies

- LL$ threshing: Histogram-like computation
- Spatial Locality: Matrix Transposition
- Optimizing Spatial Locality by Transposition.
- L1$ and Register: Matrix-Matrix Multiplication
- L1$ and Register: Batch Matrix Multiplication under a Mask

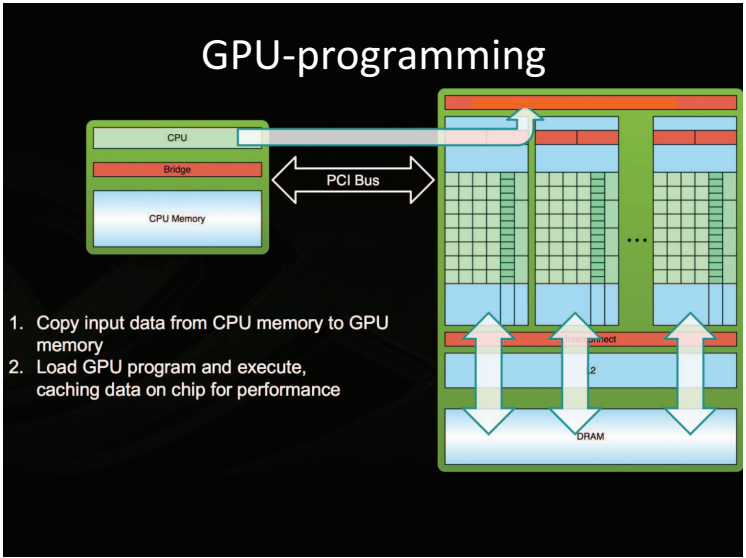
Conclusions

Basic GPU Programming

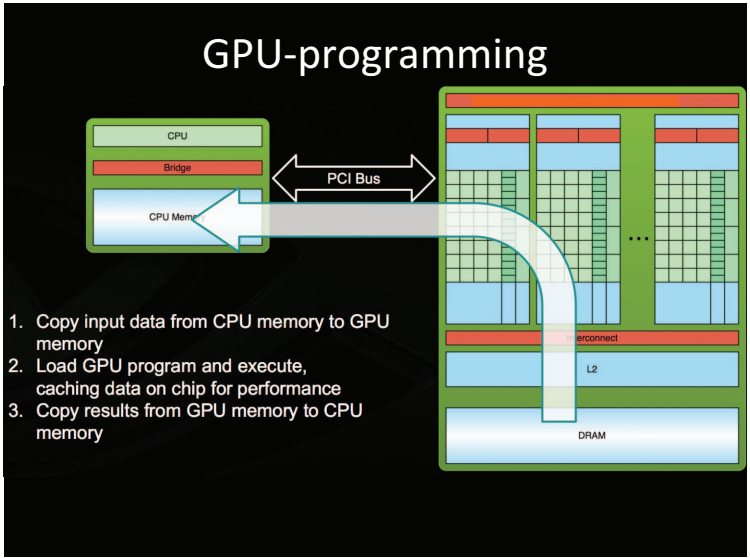
The device (GPU) and host (CPU) have different memory spaces!



Basic GPU Programming

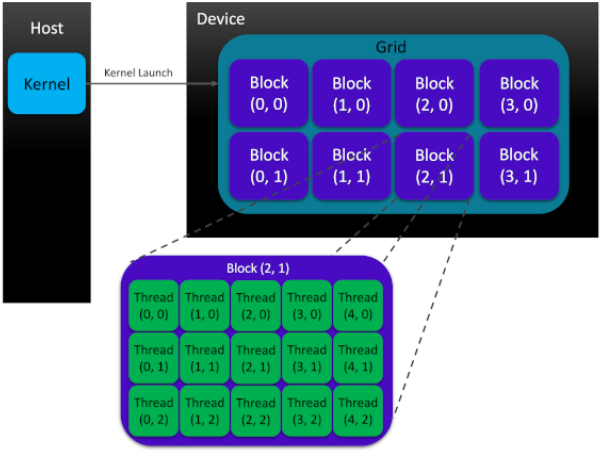


Basic GPGPU programming



Cuda: Grid-Block Structure of Threads

Credit: pictures taken from `http://education.molssi.org/gpu_programming_beginner/03-cuda-`

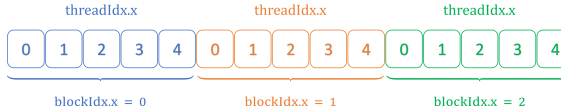


Blocks and Grids have at most three dimensions—denoted x, y, z, with x innermost and z outermost. Their sizes are specified at kernel launch. Inside the **kernel** you may use:

- `blockDim.x`: block size in dim x
- `blockIdx.x`: current block index (in x)
- `threadIdx.x`: local index of the current thread inside its block (in dim x)
- `gridDim.x` number of blocks on dim x
- Ditto for dimensions y and z.

The global thread index in dim $q \in \{x, y, z\}$**:**

`threadIdx.q`$+$**`blockIdx.q`**$\cdot$**`blockDim.q`**



Cuda: Multiply with 2 Each Element of an Array

Golden Sequential:

```
// Y and X are arrays of length n
for(int i=0; i<n; i++) {
    Y[i] = 2.0 * X[i];
}
```

Cuda: Multiply with 2 Each Element of an Array

Golden Sequential:

```
// Y and X are arrays of length n
for(int i=0; i<n; i++) {
    Y[i] = 2.0 * X[i];
}
```

Cuda Kernel:

```
__global__ void mul2Kernel(float* X, float* Y, int n) {
  // compute global thread id in dimension x
  const unsigned int gid = blockIdx.x * blockDim.x + threadIdx.x;
  if( gid < n ) {  // don't access out of bounds
    Y[gid] = 2.0 * X[gid];
  }
}
```

Calling the kernel from host/CPU-executed code:

```
unsigned int B = 256;  // chose a suitable block size in dimension x
unsigned int numblocks = (n + B - 1) / B; // number of blocks in dimension x
dim3 block(B,1,1), grid(numblocks,1,1);   // total number of threads (numblocks*B) may overshoot n!
mul2Kernel<<<grid, block>>>(d_X, d_Y, n); //call kernel, d_X and d_Y are in device memory
```

Cuda: Putting Together the Multiply-by-2 Example

```
int main (int argc, char * argv[]) {
  // 1. check validity of program input
  if (argc != 2) {
    printf( "Usage: %s <array-length>\n"
          , argv[0] );
    exit(1);
  }
  // 2. read program input as an integer
  const unsigned int n = atoi(argv[1]);

  // 3. allocate mem on host(h_) & device(d_)
  unsigned int mem_size = n * sizeof(float);
  float* h_X = (float*) malloc(mem_size);
  float* h_Y = (float*) malloc(mem_size);
  float *d_X, *d_Y;
  cudaMalloc((void**) &d_X,     mem_size);
  cudaMalloc((void**) &d_Y,     mem_size);

  // 4. random initialization of h_X
  for(unsigned int i=0; i<n; i++)
    h_X[i] = rand() / (float)RAND_MAX;
```

Cuda: Putting Together the Multiply-by-2 Example

```
int main (int argc, char * argv[]) {
  // 1. check validity of program input
  if (argc != 2) {
    printf( "Usage: %s <array-length>\n"
          , argv[0] );
    exit(1);
  }
  // 2. read program input as an integer
  const unsigned int n = atoi(argv[1]);

  // 3. allocate mem on host(h_) & device(d_)
  unsigned int mem_size = n * sizeof(float);
  float* h_X = (float*) malloc(mem_size);
  float* h_Y = (float*) malloc(mem_size);
  float *d_X, *d_Y;
  cudaMalloc((void**) &d_X,     mem_size);
  cudaMalloc((void**) &d_Y,     mem_size);

  // 4. random initialization of h_X
  for(unsigned int i=0; i<n; i++)
    h_X[i] = rand() / (float)RAND_MAX;

  // 5. copy host memory to device
  cudaMemcpy( d_A, h_A, mem_size
            , cudaMemcpyHostToDevice );

  // 6. create block, grid
  unsigned int B = 256;
  unsigned int numblocks = (n + B - 1) / B;
  dim3 block(B,1,1), grid(numblocks,1,1);

  // 7. call kernel
  mul2Kernel<<<grid, block>>>(d_X, d_Y, n);

  // 8. copy the result from device to host
  cudaMemcpy( h_Y, d_Y, mem_size
            , cudaMemcpyDeviceToHost );
  ...

  // 9. free host and device memory
  free(h_X); cudaFree(d_X);
  free(h_Y); cudaFree(d_Y);
}
```

Flat representation of multi-dimensional arrays in memory

CPU vs GPU: Bird's Eye View

How do we measure/reason about Performance?

Programming Models Demonstrated on Simple Examples
- OpenMP
- Cuda

Five Case Studies
- LL$ threshing: Histogram-like computation
- Spatial Locality: Matrix Transposition
- Optimizing Spatial Locality by Transposition.
- L1$ and Register: Matrix-Matrix Multiplication
- L1$ and Register: Batch Matrix Multiplication under a Mask

Conclusions

Histogram-Like Computation: Golden Sequential

```
void goldenSeq( uint32_t* inp_inds // length N
              , float*    inp_vals // length N
              , float*    hist     // length H
              , const uint32_t N, const uint32_t H
) {
    // Is this loop parallel?
    for(uint32_t i = 0; i < N; i++) {
        uint32_t ind = inp_inds[i];
        float    val = inp_vals[i];

        // accumulate val to position index in "histogram"
        if(0 <= ind && ind < H) { // sanity, expected to hold.
            hist[ind] += val;
        }
}   }
```

Histogram-Like Computation: Golden Sequential

```
void goldenSeq( uint32_t* inp_inds // length N
              , float*    inp_vals // length N
              , float*    hist     // length H
              , const uint32_t N, const uint32_t H
) {
    // Is this loop parallel?
    for(uint32_t i = 0; i < N; i++) {
        uint32_t ind = inp_inds[i];
        float    val = inp_vals[i];

        // accumulate val to position index in "histogram"
        if(0 <= ind && ind < H) { // sanity, expected to hold.
            hist[ind] += val;
        }
}   }
```

This is a generalized reduction:

- If all loop-carried dependencies are due to arrays that are accessed only in accumulation stmts with the same commutative and associative operator $\odot$ (e.g., +, *, min, max), such as `hist[ ind_exp ] ⊙= val_exp` and nowhere else, i.e., `hist` cannot appear in `ind_exp` or `val_exp`, or in any other non-accumulation stmt,
- **Then the loop can be parallelized by performing the accumulations atomically!**
- Useful properties of parallel loops transfer to generalized reductions, e.g., interchange & distribution.

