

OpenCL Day 2: Loop Reasoning, Optimizing Temporal and Spatial Locality

Cosmin Oancea and Troels Henriksen

January 30, 2019

Content of Day2 Lecture:

Dependence Analysis

- Problem Statement, Dependency Definition

- Dependency Matrix, Loop Parallelism and Interchange

- Loop Distribution, Array Expansion/Privatization

- Exercise: Putting it all together

Optimizing Spatial Locality (Coalesced Memory)

- Motivating Example on A Contrived Program

- Uncoalesced Matrix Transposition Kernel

- Coding Exercise: Entirely-Coalesced Transposition

- Program Optimized By Transposition + Coding Exercise

- Fully Fused Program + Coding Exercise

Optimizing Temporal Locality

- Naive Matrix-Matrix Multiplication

- Block-Tiled Matrix Multiplication + Coding Exercise

- Register-Tiled Matrix Multiplication + Coding Exercise

Problem Statement

Three Loop Examples

```
DO i = 1, N
  DO j = 1, N
    A[j,i] = A[j,i] ..
  ENDDO
ENDDO
```

```
DO i = 2, N
  DO j = 2, N
    A[j,i] = A[j-1,i-1]...
    B[j,i] = B[j-1,i]...
  ENDDO ENDDO
```

```
DO i = 2, N
  DO j = 1, N
    A[i,j] = ...
    A[i-1,j+1]
  ENDDO ENDDO
```

Iterations are ordered *lexicographically*, in the order they occur in the sequential execution: $\vec{k}=(i=2, j=4) < \vec{l}=(i=3, j=3)$.

- ▶ Which of the three loop nests is amenable to parallelization?
- ▶ Loop interchange is one of the most simple and useful code transformations, e.g., used to enhance locality of reference, and even to “create” parallelism.
- ▶ In which loop nest is it safe to interchange the loops?

Definition of a Dependency

Load-Store Classification of Dependencies

True Dependency(RAW)	Anti Dependency(WAR)	Output dependency(WAW)
S1 X = ..	S1 .. = X	S1 X = ...
S2 .. = X	S2 X = ..	S2 X = ...

Def. Loop Dependence: There is a dependence from statement $S1$ to $S2$ in a loop nest *iff* there exist iterations $\vec{k} \leq \vec{l}$ and an execution path from statement $S1$ to statement $S2$ **such that:**

1. $S1$ accesses memory location M on iteration \vec{k} , and
2. $S2$ accesses memory location M on iteration \vec{l} , and
3. one of these accesses is a write.

We say that $S1$ is the source and $S2$ is the sink of the dependence, because $S1$ executes before $S2$ in the sequential program execution. Dependence depicted with an arrow pointing from source to sink.

Definition of a Dependency

Load-Store Classification of Dependencies

True Dependency(RAW)	Anti Dependency(WAR)	Output dependency(WAW)
S1 X = ..	S1 .. = X	S1 X = ...
S2 .. = X	S2 X = ..	S2 X = ...

Def. Loop Dependence: There is a dependence from statement $S1$ to $S2$ in a loop nest *iff* there exist iterations $\vec{k} \leq \vec{l}$ and an execution path from statement $S1$ to statement $S2$ **such that:**

1. $S1$ accesses memory location M on iteration \vec{k} , and
2. $S2$ accesses memory location M on iteration \vec{l} , and
3. one of these accesses is a write.

We say that $S1$ is the source and $S2$ is the sink of the dependence, because $S1$ executes before $S2$ in the sequential program execution. Dependence depicted with an arrow pointing from source to sink.

We are most interested in cross iteration dependencies, i.e., $\vec{k} < \vec{l}$.

Loop-Nest Dependencies

Lexicographic ordering, e.g., $\vec{k} = (i=2, j=4) < \vec{l} = (i=3, j=3)$.

Three Loop Examples

```
DO i = 1, N
  DO j = 1, N
    A[j,i] = A[j,i] ..
  ENDDO
ENDDO
```

```
DO i = 2, N
  DO j = 2, N
    A[j,i] = A[j-1,i-1]...
    B[j,i] = B[j-1,i]...
  ENDDO
ENDDO
```

```
DO i = 2, N
  DO j = 1, N
    A[i,j] = ...
    A[i-1,j+1]
  ENDDO
ENDDO
```

Loop-Nest Dependencies

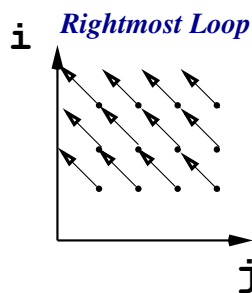
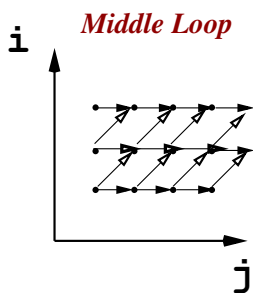
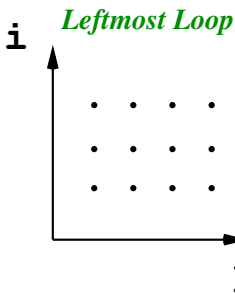
Lexicographic ordering, e.g., $\vec{k} = (i=2, j=4) < \vec{l} = (i=3, j=3)$.