Histogram-Like Computation: Direct OpenMP Parallelization

```
void goldenSeq( uint32_t* inp_inds // length N
              , float*    inp_vals // length N
              , float*    hist     // length H
              , const uint32_t N, const uint32_t H
) {
    #pragma omp parallel for schedule(static)
    for(uint32_t i = 0; i < N; i++) {
        uint32_t ind = inp_inds[i];
        float    val = inp_vals[i];

        if(0 <= ind && ind < H) { // sanity, expect to hold.
            #pragma omp atomic
            hist[ind] += val;
        }
}   }
```

What happens if indices are random and `hist`'s size is several time bigger than LL$?

Histogram-Like Computation: Direct OpenMP Parallelization

```
void goldenSeq( uint32_t* inp_inds // length N
              , float*    inp_vals // length N
              , float*    hist     // length H
              , const uint32_t N, const uint32_t H
) {
    #pragma omp parallel for schedule(static)
    for(uint32_t i = 0; i < N; i++) {
        uint32_t ind = inp_inds[i];
        float    val = inp_vals[i];

        if(0 <= ind && ind < H) { // sanity, expect to hold.
            #pragma omp atomic
            hist[ind] += val;
        }
}   }
```

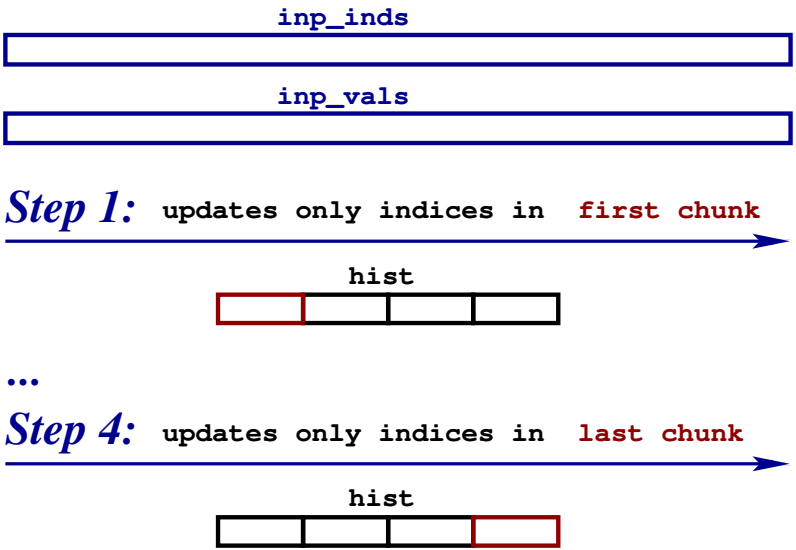
What happens if indices are random and `hist`'s size is several time bigger than LL$?

Answer: LL$ threshing!

How can we optimize that?

[1] T. Henriksen, S. Hellfritzsch, P. Sadayappan and C. Oancea, "Compiling Generalized Histograms for GPU", In Procs of SC20.

Histogram-Like: Multi-Pass Optimization by Picture



Step i traverses all the input indices stored in `inp_inds` but updates the histogram only on the indices that fall within the **i^{th} chunk**.

Histogram-Like Computation: Multi-Pass Optimization (OpenMP)

```
void multiStep( uint32_t* inp_inds // length N
              , float*    inp_vals // length N
              , float*    hist     // length H
              , const uint32_t N, const uint32_t H
              , const uint32_t L3
              ) {
  // we use L3_FRAC = 3/7 of L3 cache to hold `hist`
  uint32_t CHUNK = (L3_FRAC * L3) / sizeof(float);
  uint32_t num_partitions = (H+CHUNK-1) / CHUNK;
  // sequentially process each chunk
  for(uint32_t k=0; k<num_partitions; k++) {
    // in here, we process only indices falling
    // in interval [k*CHUNK ... (k+1)*CHUNK-1]
    uint32_t low_bound = k*CHUNK;
    uint32_t upp_bound = min( (k+1)*CHUNK, H );
    #pragma omp parallel for schedule(static)
    for(uint32_t i = 0; i < N; i++) {
      uint32_t ind = inp_inds[i];
      float    val = inp_vals[i];
      if(ind >= low_bound && ind < upp_bound) {
        #pragma omp atomic
        hist[ind] += val;
} } } }
```

Histogram-Like Computation: Multi-Pass Optimization (OpenMP)

```
void multiStep( uint32_t* inp_inds // length N
              , float*    inp_vals // length N
              , float*    hist     // length H
              , const uint32_t N, const uint32_t H
              , const uint32_t L3
              ) {
  // we use L3_FRAC = 3/7 of L3 cache to hold 'hist'
  uint32_t CHUNK = (L3_FRAC * L3) / sizeof(float);
  uint32_t num_partitions = (H+CHUNK-1) / CHUNK;
  // sequentially process each chunk
  for(uint32_t k=0; k<num_partitions; k++) {
    // in here, we process only indices falling
    // in interval [k*CHUNK ... (k+1)*CHUNK-1]
    uint32_t low_bound = k*CHUNK;
    uint32_t upp_bound = min( (k+1)*CHUNK, H );
    #pragma omp parallel for schedule(static)
    for(uint32_t i = 0; i < N; i++) {
      uint32_t ind = inp_inds[i];
      float    val = inp_vals[i];
      if(ind >= low_bound && ind < upp_bound) {
        #pragma omp atomic
        hist[ind] += val;
} } } }
```

We use a multi-pass technique [1] that:

- partitions the histogram into q chunks, such as a chunk fits in LL$,
- process each histogram chunk in parallel by (redundantly) traversing the whole input (and ignoring the indices that do not fall into the currently-processed histogram chunk).
- Run demo: breaks even on CPU, but very beneficial on GPU!
- Dummy technique but surprisingly effective—recomputation allows the resident set to fit the LL$!

Histogram-Like Computation: Cuda Exercise 1

The programming exercise is to implement the multi-pass technique in Cuda, by pattern-matching the provided "naive" implementation (folder histo-L3-thrashing).

The host & device code (files main-gpu.cu & kernels.cu.h) for multi-step optimization dummily uses the naive approach:

```
template<int B> void
multiStepHisto( uint32_t* d_inp_inds
              , float* d_inp_vals
              , float*     d_hist
              , const uint32_t N
              , const uint32_t H
              , const uint32_t L3 ) {
  // these are correct, do not touch :))
  uint32_t grid = (N + B - 1) / B;
  cudaMemset(d_hist, 0, H * sizeof(float));

  // introduce the chunking loop similar to parallelPlan.h
  // and set the correct lower/upper bounds to kernel call
  // then modify the multiStepKernel code in kernels.cu.h
  multiStepKernel<<<grid,B>>>
    (d_inp_inds, d_inp_vals, d_hist, N, 0, H);
}
```

Histogram-Like Computation: Cuda Exercise 1

The programming exercise is to implement the multi-pass technique in Cuda, by pattern-matching the provided “naive” implementation (folder histo-L3-thrashing).

The host & device code (files main-gpu.cu & kernels.cu.h) for multi-step optimization dummily uses the naive approach:

```
template<int B> void
multiStepHisto( uint32_t* d_inp_inds
              , float* d_inp_vals
              , float*     d_hist
              , const uint32_t N
              , const uint32_t H
              , const uint32_t L3 ) {
  // these are correct, do not touch :))
  uint32_t grid = (N + B - 1) / B;
  cudaMemset(d_hist, 0, H * sizeof(float));

  // introduce the chunking loop similar to parallelPlan.h
  // and set the correct lower/upper bounds to kernel call
  // then modify the multiStepKernel code in kernels.cu.h
  multiStepKernel<<<grid,B>>>
    (d_inp_inds,d_inp_vals,d_hist,N,0,H);
}
```

```
__global__ void multiStepKernel (
    uint32_t* inp_inds, float* inp_vals,
    volatile float* hist, const uint32_t N,
    const uint32_t LB, const uint32_t UB ) {
// LB and UB are the (inclusive) lower and (exclusive) upper
// bounds of indices falling in the current chunk of hist
  uint32_t gid =  blockIdx.x*blockDim.x +
                  threadIdx.x;
  if(gid < N) {
    uint32_t ind = inp_inds[gid];

    // change the if condition to succeed when 'ind'
    // is within the bounds of the current chunk.
    if(ind < H) {
      float val = inp_vals[gid];
      atomicAdd((float*)&hist[ind], val);
} } }
```

Histogram-Like Computation: OpenMP & Cuda

The last argument of the program is the size of the LL$. Please set it according to the CPU/GPU hardware on which you are running, if you would like to observe impact!

For example, in `histo-L3-thrashing/Makefile`:

```
...
run_gpu: $(EXEC_GPU)
    ./$(EXEC_GPU) 536870912 41943040

run_cpu: $(EXEC_CPU)
    ./$(EXEC_CPU) 536870912 134217728
...
```

Those are the sizes in bytes of the LL$ of

- Nvidia's A100 GPU (40MB)
- AMD EPYC 7352 24-Core CPU (128MB)

Flat representation of multi-dimensional arrays in memory

CPU vs GPU: Bird's Eye View

How do we measure/reason about Performance?

Programming Models Demonstrated on Simple Examples

- OpenMP
- Cuda

Five Case Studies

- LL$ threshing: Histogram-like computation
- Spatial Locality: Matrix Transposition
- Optimizing Spatial Locality by Transposition.
- L1$ and Register: Matrix-Matrix Multiplication
- L1$ and Register: Batch Matrix Multiplication under a Mask

Conclusions

Matrix Transposition: Golden Sequential

Picture courtesy of
https://inst.eecs.berkeley.edu/~cs61c/su13/labs/06/

```
void goldenSeq( float* A     // [heightA][widthA]
              , float* A_tr  // [widthA][heightA]
              , const int heightA
              , const int widthA ) {

  #pragma omp parallel for collapse(2)
  for(int i = 0; i < heightA; i++) {
    for(int j=0; j < widthA; j++) {
        A_tr[j*heightA + i] = A[i*widthA + j];
        // A_tr[j][i] = A[i][j];
    }
  }
}
```

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Transpose

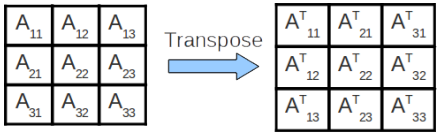
1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24
5	10	15	20	25

Probably not very efficient. How do we speed it up?

Matrix Transposition: Blocked Version for OpenMP by Picture

Picture courtesy of
https://inst.eecs.berkeley.edu/~cs61c/su13/labs/06/

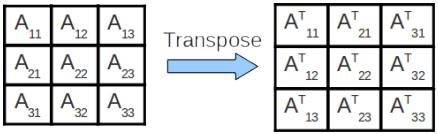
```
template<int TILE> void blockedTranspose (
                    float* A     // [heightA][widthA]
                  , float* A_tr  // [widthA][heightA]
                  , const int heightA
                  , const int widthA ) {

  #pragma omp parallel for collapse(2)
  for(int ii=0; ii<heightA; ii+=TILE) {
    for(int jj=0; jj<widthA; jj+=TILE) {

        for(int i=ii; i<min(ii+TILE,heightA); i++) {
          for(int j=jj; j<min(jj+TILE,widthA); j++){
            A_tr[j*heightA + i] = A[i*widthA + j];
            // A_tr[j][i] = A[i][j];
          }
        }
} } }
```



Matrix Transposition: Blocked Version for OpenMP by Picture

Picture courtesy of
https://inst.eecs.berkeley.edu/~cs61c/su13/labs/06/

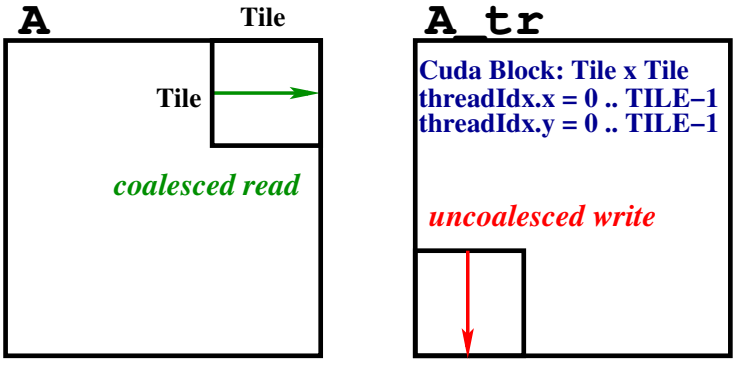
```
template<int TILE> void blockedTransposition (
                    float* A     // [heightA][widthA]
                  , float* A_tr  // [widthA][heightA]
                  , const int heightA
                  , const int widthA ) {

  #pragma omp parallel for collapse(2)
  for(int ii=0; ii<heightA; ii+=TILE) {
    for(int jj=0; jj<widthA; jj+=TILE) {

        for(int i=ii; i<min(ii+TILE,heightA); i++) {
          for(int j=jj; j<min(jj+TILE,widthA); j++){
            A_tr[j*heightA + i] = A[i*widthA + j];
            // A_tr[j][i] = A[i][j];
          }
        }
} } }
```



- Blocks have size TILE×TILE, with TILE a multiple of memory-block size;
- **If the block fits in cache, spatial locality of A_tr is optimized.**
- **Best performance on multi-core requires blocking each $ level.**

Matrix Transposition: Blocked Version for GPU/Cuda by Picture

The blocking technique will help some, but it would not fully solved the problem.

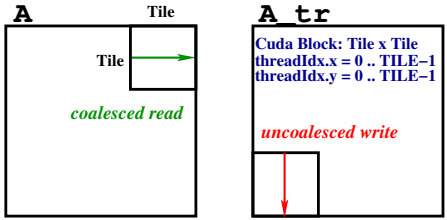


Spatial locality on GPU
≡
coalesced access to global memory.

Coalesced Access: a group of threads (named warp in Cuda) executing in lock step a load/store SIMD instruction access consecutive memory locations (a contiguous chunk of memory locations). This is the exact oposite of CPU's spatial locality!

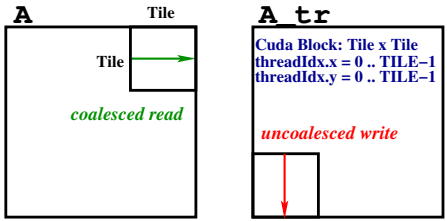
Cuda Transposition: Naive Uncoalesced Version

```
{ // Host/CPU Code, i.e., calling the Cuda Kernel:
  int  dimy = (heightA + TILE - 1) / TILE;
  int  dimx = (widthA + TILE - 1) / TILE;
  dim3 block(TILE, TILE, 1);
  dim3 grid (dimx, dimy, 1);

  naiveTransposeKer<<< grid, block >>>
           (d_A, d_A_tr, heightA, widthA);
}

// Cuda Kernel Code:
__global__ void naiveTransposeKer(
        float* A, float* A_tr, int heightA, int widthA
) {
    int gidx = blockIdx.x*blockDim.x + threadIdx.x;
    int gidy = blockIdx.y*blockDim.y + threadIdx.y;

    if( (gidx >= widthA) || (gidy >= heightA) ) return;

    A_tr[gidx*heightA + gidy] = A[gidy*widthA + gidx];

//  A_tr[gidx][gidy] = A[gidy][gidx];
}
```



Cuda Transposition: Naive Uncoalesced Version

```
{ // Host/CPU Code, i.e., calling the Cuda Kernel:
  int  dimy = (heightA + TILE - 1) / TILE;
  int  dimx = (widthA + TILE - 1) / TILE;
  dim3 block(TILE, TILE, 1);
  dim3 grid (dimx, dimy, 1);

  naiveTransposeKer<<< grid, block >>>
           (d_A, d_A_tr, heightA, widthA);
}

// Cuda Kernel Code:
__global__ void naiveTransposeKer(
        float* A, float* A_tr, int heightA, int widthA
) {
    int gidx = blockIdx.x*blockDim.x + threadIdx.x;
    int gidy = blockIdx.y*blockDim.y + threadIdx.y;

    if( (gidx >= widthA) || (gidy >= heightA) ) return;

    A_tr[gidx*heightA + gidy] = A[gidy*widthA + gidx];

//  A_tr[gidx][gidy] = A[gidy][gidx];
}
```



If `TILE == 32`, i.e., the warp size, then:

- Consecutive threads in a Cuda block will have **the same** `threadIdx.y` **and consecutive** `threadIdx.x`;
- Hence `A[gidy][gidx]` is coalesced, i.e., accesses consecutive locations;
- `A_tr[gidx][gidy]` is uncoalesced, i.e., results in a strided access with stride equal to `heightA`.

More Cuda: Threads in a Cuda Block Can Use Shared Memory & Barriers

The threads inside a Cuda block can communicate by means of shared (scratchpad) memory and barrier synchronization:

- shared memory has order-of-magnitude lower latency than global memory;
  - **uncoalesced accesses to shared memory do not affect performance.**
  - shared memory used as a staging buffer for global memory (user-managed cache).
- inside a Cuda kernel, one may declare a $T \times T$ 2D array stored in shared memory:

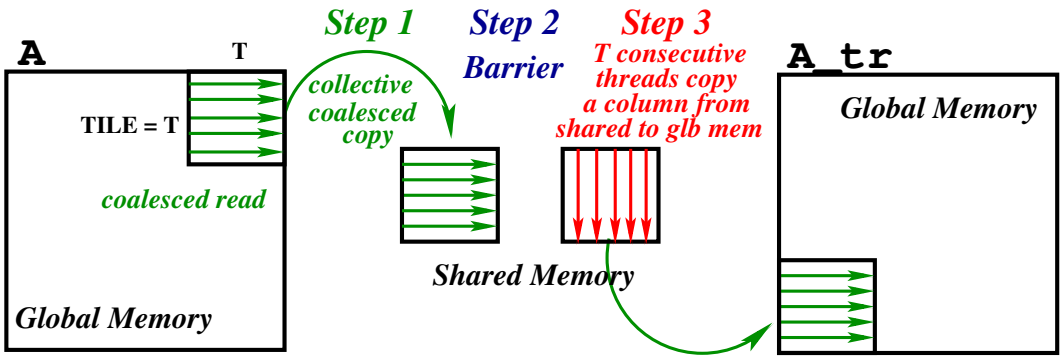
```
__shared__ float tile[T][T];
```

- threads in the same block can be synchronized by means of barriers:

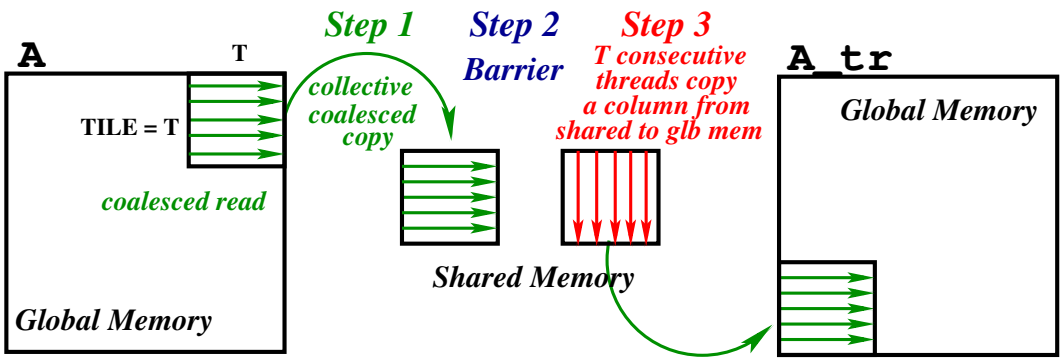
```
__syncthreads();
```

- all threads **must** reach the barrier in order for any to proceed further.
  - **Important Consequence:** if you place a barrier inside an `if` branch that is not taken by all threads, then non-termination is possible.

Cuda Transposition: Picture Recipe for Achieving Coalesced Access



Cuda Transposition: Picture Recipe for Achieving Coalesced Access



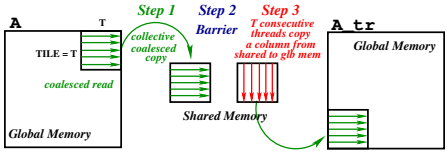
- Step 1 collectively copies with the threads of a Cuda block the corresponding matrix block from global memory to shared memory in coalesced fashion.
- Step 2 inserts a barrier to ensure that all threads have finished copying.
- In Step 3, every `T=TILE` consecutive threads (i.e., having the same value for `threadIdx.y` but different for `threadIdx.x`) copy a column from shared memory and place it as a row in global memory. Hence the access to the global memory of `A_tr` is coalesced, i.e., `TILE` consecutive threads write consecutive locations.
- Note that, in Step 3, the read from shared memory is uncoalesced, but shared memory does not suffer from it!