Three Loop Examples

```
DO i = 1, N
  DO j = 1, N
    A[j,i] = A[j,i] ..
  ENDDO
ENDDO
```

```
DO i = 2, N
  DO j = 2, N
    A[j,i] = A[j-1,i-1]...
    B[j,i] = B[j-1,i]...
  ENDDO ENDDO
```

```
DO i = 2, N
  DO j = 1, N
    A[i,j] = ...
    A[i-1,j+1]
  ENDDO ENDDO
```



How can I summarize this information?

Aggregate Dependencies via Direction Vectors

Three Loop Examples

```
DO i = 1, N
  DO j = 1, N
    A[j,i] = A[j,i] ..
  ENDDO
ENDDO
```

```
DO i = 2, N
  DO j = 2, N
    A[j,i] = A[j-1,i-1]...
    B[j,i] = B[j-1,i]...
  ENDDO
ENDDO
```

```
DO i = 2, N
  DO j = 1, N
    A[i,j] = ...
    A[i-1,j+1]
  ENDDO
ENDDO
```

Dependencies depicted via an edge *from* the stmt that executes first in the loop nest (*source*), *to* the one that executes later (*sink*).

Def. Dependence Direction: Assume there exists a dependence from $S1$ in iteration \vec{k} to $S2$ in \vec{l} ($\vec{k} \leq \vec{l}$). *The direction vector is:*

1. $\vec{D}(\vec{k}, \vec{l})_m = "<"$ if $\vec{k}_m < \vec{l}_m$,
2. $\vec{D}(\vec{k}, \vec{l})_m = "="$ if $\vec{k}_m = \vec{l}_m$,
3. $\vec{D}(\vec{k}, \vec{l})_m = ">"$ if $\vec{k}_m > \vec{l}_m$.
4. $\vec{D}(\vec{k}, \vec{l})_m = "*" \text{ otherwise.}$

If the source is a write and the sink a read then RAW dependency, if the source is a read then WAR, if both are writes then WAW.

How to Compute the Direction Vectors?

- ▶ For any two statements $S1$ and $S2$ that may access the same array A (and one of the accesses is a write),
- ▶ in two symbolic iterations $I^1 \equiv (i_1^1, \dots, i_m^1)$ and $I^2 \equiv (i_1^2, \dots, i_m^2)$ (assuming $I^1 \leq I^2$)
- ▶ on indices $A[e_1^1, \dots, e_n^1]$ and $A[e_1^2, \dots, e_n^2]$, respectively,
- ▶ then *the direction vectors may be derived* from the equations
$$\begin{cases} e_1^1 = e_1^2 \\ \dots \\ e_n^1 = e_n^2 \end{cases}$$

(The system of equations models the definition of a dependency: both accesses need to refer to the same memory location!)

When is a Loop Parallel?

Direction Vectors/Matrix for Three Loops

```
DO i = 1, N
  DO j = 1, N
S1    A[j,i]=A[j,i]..
      ENDDO
    ENDDO
```

When is a Loop Parallel?

Direction Vectors/Matrix for Three Loops

```
DO i = 1, N          DO i = 2, N
  DO j = 1, N          DO j = 2, N
S1    A[j,i]=A[j,i].. S1    A[j,i]=A[j-1,i]...
      ENDDO            S2    B[j,i]=B[j-1,i-1]...
      ENDDO            ENDDO
For S1→S1:            ENDDO
  (j1,i1)=(j2,i2)
  i1 = i2 & j1 = j2
```

Direction matrix:

S1→S1: [=,=]

When is a Loop Parallel?

Direction Vectors/Matrix for Three Loops

<pre>DO i = 1, N DO j = 1, N S1 A[j,i]=A[j,i].. ENDDO ENDDO</pre>	<pre>DO i = 2, N DO j = 2, N S1 A[j,i]=A[j-1,i]... S2 B[j,i]=B[j-1,i-1]... ENDDO ENDDO</pre>	<pre>DO i = 2, N DO j = 1, N S1 A[i,j]=A[i-1,j+1].. ENDDO ENDDO</pre>
<pre>For S1→S1: (j1,i1)=(j2,i2) i1 = i2 & j1 = j2</pre>	<pre>S1→S1: (j1,i1)=(j2-1,i2) i1 = i2 & j1 < j2 S2→S2: (j1,i1)=(j2-1,i2-1) i1 < i2 & j1 < j2</pre>	<pre>For S1→S1:</pre>
Direction matrix:		
S1→S1: [=,=]	S1→S1: [=,<]	
	S2→S2: [<,<]	

When is a Loop Parallel?