Cuda Transposition: Coalesced Version

```
{ // Host/CPU Code, i.e., calling the Cuda Kernel:
  int  dimy = (heightA + TILE - 1) / TILE;
  int  dimx = (widthA + TILE - 1) / TILE;
  dim3 block(TILE, TILE, 1), grid (dimx, dimy, 1);
  coalsTransposeKer<TILE><<< grid, block >>>
            (d_A, d_A_tr, heightA, widthA);
}
// Cuda Kernel Code:
template <int TILE> __global__ void coalsTransposeKer (
        float* A, float* A_tr, int heightA,int widthA){
  __shared__ float shmem[TILE][TILE+1];
  int x = blockIdx.x * T + threadIdx.x;
  int y = blockIdx.y * T + threadIdx.y;
  if( x < widthA && y < heightA )   // Step 1
    shmem[threadIdx.y][threadIdx.x] = A[y*widthA + x];

  __syncthreads(); // Step 2

  x = blockIdx.y * T + threadIdx.x;
  y = blockIdx.x * T + threadIdx.y;
  if( x < heightA && y < widthA )   // Step 3
    A_tr[y*heightA + x]  =  shmem[threadIdx.x][threadIdx.y];
}
```

Time for Demo!

This implementation can be further improved ($\sim 2\times$ speedup) but we aim to keep it simple!



Why not `shmem[TILE][TILE]`?

- number of banks of shared memory is 16 or 32 for Nvidia;
- common `TILE` values are 16 or 32;
- 16 consecutive threads would read the same memory bank at a time;
- Solution: `shmem[TILE][TILE+1]`

Flat representation of multi-dimensional arrays in memory

CPU vs GPU: Bird's Eye View

How do we measure/reason about Performance?

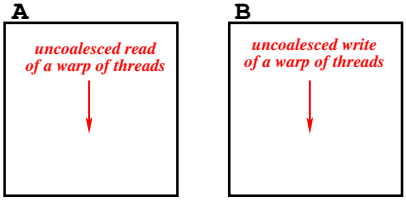
Programming Models Demonstrated on Simple Examples
- OpenMP
- Cuda

Five Case Studies
- LL$ threshing: Histogram-like computation
- Spatial Locality: Matrix Transposition
- Optimizing Spatial Locality by Transposition.
- L1$ and Register: Matrix-Matrix Multiplication
- L1$ and Register: Batch Matrix Multiplication under a Mask

Conclusions

Golden Sequential Version of a Contrived but Illustrative Program

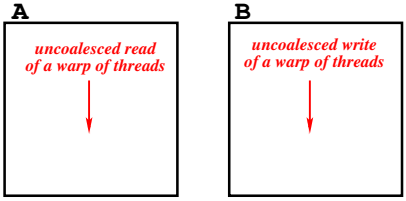
```
void goldenSeq( float* A  // [num_rows][num_cols]
              , float* B  // [num_rows][num_cols]
              , const uint64_t num_rows
              , const uint64_t num_cols
) {
    #pragma omp parallel for schedule(static)
    for(uint64_t i = 0; i < num_rows; i++) { // parallel
        float accum = 0.0;

        // this loop cannot be parallelized due to accum
        for(uint64_t j = 0; j < num_cols; j++) {
            float a_el  = A[i*num_cols + j];
        //  float a_el = A[i][j]
            accum = sqrt(accum) + a_el*a_el;
            B[i*num_cols + j] = accum;
        //  B[i][j] = accum;
        }
    }
}
```



- Each thread reads/writes an entire row of A/B;
- Perfect spatial locality for multi-core execution.
- **Uncoalesced access for GPU execution (terrible). What to do?**

Golden Sequential Version of a Contrived but Illustrative Program

```
void goldenSeq( float* A  // [num_rows][num_cols]
              , float* B  // [num_rows][num_cols]
              , const uint64_t num_rows
              , const uint64_t num_cols
) {
    #pragma omp parallel for schedule(static)
    for(uint64_t i = 0; i < num_rows; i++) { // parallel
        float accum = 0.0;

        // this loop cannot be parallelized due to accum
        for(uint64_t j = 0; j < num_cols; j++) {
            float a_el  = A[i*num_cols + j];
        //  float a_el = A[i][j]
            accum = sqrt(accum) + a_el*a_el;
            B[i*num_cols + j] = accum;
        //  B[i][j] = accum;
        }
    }
}
```



- Each thread reads/writes an entire row of A/B;
- Perfect spatial locality for multi-core execution.
- **Uncoalesced access for GPU execution (terrible). What to do?**

GPU execution: in the same SIMD instruction the threads in a warp would read/write global memory with a stride of `num_cols`, hence uncoalesced access!

Coalesced Access by Transposition (GPU/Cuda)

GPU Pseudocode: outer loop parallel (kernel contains the inner sequential loop):

```
void gpuOptim ( float* A  // [num_rows][num_cols]
              , float* B  // [num_rows][num_cols]
              , const uint64_t num_rows
              , const uint64_t num_cols
) {
  float* A_tr = transpose(A, num_rows, num_cols);
  // A_tr , B_tr : [num_cols][num_rows]

  // Compute the transposed of B using the transposed of A
  for(uint64_t i = 0; i < num_rows; i++) { // parallel
    float accum = 0.0;
    for(uint64_t j = 0; j < num_cols; j++) { // seq
      float a_el  = A_tr[j*num_rows + i];
//    float a_el = A_tr[j][i]
      accum = sqrt(accum) + a_el*a_el;
      B_tr[j*num_rows + i] = accum;
//    B_tr[j][i] = accum;
    }
  }
  B = transpose(B_tr, num_cols, num_rows);
}
```

Coalesced Access by Transposition (GPU/Cuda)

GPU Pseudocode: outer loop parallel (kernel contains the inner sequential loop):

```
void gpuOptim ( float* A  // [num_rows][num_cols]
              , float* B  // [num_rows][num_cols]
              , const uint64_t num_rows
              , const uint64_t num_cols
) {
  float* A_tr = transpose(A, num_rows, num_cols);
  // A_tr, B_tr : [num_cols][num_rows]

  // Compute the transposed of B using the transposed of A
  for(uint64_t i = 0; i < num_rows; i++) { // parallel
    float accum = 0.0;
    for(uint64_t j = 0; j < num_cols; j++) { // seq
      float a_el  = A_tr[j*num_rows + i];
//    float a_el = A_tr[j][i]
      accum = sqrt(accum) + a_el*a_el;
      B_tr[j*num_rows + i] = accum;
//    B_tr[j][i] = accum;
    }
  }
  B = transpose(B_tr, num_cols, num_rows);
}
```

- The parallel loop now reads from the transpose of A and computes the transposed of B;
- **Excellent spatial locality for GPU** but **terrible for CPUs.**
- Significant speedup on GPUs, even though the optimized program performs $3\times$ more memory accesses than the original (two transpositions).
- This version can be optimized by a $\sim 2\times$ factor by using shared-memory as a staging buffer (not in this lecture).
  DO NOT FORGET DEMO!

Cuda Exercise 2: Coalesced Access by Transposition

```
{ // The host/CPU code that calls the kernel is already implemented
    uint32_t grid = (num_rows + B - 1) / B; // B is the Cuda block size
    callTransposeKer<ElTp, 32>( d_A,  d_Atr, num_rows, num_cols, true );
    transKernel<<<grid, B>>>(d_Atr, d_Btr, num_rows, num_cols );
    callTransposeKer<ElTp, 32>(d_Btr, d_B,   num_cols, num_rows, true );
}
// Cuda kernels: in file gpu-coalescing/kernels.cu.h
template<class ElTp> __global__ void // A_tr, B_tr : [num_cols][num_rows]
transKernel(ElTp* A_tr, ElTp* B_tr, uint32_t num_rows, uint32_t num_cols) {
    // Cuda Exercise: please implement me, e.g., by pattern-matching naiveKernel below
    // but changing the read and write from A and B to refer to A_tr and B_tr, respectivelly
    // (of course it needs to result in a semantically equivalent program that validates)
}
template<class ElTp> __global__ void // A, B : [num_rows][num_cols]
naiveKernel(ElTp* A, ElTp* B, uint32_t num_rows, uint32_t num_cols) {
    uint32_t gid = blockIdx.x * blockDim.x + threadIdx.x;
    if(gid >= num_rows) return;
    ElTp accum = 0;
    for(int j=0; j<num_cols; j++) {
        ElTp el_a  = A[ gid*num_cols + j ];
        accum = sqrt(accum) + el_a * el_a;
        B[ gid*num_cols + j ] = accum;
}   }
```

Flat representation of multi-dimensional arrays in memory

CPU vs GPU: Bird's Eye View

How do we measure/reason about Performance?

Programming Models Demonstrated on Simple Examples

- OpenMP
- Cuda

Five Case Studies

- LL$ threshing: Histogram-like computation
- Spatial Locality: Matrix Transposition
- Optimizing Spatial Locality by Transposition.
- L1$ and Register: Matrix-Matrix Multiplication
- L1$ and Register: Batch Matrix Multiplication under a Mask

Conclusions

Matrix-Matrix Multiplication (MMM): Golden Sequential

```
/**
 * Computes matrix multiplication C = A*B
 * for some (generic) numeric type ElTp.
 **/
template<class ElTp>
void goldenSeq( ElTp* A  // [heightA][widthA]
              , ElTp* B  // [ widthA][widthB]
              , ElTp* C  // [heightA][widthB]
              , int heightA
              , int widthB
              , int widthA
) {
  #pragma omp parallel for collapse(2)
  for(int i=0; i<heightA; i++) {     // parallel
    for(int j=0; j<widthB; j++) {    // parallel
      ElTp c = 0;
      for(int k=0; k<widthA; k++) { // sequential
        c += A[i*widthA +k] * B[k*widthB + j];
//      c += A[i][k] * B[k][j];
      }
      C[i*widthB + j] = c; // C[i][j] = c;
} } }
```

Matrix-Matrix Multiplication (MMM): Golden Sequential

```
/**
 * Computes matrix multiplication C = A*B
 * for some (generic) numeric type ElTp.
 **/
template<class ElTp>
void goldenSeq( ElTp* A  // [heightA][widthA]
              , ElTp* B  // [ widthA][widthB]
              , ElTp* C  // [heightA][widthB]
              , int heightA
              , int widthB
              , int widthA
) {
  #pragma omp parallel for collapse(2)
  for(int i=0; i<heightA; i++) {     // parallel
    for(int j=0; j<widthB; j++) {    // parallel
      ElTp c = 0;
      for(int k=0; k<widthA; k++) { // sequential
        c += A[i*widthA +k] * B[k*widthB + j];
//      c += A[i][k] * B[k][j];
      }
      C[i*widthB + j] = c; // C[i][j] = c;
} } }
```

- Does this run fast, would you think?
- **How/what can we improve?**

Matrix-Matrix Multiplication (MMM): Golden Sequential

```
/**
 * Computes matrix multiplication C = A*B
 * for some (generic) numeric type ElTp.
 **/
template<class ElTp>
void goldenSeq( ElTp* A  // [heightA][widthA]
              , ElTp* B  // [ widthA][widthB]
              , ElTp* C  // [heightA][widthB]
              , int heightA
              , int widthB
              , int widthA
) {
  #pragma omp parallel for collapse(2)
  for(int i=0; i<heightA; i++) {     // parallel
    for(int j=0; j<widthB; j++) {    // parallel
      ElTp c = 0;
      for(int k=0; k<widthA; k++) { // sequential
        c += A[i*widthA +k] * B[k*widthB + j];
//      c += A[i][k] * B[k][j];
      }
      C[i*widthB + j] = c; // C[i][j] = c;
} } }
```

- Does this run fast, would you think?
- **How/what can we improve?**
  1. Improve spatial locality by using the transposed of B.
     (But we can do better without it!)
  2. Temporal locality opportunity:

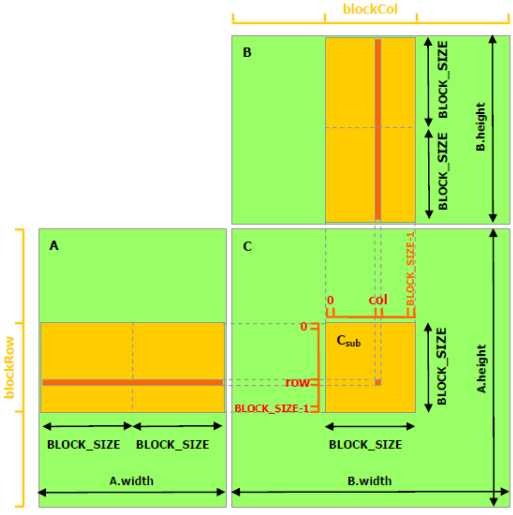
Matrix-Matrix Multiplication (MMM): Golden Sequential

```
/**
 * Computes matrix multiplication C = A*B
 * for some (generic) numeric type ElTp.
 **/
template<class ElTp>
void goldenSeq( ElTp* A  // [heightA][widthA]
              , ElTp* B  // [ widthA][widthB]
              , ElTp* C  // [heightA][widthB]
              , int heightA
              , int widthB
              , int widthA
) {
  #pragma omp parallel for collapse(2)
  for(int i=0; i<heightA; i++) {     // parallel
    for(int j=0; j<widthB; j++) {    // parallel
      ElTp c = 0;
      for(int k=0; k<widthA; k++) { // sequential
        c += A[i*widthA +k] * B[k*widthB + j];
//      c += A[i][k] * B[k][j];
      }
      C[i*widthB + j] = c; // C[i][j] = c;
} } }
```

- Does this run fast, would you think?
- **How/what can we improve?**
  1. Improve spatial locality by using the transposed of B.
     (But we can do better without it!)
  2. Temporal locality opportunity: the reads from A & B are invariant to the loops of indices `j` & `i`, respectivelly. Hence the same element is read multiple times.
- **Here we will focus on optimizaing temporal locality.**

MMM Tiling by Picture

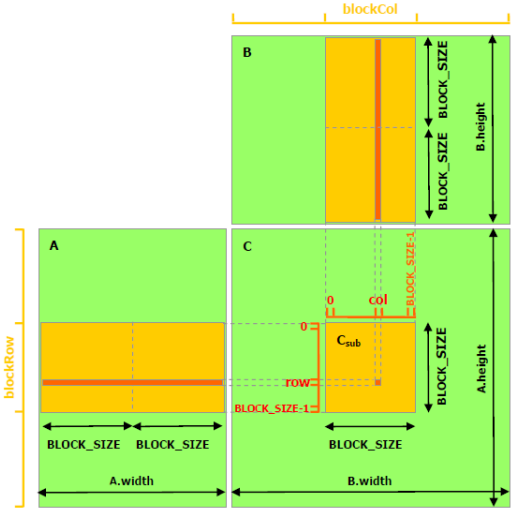
Picture courtesy of https://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault/Cours13/TD3.html



- **Main Idea is to block the computation:** each phase multiplies a block of A with a block of B.

MMM Tiling by Picture

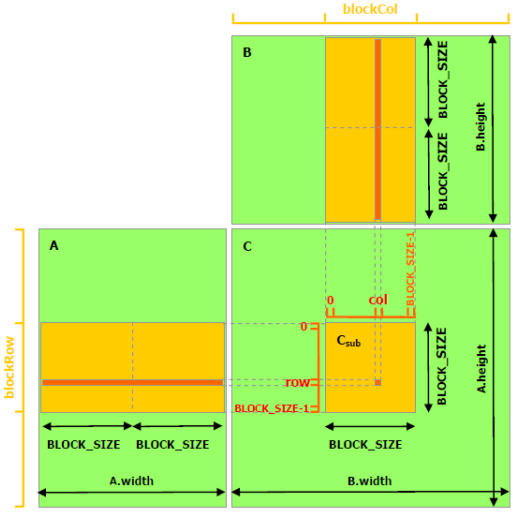
Picture courtesy of https://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault/Cours13/TD3.html



- **Main Idea is to block the computation:** each phase multiplies a block of A with a block of B.
- If the blocks from A and B fit in cache, then we have temporal reuse: the same row chunk of A is repeatedly multiplied with all the column chunks of B, and vice-versa.
- For multi-core CPU (OpenMP):
  - a thread performs the block-block multiplication;
  - best performance when blocking (tiling) is performed (recursively) at each cache level.

MMM Tiling by Picture

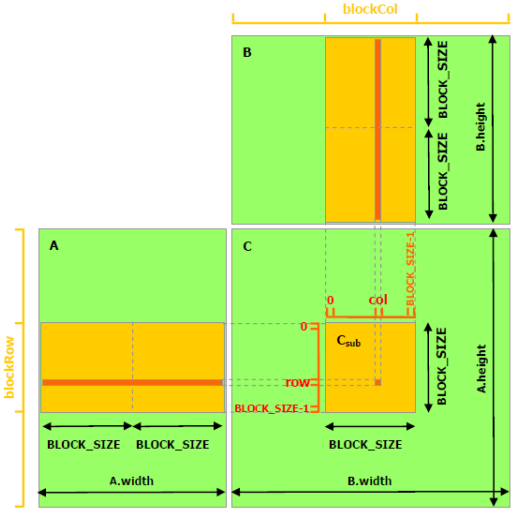
Picture courtesy of https://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault/Cours13/TD3.html



- **Main Idea is to block the computation:** each phase multiplies a block of A with a block of B.
- For GPU (Cuda):
  - one Cuda block performs the block-block mult;
  - blocks of A and B are collectively copied to shared memory, and reused from there;
  - each thread computes one element of C.

MMM Tiling by Picture

Picture courtesy of https://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault/Cours13/TD3.html



- **Main Idea is to block the computation:** each phase multiplies a block of A with a block of B.
- For GPU (Cuda):
  - one Cuda block performs the block-block mult;
  - blocks of A and B are collectively copied to shared memory, and reused from there;
  - each thread computes one element of C.
- For Cuda/GPU (but similar for multi-cores/OpenMP):
  1. **Block width and height need not be the same;**
  2. **Exploit both shared+register memory, i.e., the higher the block size the higher the degree of re-use:**
     - Use larger blocks: $(T \times R) \times T$;
     - Cuda block: $T \times T$;
     - Each thread computes in registers $R \times R$ elements of C.

OpenMP/CPU Pseudocode for MMM C = A * B

T and R are statically known constants, A: [heightA][widthA], B: [widthA][widthB], C: [heightA][widthB].

```
#pragma omp parallel for collapse(2)      // Cuda:
for(int iii=0; iii<heightA; iii+=T*R){    // Grid.y
  for(int jjj=0; jjj<widthB; jjj+=T*R){   // Grid.x
    for(int ii=iii; ii<iii+T*R; ii+=R) { // Block.y
      for(int jj=jjj; jj<jjj+T*R; jj+=R){// Block.x

        float css[R][R]; // per thread result
        for(int i_r=0; i_r<R; i_r++)
          for(int j_r=0; j_r<R; j_r++)
            css[i_r][j_r] = 0;

        for(int kk=0; kk<widthA; kk+=T) {
          // In Cuda: Aloc[T*R][T], Bloc[T][T*R]
          //     mapped in shared memory
          float Aloc[R][T], Bloc[T][R];

          // write 0 in Aloc/Bloc if out of the bounds in A/B
          copySliceGlb2Sh(A[iii: iii+T][kk: kk+T], Aloc);
          copySliceGlb2Sh(B[kk: kk+T][jjj: jjj+R], Bloc);

          // main computation with A/B remapped
          // to Aloc/Bloc (see next column)
          #pragma unroll
          for(int k_r=0; k_r<T; k_r++){
            #pragma unroll
            for(int i_r=0; i_r<R; i_r++){
              #pragma unroll
              for(int j_r=0;j_r<R;j_r++){
                css[i_r][j_r] +=
                    Aloc[i_r][k_r] *
                    Bloc[k_r][j_r];
} } } } // end loops j_r, i_r, k_r, kk

        // update global C
        for(int i_r=0; i_r<R; i_r++) {
          const int i = ii+i_r;
          for(int j_r=0; j_r<R; j_r++) {
            const int j = jj+j_r;
            if(i<heightA && j<widthB)
              C[i*widthB+j] =
                    css[i_r][j_r];
          } // end loop j_r
        } // end loop i_r
} } } } // end loops jj, ii, jjj, iii
```

Think Like a Compiler: Dependency-Analysis on Arrays

Loop Stripmining: is always safe to perform!

```
for(int i=0; i<N; i++)
  body(i);
```

≡

```
for(int ii=0; ii<N; ii+=T)
  for(int i=ii; i<min(ii+T,N);
                i++ )
    body(i);
```

≡

```
for(int ii=0; ii<N; ii+=T)
  for(int i_r=0; i_r<T;i_r++){
    int i = ii + i_r;
    if(i < N) body(i);        }
```

Think Like a Compiler: Dependency-Analysis on Arrays

Loop Stripmining: is always safe to perform!

```
for(int i=0; i<N; i++)
  body(i);
```

≡

```
for(int ii=0; ii<N; ii+=T)
  for(int i=ii; i<min(ii+T,N);
                i++ )
    body(i);
```

≡

```
for(int ii=0; ii<N; ii+=T)
  for(int i_r=0; i_r<T;i_r++){
    int i = ii + i_r;
    if(i < N) body(i);        }
```

Loop Interchange in a perfect nest: always safe to interchange a parallel loop inwards.

```
for(int i=0; i<M; i++)
  for (int j=0; j<N; j++)
    body(i, j);
```

⟹

```
for (int j=0; j<N; j++)
  for(int i=0; i<M; i++)
    body(i, j);
```

Think Like a Compiler: Dependency-Analysis on Arrays

Loop Stripmining: is always safe to perform!

```
for(int i=0; i<N; i++)
  body(i);
```

≡

```
for(int ii=0; ii<N; ii+=T)
  for(int i=ii; i<min(ii+T,N);
                i++ )
    body(i);
```

≡

```
for(int ii=0; ii<N; ii+=T)
  for(int i_r=0; i_r<T; i_r++){
    int i = ii + i_r;
    if(i < N) body(i);        }
```

Loop Interchange in a perfect nest: always safe to interchange a parallel loop inwards.

```
for(int i=0; i<M; i++)
  for (int j=0; j<N; j++)
    body(i, j);
```

⟹

```
for (int j=0; j<N; j++)
  for(int i=0; i<M; i++)
    body(i, j);
```

Block tiling: stripmine both loops & interchange the outer-loop tile inwards (if safe).

```
for(int i=0; i<M; i++)
  for (int j=0; j<N; j++)
    body(i, j);
```

⟹

```
for(int ii = 0; ii < M; ii += T1)
  for(int jj = 0; jj < N; jj += T2)
    for(int i=ii; i < min(ii+T1,M); i++)
      for(int j=jj; j < min(jj+T2,N); j++)
        body(i, j);
```

Think Like a Compiler: Dependency-Analysis on Arrays (continuation)

Loop Distribution: it is always safe to distribute a parallel loop across its statements as long as you expand variables declared locally in the loop body with an extra dimension of loop-count size.

```
for(int i=0; i<N; i++) {
    float acc = i*i;
    body(i, acc);
}
```

$\Longrightarrow$

```
float acc[N];
for(int i=0; i<N; i++)
    acc[i] = i*i;
for(int i=0; i<N; i++)
    body(i, acc[i]);
```

Think Like a Compiler: Dependency-Analysis on Arrays (continuation)

Loop Distribution: it is always safe to distribute a parallel loop across its statements as long as you expand variables declared locally in the loop body with an extra dimension of loop-count size.

```
for(int i=0; i<N; i++) {
    float acc = i*i;
    body(i, acc);
}
```

$\Longrightarrow$

```
float acc[N];
for(int i=0; i<N; i++)
    acc[i] = i*i;
for(int i=0; i<N; i++)
    body(i, acc[i]);
```

Remapping a read-only array at a program point: compute the read set of the array following the program point (same scope) + copy the read set to a smaller array + remap the following computation to only use the smaller array.

Think Like a Compiler: Dependency-Analysis on Arrays (continuation)

Loop Distribution: it is always safe to distribute a parallel loop across its statements as long as you expand variables declared locally in the loop body with an extra dimension of loop-count size.

```
for(int i=0; i<N; i++) {
    float acc = i*i;
    body(i, acc);
}
```

$\Longrightarrow$

```
float acc[N];
for(int i=0; i<N; i++)
    acc[i] = i*i;
for(int i=0; i<N; i++)
    body(i, acc[i]);
```

Remapping a read-only array at a program point: compute the read set of the array following the program point (same scope) + copy the read set to a smaller array + remap the following computation to only use the smaller array.

Loop unrolling: may benefit ILP optimizations, including scalarization of arrays.

```
int acc[i];
#pragma unroll
for(int i=0; i<8; i++)
    acc[i] = f(i);
...
#pragma unroll
for(int i=0; i<8; i++)
    C[g(i)] = acc[i];
```

$\Longrightarrow$

```
float acc0 = f(0);
...
float acc7 = f(7);
.........
C[g(0)] = acc0;
...
C[g(7)] = acc7;
```

MMM: Think-Like-a-Compiler Optimization Recipe

Step 1.1: Tile loops of indices `i` and `j` twice by `T*R` and then `R`, i.e., stripmine each twice then interchange the tiles inside.

Step1.2: Also stripmine once the loop of index `k` by a tile `T`.

For simplicity assume all matrix dimensions are multiples of `T·R`.

```
// A: [heightA][widthA]
// B: [ widthA][widthB]
// C: [heightA][widthB]
for(int i=0; i<heightA; i++){   // parallel
  for(int j=0; j<widthB; j++){ // parallel
    float c = 0;
    for(int k=0; k<widthA; k++) { // seq
      c += A[i][k] * B[k][j];
    }
    C[i*widthB + j] = c; //C[i][j] = c;
} }
```

MMM: Think-Like-a-Compiler Optimization Recipe

Step 1.1: Tile loops of indices `i` and `j` twice by `T*R` and then `R`, i.e., stripmine each twice then interchange the tiles inside.

Step1.2: Also stripmine once the loop of index `k` by a tile `T`.

For simplicity assume all matrix dimensions are multiples of `T·R`.

```
// A: [heightA][widthA]
// B: [ widthA][widthB]
// C: [heightA][widthB]
for(int i=0; i<heightA; i++){   // parallel
  for(int j=0; j<widthB; j++){ // parallel
    float c = 0;
    for(int k=0; k<widthA; k++) { // seq
      c += A[i][k] * B[k][j];
    }
    C[i*widthB + j] = c; //C[i][j] = c;
} }
```

```
for(int iii=0; iii<heightA; iii+=T*R) {    // par
  for(int jjj=0; jjj<widthB; jjj+=T*R) {   // par
    for(int ii=iii; ii<iii+T*R; ii+=R)  { // par
      for(int jj=jjj; jj<jjj+T*R; jj+=R) {// par
        for(int i_r = 0; i_r<R; i_r++)  { // par
          for(int j_r = 0; j_r<R; j_r++) {// par

            float c = 0;
            for(int kk=0; kk<widthA; kk+=T){ // seq
              for(int k_r=0; k_r<T; k_r++) { // seq
                c +=  A[ii+i_r][kk+k_r] *
                      B[kk+k_r][jj+j_r];
              }
            }
            C[ii+i_r][jj+j_r] = c;
} } } } } }
```

MMM Think-Like-a-Compiler: Cuda/GPU Interpretation

```
for(int iii=0; iii<heightA; iii+=T*R) {    // Block.y
  for(int jjj=0; jjj<widthB; jjj+=T*R) {   // Block.x
    for(int ii=iii; ii<iii+T*R; ii+=R){    // threadIdx.y
      for(int jj=jjj; jj<jjj+T*R; jj+=R){  // threadIdx.x
        for(int i_r = 0; i_r<R; i_r++)  {  // seq
          for(int j_r = 0; j_r<R; j_r++) {// seq

            float c = 0;
            for(int kk=0; kk<widthA; kk+=T){ // seq
              for(int k_r=0; k_r<T; k_r++) { // seq
                c +=  A[ii+i_r][kk+k_r] *
                      B[kk+k_r][jj+j_r];
              }
            }
            C[ii+i_r][jj+j_r] = c;
} } } } } }
```

Cuda Block: $T \times T$

Grid: $\lceil \frac{\text{heightA}}{T \cdot R} \rceil \times \lceil \frac{\text{widthB}}{T \cdot R} \rceil$

```
iii= blockIdx.y * T * R
jjj= blockIdx.x * T * R
ii= iii + threadIdx.y * R
jj= jjj + threadIdx.x * R
```

Each thread computes a $R \times R$ tile of the result C.

MMM Think-Like-a-Compiler: Step 2 of Optimization Recipe

Step 2: Distribute and interchange in innermost position the loops i_r & j_r.

```
for(int iii=0; iii<heightA; iii+=T*R) {    // par
  for(int jjj=0; jjj<widthB; jjj+=T*R) {   // par
    for(int ii=iii; ii<iii+T*R; ii+=R){    // par
      for(int jj=jjj; jj<jjj+T*R; jj+=R){  // par
        for(int i_r = 0; i_r <R; i_r++) {  // par
          for(int j_r = 0; j_r <R; j_r++) {// par

            float c = 0;
            for(int kk=0; kk<widthA; kk+=T){  // seq
              for(int k_r=0; k_r<T; k_r++) {  // seq
                c +=  A[ii+i_r][kk+k_r] *
                      B[kk+k_r][jj+j_r];
              }
            }
            C[ii+i_r][jj+j_r] = c;
} } } } } }
```

MMM Think-Like-a-Compiler: Step 2 of Optimization Recipe

Step 2: Distribute and interchange in innermost position the loops i_r & j_r.

```
for(int iii=0; iii<heightA; iii+=T*R) {    // par
  for(int jjj=0; jjj<widthB; jjj+=T*R) {   // par
    for(int ii=iii; ii<iii+T*R; ii+=R){    // par
      for(int jj=jjj; jj<jjj+T*R; jj+=R){  // par
        for(int i_r = 0; i_r <R; i_r++) {  // par
          for(int j_r = 0; j_r <R; j_r++) {// par

            float c = 0;
            for(int kk=0; kk<widthA; kk+=T){  // seq
              for(int k_r=0; k_r<T; k_r++) { // seq
                c +=  A[ii+i_r][kk+k_r] *
                      B[kk+k_r][jj+j_r];
              }
            }
            C[ii+i_r][jj+j_r] = c;
} } } } } }
```

```
for iii, jjj, ii, jj {
  float c[R][R]; // array expansion!
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      c[i_r][j_r] = 0;

  for(int kk=0; kk<widthA; kk+=T){
    for(int k_r=0; k_r<T; k_r++) {
      for(int i_r=0; i_r<R; i_r++) {
        for(int j_r=0; j_r<R; j_r++) {
          c[i_r][j_r] +=
                A[ii+i_r][kk+k_r] *
                B[kk+k_r][jj+j_r];
  } } } }
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      C[ii+i_r][jj+j_r] = c[i_r][j_r];
}
```

MMM Think-Like-a-Compiler: Step 3 of Optimization Recipe

Step 3: Remap arrays A and B just inside the loop of index kk.

```
for iii, jjj, ii, jj {
  float c[R][R];
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      c[i_r][j_r] = 0;
  for(int kk=0; kk<widthA; kk+=T){
    //What slices of A and B are used in this scope?
    for(int k_r=0; k_r<T; k_r++) {
      for(int i_r=0; i_r<R; i_r++) {
        for(int j_r=0; j_r<R; j_r++) {
          c[i_r][j_r] +=  A[ii+i_r][kk+k_r] *
                          B[kk+k_r][jj+j_r];
  } } } }
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      C[ii+i_r][jj+j_r] = c[i_r][j_r];
}
```

For CPU:

MMM Think-Like-a-Compiler: Step 3 of Optimization Recipe

Step 3: Remap arrays A and B just inside the loop of index kk.

```
for iii, jjj, ii, jj {
  float c[R][R];
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      c[i_r][j_r] = 0;
  for(int kk=0; kk<widthA; kk+=T){
    //What slices of A and B are used in this scope?
    for(int k_r=0; k_r<T; k_r++) {
      for(int i_r=0; i_r<R; i_r++) {
        for(int j_r=0; j_r<R; j_r++) {
          c[i_r][j_r] +=  A[ii+i_r][kk+k_r] *
                          B[kk+k_r][jj+j_r];
  } } } }
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      C[ii+i_r][jj+j_r] = c[i_r][j_r];
}
```

For CPU: A[ii:ii+R][kk:kk+T]
and B[kk:kk+T][jj:jj+R]

MMM Think-Like-a-Compiler: Step 3 of Optimization Recipe

Step 3: Remap arrays A and B just inside the loop of index kk.

```
for iii, jjj, ii, jj {
  float c[R][R];
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      c[i_r][j_r] = 0;
  for(int kk=0; kk<widthA; kk+=T){
    //What slices of A and B are used in this scope?
    for(int k_r=0; k_r<T; k_r++) {
      for(int i_r=0; i_r<R; i_r++) {
        for(int j_r=0; j_r<R; j_r++) {
          c[i_r][j_r] +=  A[ii+i_r][kk+k_r] *
                          B[kk+k_r][jj+j_r];
  } } } }
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      C[ii+i_r][jj+j_r] = c[i_r][j_r];
}
```

For CPU: A[ii:ii+R][kk:kk+T]
and B[kk:kk+T][jj:jj+R]

```
for iii, jjj, ii, jj {
  ...
  float Aloc[R][T];
  float Bloc[T][R];
  for(int kk=0; kk<widthA; kk+=T){
    for(int i_r=0; i_r<R; i_r++) {
      for(int k_r=0; k_r<T; k_r++) {
        const int i= ii+i_r, k= kk+k_r;
        Aloc[i_r][k_r] =
          (i<heightA && k<widthA) ?
            A[i][k] : 0;
    } } // ... similar for Bloc
    for(int k_r=0; k_r<T; k_r++) {
      for(int i_r=0; i_r<R; i_r++) {
        for(int j_r=0; j_r<R; j_r++) {
          c[i_r][j_r] +=  Aloc[i_r][k_r] *
                          Bloc[k_r][j_r];
  } } } }
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      C[ii+i_r][jj+j_r] = c[i_r][j_r];
}
```

MMM Think-Like-a-Compiler: Cuda Exercise 3

What slices of A and B are used inside the loop of index `kk` by the whole Cuda block?
(i.e., eliminate ii and jj as well; you may use iii and jjj inside the slice notation).

```
for iii, jjj {
for(int ii=iii; ii<iii+T*R; ii+=R) {
for(int jj=jjj; jj<jjj+T*R; jj+=R) {
  float c[R][R];
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      c[i_r][j_r] = 0;
  for(int kk=0; kk<widthA; kk+=T){
    //collective copy from global to shared memory
    //of the slices of A and B used in this scope
    for(int k_r=0; k_r<T; k_r++) {
      for(int i_r=0; i_r<R; i_r++) {
        for(int j_r=0; j_r<R; j_r++) {
          c[i_r][j_r]+= A[ii+i_r][kk+k_r]*
                        B[kk+k_r][jj+j_r];
  } } } }
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      C[ii+i_r][jj+j_r] = c[i_r][j_r];
} } }
```

Task 3.1: Which is the maximal slice of `A[ii+i_r][kk+k_r]` accessed inisde loop kk and expressed in terms of `iii` and kk, i.e., eliminate `ii`, `i_r`, `k_r`?

MMM Think-Like-a-Compiler: Cuda Exercise 3

What slices of A and B are used inside the loop of index `kk` by the whole Cuda block?
(i.e., eliminate ii and jj as well; you may use iii and jjj inside the slice notation).

```
for iii, jjj {
for(int ii=iii; ii<iii+T*R; ii+=R) {
for(int jj=jjj; jj<jjj+T*R; jj+=R) {
  float c[R][R];
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      c[i_r][j_r] = 0;
  for(int kk=0; kk<widthA; kk+=T){
    //collective copy from global to shared memory
    //of the slices of A and B used in this scope
    for(int k_r=0; k_r<T; k_r++) {
      for(int i_r=0; i_r<R; i_r++) {
        for(int j_r=0; j_r<R; j_r++) {
          c[i_r][j_r]+= A[ii+i_r][kk+k_r]*
                        B[kk+k_r][jj+j_r];
  } } } }
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      C[ii+i_r][jj+j_r] = c[i_r][j_r];
} } }
```

Task 3.1: Which is the maximal slice of `A[ii+i_r][kk+k_r]` accessed inisde loop kk and expressed in terms of `iii` and kk, i.e., eliminate `ii`, `i_r`, `k_r`?

`A[ii+i_r][kk+k_r]` $\in$
`A[iii:  iii+T*R][kk:  kk+T]`

Similar for B!

MMM Think-Like-a-Compiler: Cuda Exercise 3

What slices of A and B are used inside the loop of index `kk` by the whole Cuda block?
(i.e., eliminate ii and jj as well; you may use iii and jjj inside the slice notation).

```
for iii, jjj {
for(int ii=iii; ii<iii+T*R; ii+=R) {
for(int jj=jjj; jj<jjj+T*R; jj+=R) {
  float c[R][R];
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      c[i_r][j_r] = 0;
  for(int kk=0; kk<widthA; kk+=T){
    //collective copy from global to shared memory
    //of the slices of A and B used in this scope
    for(int k_r=0; k_r<T; k_r++) {
      for(int i_r=0; i_r<R; i_r++) {
        for(int j_r=0; j_r<R; j_r++) {
          c[i_r][j_r]+= A[ii+i_r][kk+k_r]*
                        B[kk+k_r][jj+j_r];
  } } } }
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      C[ii+i_r][jj+j_r] = c[i_r][j_r];
} } }
```

Task 3.1: Which is the maximal slice of `A[ii+i_r][kk+k_r]` accessed inisde loop kk and expressed in terms of `iii` and kk, i.e., eliminate `ii`, `i_r`, `k_r`?

`A[ii+i_r][kk+k_r]` $\in$
`A[iii: iii+T*R][kk: kk+T]`

Similar for B!

Cuda Block Size: T $\times$ T

```
iii= blockIdx.y * T * R
jjj= blockIdx.x * T * R
ii= iii + threadIdx.y * R
jj= jjj + threadIdx.x * R
```

Each thread computes a R $\times$ R tile of the result C.

MMM Think-Like-a-Compiler: Cuda Exercise 3

What slices of A and B are used inside the loop of index `kk` by the whole Cuda block?

(i.e., eliminate ii and jj as well; you may use iii and jjj inside the slice notation).

```
for iii, jjj {
for(int ii=iii; ii<iii+T*R; ii+=R) {
for(int jj=jjj; jj<jjj+T*R; jj+=R) {
  __shared__ float Aloc[T*R][T], Bloc[T*R][T];
  float c[R][R];
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      c[i_r][j_r] = 0;
  for(int kk=0; kk<widthA; kk+=T){
    //collective copy from global to shared memory
    //of the slices of A and B used in this scope
    for(int k_r=0; k_r<T; k_r++) {
      for(int i_r=0; i_r<R; i_r++) {
        for(int j_r=0; j_r<R; j_r++) {
          c[i_r][j_r] += A[ii+i_r][kk+k_r] *
                        B[kk+k_r][jj+j_r];
  } } } }
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      C[ii+i_r][jj+j_r] = c[i_r][j_r];
} } }
```

Task 3.2: Insert the Cuda code that collectively copies—with all the $T \times T$ threads of the Cuda block the slices—just inside loop of index kk, the corresponding slice from A and B (global mem) into shared-memory arrays `Aloc[T*R][T]` and `Bloc[T][T*R]`.

MMM Think-Like-a-Compiler: Cuda Exercise 3

What slices of A and B are used inside the loop of index kk by the whole Cuda block?

(i.e., eliminate ii and jj as well; you may use iii and jjj inside the slice notation).

```
for iii, jjj {
for(int ii=iii; ii<iii+T*R; ii+=R) {
for(int jj=jjj; jj<jjj+T*R; jj+=R) {
  __shared__ float Aloc[T*R][T], Bloc[T*R][T];
  float c[R][R];
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      c[i_r][j_r] = 0;
  for(int kk=0; kk<widthA; kk+=T){
    //collective copy from global to shared memory
    //of the slices of A and B used in this scope
    for(int k_r=0; k_r<T; k_r++) {
      for(int i_r=0; i_r<R; i_r++) {
        for(int j_r=0; j_r<R; j_r++) {
          c[i_r][j_r] += A[ii+i_r][kk+k_r] *
                        B[kk+k_r][jj+j_r];
  } } } }
  for(int i_r=0; i_r<R; i_r++)
    for(int j_r=0; j_r<R; j_r++)
      C[ii+i_r][jj+j_r] = c[i_r][j_r];
} } }
```

Task 3.2: Insert the Cuda code that collectively copies—with all the $T \times T$ threads of the Cuda block the slices—just inside loop of index kk, the corresponding slice from A and B (global mem) into shared-memory arrays `Aloc[T*R][T]` and `Bloc[T][T*R]`.

Task 3.3: Change the accesses to A and B inside the computation of c to refer to `Aloc` and `Bloc` instead!

Search for "Exercise" in file `mmm/kernels.cu.h` & insert your code.

MMM in Cuda and OpenMP: Performance Results

Show MMM Performance Results

Flat representation of multi-dimensional arrays in memory

CPU vs GPU: Bird's Eye View

How do we measure/reason about Performance?

Programming Models Demonstrated on Simple Examples
- OpenMP
- Cuda

Five Case Studies
- LL$ threshing: Histogram-like computation
- Spatial Locality: Matrix Transposition
- Optimizing Spatial Locality by Transposition.
- L1$ and Register: Matrix-Matrix Multiplication
- L1$ and Register: Batch Matrix Multiplication under a Mask

Conclusions

Batch Matrix Multiplication under a Mask: Golden Sequential

[2] F. Gieseke, S. Rosca, T. Henriksen, J. Verbesselt, C. Oancea, "Massively-Parallel Change Detection for Satellite Time Series Data with Missing Values", ICDE'20.

```
void goldenSeq( float* A // [K][N]
              , float* B // [N][K]
              , char*  X // [M][N]
              , float* Y // [M][K][K]
              , int M, int K, int N   ) {
  #pragma omp parallel for schedule(static)
  for(int i=0; i<M; i++) {  // parallel
    for(int j1=0; j1<K; j1++)  { // par
      for(int j2=0; j2<K; j2++){ // par
        float acc = 0.0;
        for(int q=0; q<N; q++) { // seq
          float a = A[j1][q];
          float b = B[q][j2];
          float v = ( X[i][q] != 0 ) ?
                    1 : 0;
          acc += a * b * v;
        }
        Y[i][j1][j2] = acc;
} } } }
```

Q: **What hints that temporal locality can be optimized?**

Batch Matrix Multiplication under a Mask: Golden Sequential

[2] F. Gieseke, S. Rosca, T. Henriksen, J. Verbesselt, C. Oancea, "Massively-Parallel Change Detection for Satellite Time Series Data with Missing Values", ICDE'20.

```
void goldenSeq( float* A // [K][N]
              , float* B // [N][K]
              , char*  X // [M][N]
              , float* Y // [M][K][K]
              , int M, int K, int N   ) {
  #pragma omp parallel for schedule(static)
  for(int i=0; i<M; i++) {  // parallel
    for(int j1=0; j1<K; j1++)  { // par
      for(int j2=0; j2<K; j2++){ // par
        float acc = 0.0;
        for(int q=0; q<N; q++) { // seq
          float a = A[j1][q];
          float b = B[q][j2];
          float v = ( X[i][q] != 0 ) ?
                      1 : 0;
          acc += a * b * v;
        }
        Y[i][j1][j2] = acc;
} } } }
```

Q: **What hints that temporal locality can be optimized?**

A: the indexing of arrays A, B, X is invariant to 2 parallel dimensions

- K is typically small ($K \leq 8$).

Q: **What is the optimization recipe?**

Batch Matrix Multiplication under a Mask: Golden Sequential

[2] F. Gieseke, S. Rosca, T. Henriksen, J. Verbesselt, C. Oancea, "Massively-Parallel Change Detection for Satellite Time Series Data with Missing Values", ICDE'20.

```
void goldenSeq( float* A // [K][N]
              , float* B // [N][K]
              , char*  X // [M][N]
              , float* Y // [M][K][K]
              , int M, int K, int N   ) {
  #pragma omp parallel for schedule(static)
  for(int i=0; i<M; i++) {  // parallel
    for(int j1=0; j1<K; j1++)  { // par
      for(int j2=0; j2<K; j2++){ // par
        float acc = 0.0;
        for(int q=0; q<N; q++) { // seq
          float a = A[j1][q];
          float b = B[q][j2];
          float v = ( X[i][q] != 0 ) ?
                      1 : 0;
          acc += a * b * v;
        }
        Y[i][j1][j2] = acc;
} } } }
```

Q: **What hints that temporal locality can be optimized?**

A: the indexing of arrays A, B, X is invariant to 2 parallel dimensions

- K is typically small ($K \leq 8$).

Q: **What is the optimization recipe?**

A: Strip-mine the outermost loop (e.g., tile$=4$ for CPU & 31 for GPU) and move the tile innermost.

1. `a` and `b` do not depend on `i` $\Rightarrow$ will be reused from registers.
2. `X` does not depend on `j1` and `j2` $\Rightarrow$ reused across those loops from L1.

Batch MMM under Mask: Applying the Optimization Recipe (OpenMP)

```
// A:[K][N], B:[N][K], X:[M][N], Y:[M][K][K]
#pragma omp parallel for schedule(static)
for(int ii=0; ii<M; ii+=T) {  // parallel
  for(int j1=0; j1<K; j1++)  {
    for(int j2=0; j2<K; j2++){
      float acc[T]; // array expansion
      for(int i_r=0; i_r<T; i_r++)
        acc[i_r] = 0.0;

      for(int q=0; q<N; q++) { // seq
        float a = A[j1][q];
        float b = B[q][j2];
        for(int i=ii; i<min(ii+T,M); i++){
          float v = (X[i][q]!=0)? 1 : 0;
          acc[i-ii] += a * b * v;
        }
      }
      for(int i_r=0; i_r<T; i_r++)
        if(ii+i_r < M)
          Y[ii+i_r][j1][j2] = acc[i_r];
} } }
```

Show Performance for CPU and GPU

Batch MMM Think-Like-a-Compiler: Cuda Improvements

```
// A:[K][N], B:[N][K], X:[M][N], Y:[M][K][K]
for(int ii=0; ii<M; ii+=T) {  // blockIdx.x
  for(int j1=0; j1<K; j1++)  {//threadIdx.y
    for(int j2=0; j2<K; j2++){//threadIdx.x
      float acc[T]; // array expansion
      for(int i_r=0; i_r<T; i_r++)
        acc[i_r] = 0.0;

      for(int q=0; q<N; q++) { // seq
        float a = A[j1][q];
        float b = B[q][j2];
        for(int i=ii; i<min(ii+T,M); i++){
          float v = (X[i][q]!=0)? 1 : 0;
          acc[i-ii] += a * b * v;
        }
      }
      for(int i_r=0; i_r<T; i_r++)
        if(ii+i_r < M)
          Y[ii+i_r][j1][j2] = acc[i_r];
} } }
```

Cuda Grid: $\lceil \frac{M}{T} \rceil$, Cuda Block: $K \times K$

`ii= blockIdx.x * T`

`j1= threadIdx.y`

`j2= threadIdx.x`

Each thread computes T elements.

Enhanced Optimization Recipe for Cuda:

1. Currently `X[i][q]` is in global memory (slow), we would like to reuse it from shared memory.
2. The slice of X read in loop of index `i` is:

Batch MMM Think-Like-a-Compiler: Cuda Improvements

```
// A:[K][N], B:[N][K], X:[M][N], Y:[M][K][K]
for(int ii=0; ii<M; ii+=T) {  // blockIdx.x
  for(int j1=0; j1<K; j1++)  {//threadIdx.y
    for(int j2=0; j2<K; j2++){//threadIdx.x
      float acc[T]; // array expansion
      for(int i_r=0; i_r<T; i_r++)
        acc[i_r] = 0.0;

      for(int q=0; q<N; q++) { // seq
        float a = A[j1][q];
        float b = B[q][j2];
        for(int i=ii; i<min(ii+T,M); i++){
          float v = (X[i][q]!=0)? 1 : 0;
          acc[i-ii] += a * b * v;
        }
      }
      for(int i_r=0; i_r<T; i_r++)
        if(ii+i_r < M)
          Y[ii+i_r][j1][j2] = acc[i_r];
} } }
```

Cuda Grid: $\lceil \frac{M}{T} \rceil$, Cuda Block: $K \times K$

`ii= blockIdx.x * T`

`j1= threadIdx.y`

`j2= threadIdx.x`

Each thread computes T elements.

Enhanced Optimization Recipe for Cuda:

1. Currently `X[i][q]` is in global memory (slow), we would like to reuse it from shared memory.
2. The slice of X read in loop of index `i` is: `X[ii:ii+T][q]`, which fits in a shared-memory buffer of size T.
3. The plan is to copy with the first T threads of the Cuda block the T elements of `X[ii:ii+T][q]`, then barier, then execute loop `i`, then again barrier.
4. This however would result in uncoalesced access to X, hence we need to work with `X_tr`, the transpose of X.

Batch MMM Think-Like-a-Compiler: Cuda Pseudocode & Exercise 4

```
// A:[K][N], B:[N][K], X_tr:[N][M], Y:[M][K][K]
for(int ii=0; ii<M; ii+=T) { // ii = blockIdx.x*T
  for(int j1=0; j1<K; j1++)  {// j1 = threadIdx.y
    for(int j2=0; j2<K; j2++){// j2 = threadIdx.x
      __shared__ float Xsh_tr[T]; float acc[T];
      for(int i_r=0; i_r<T; i_r++)
        acc[i_r] = 0.0;
      for(int q=0; q<N; q++) { // seq
        float ab = A[j1][q] * B[q][j2];

        int tid = threadIdx.y * K + threadIdx.x;
        int i = ii + tid;
        char x = (tid<T && i<M)? X_tr[q][i] : 0;
        Xsh_tr[tid] = x;
        #pragma unroll
        for(int i_r=0; i<T; i_r++){
          float v = (Xsh_tr[i_r]!=0)? 1 : 0;
          acc[i_r] += ab * v;
      } }
      for(int i_r=0; i_r<T; i_r++)
        if(ii+i_r < M)
          Y[ii+i_r][j1][j2] = acc[i_r];
} } }
```

Cuda Grid: $\lceil \frac{M}{T} \rceil$, Cuda Block: $K \times K$

Enhanced Optimization Recipe for Cuda (previous slide):

1. Currently `X[i][q]` is in global memory (slow), we would like to reuse it from shared memory.
2. The slice of X read in loop of index `i` is: `X[ii:ii+T][q]`, which fits in a shared-memory buffer of size `T`.
3. The plan is to copy with the first `T` threads of the Cuda block the `T` elements of `X[ii:ii+T][q]`, then barier, then execute loop `i`, then again barrier.
4. This however would result in uncoalesced access to X, hence we need to work with `X_tr`, the transpose of X.

Cuda Exercise 4: With the help of the pseudocode on the left, implement in folder `batch-mmm`, file `kernels.cu.h`, kernel `bmmmTiledKer`. Remember to flatten the indices to all arrays.

Summary

What have we taken a glimpse at today?

- **Two different programming models:**
  - OpenMP for multi-core CPU
  - Cuda for (Nvidia) GPUs

- **Five case studies showcasing techniques to optimize locality at various levels**
  - LL$ threshing: Histogram-Like Computation
  - spatial locality: matrix transposition & contribed program
  - L1$ & registers: matrix multiplication and batch matrix multiplication under mask;

- **Reasoned in two ways:**
  - as humans do: “a picture makes for 100 words”;
  - as compiler do: loop strip-mining, interchange, distribution.

- **Demostrated significant performance gains!**