Direction Vectors/Matrix for Three Loops

<pre>DO i = 1, N DO j = 1, N S1 A[j,i]=A[j,i].. ENDDO ENDDO</pre>	<pre>DO i = 2, N DO j = 2, N S1 A[j,i]=A[j-1,i]... S2 B[j,i]=B[j-1,i-1]... ENDDO ENDDO</pre>	<pre>DO i = 2, N DO j = 1, N S1 A[i,j]=A[i-1,j+1].. ENDDO ENDDO</pre>
<pre>For S1→S1: (j1,i1)=(j2,i2) i1 = i2 & j1 = j2</pre>	<pre>S1→S1: (j1,i1)=(j2-1,i2) i1 = i2 & j1 < j2 S2→S2: (j1,i1)=(j2-1,i2-1) i1 < i2 & j1 < j2</pre>	<pre>For S1→S1: (i1,j1) = (i2-1,j2+1) i1 < i2 & j1 > j2</pre>
Direction matrix: S1→S1: [=,=]	S1→S1: [=,<] S2→S2: [<,<]	Direction matrix: S1→S1: [<,>]

Th. Parallelism: A loop in a loop nest is parallel *iff* for each entry in its direction column, the entry is either = or there exists an outer loop whose corresponding direction is <.

A direction vector cannot have > as the first non-= symbol, as that would mean that I depend on something in the future.

Parallelism and Loop Interchange

Direction Vectors/Matrix for Three Loops

DO i = 1, N DO j = 1, N S1 A[j,i]=A[j,i].. ENDDO ENDDO	DO i = 2, N DO j = 2, N S1 A[j,i]=A[j-1,i].. S2 B[j,i]=B[j-1,i-1].. ENDDO ENDDO	DO i = 2, N DO j = 1, N S1 A[i,j] = ... A[i-1,j+1] ENDDO ENDDO
For S1→S1: (j1,i1)=(j2,i2) i1 = i2 & j1 = j2	S1→S1: (j1,i1)=(j2-1,i2) i1 = i2 & j1 < j2 S2→S2: (j1,i1)=(j2-1,i2-1) i1 < i2 & j1 < j2	For S1→S1: (i1,j1) = (i1-1,j2+1) i1 < i2 & j1 > j2
Direction matrix: S1→S1: [=,=]	S1→S1: [=,<] S2→S2: [<,<]	Direction matrix: S1→S1: [<,>]

Th. Loop Interchange: Interchanging two loops in a loop nest is legal *iff* interchanging the columns of the direction matrix in the same way *does NOT* result in a > direction as the leftmost non-= direction in a row.

Parallelism and Loop Interchange

Direction Vectors/Matrix for Three Loops

```
DO i = 1, N
  DO j = 1, N
S1    A[j,i]=A[j,i]..
      ENDDO
    ENDDO
```

For S1→S1: j1 = j2
 i1 = i2
(i2,j2)-(i1,j1)=
[=,=]

```
DO i = 2, N
  DO j = 2, N
S1    A[j,i]=A[j-1,i]...
S2    B[j,i]=B[j-1,i-1]...
      ENDDO ENDDO
```

For S1→S1: j1 = j2-1
 i1 = i2
(i2,j2)-(i1,j1)=[=,<]
For S2→S2: j1 = j2-1
 i1 = i2-1
(i2,j2)-(i1,j1)=[<,<]

```
DO i = 2, N
  DO j = 1, N
S1    A[i,j]=A[i-1,j+1]..
      ENDDO
    ENDDO
```

For S1→S1: i1 = i2-1
 j1 = j2+1
(i2,j2)-(i1,j1)=
[<,>]

Interchange is safe for the 1st and 2nd nests, but not for the 3rd!

e.g., [=,<] → [<,<] (for the second loop nest)
 [<,<] [<,<]

Parallelism and Loop Interchange

Direction Vectors/Matrix for Three Loops

```
DO i = 1, N
  DO j = 1, N
S1    A[j,i]=A[j,i]..
      ENDDO
    ENDDO
```

For S1→S1: j1 = j2
 i1 = i2
(i2,j2)-(i1,j1)=
[=,=]

```
DO i = 2, N
  DO j = 2, N
S1    A[j,i]=A[j-1,i]...
S2    B[j,i]=B[j-1,i-1]...
      ENDDO ENDDO
```

For S1→S1: j1 = j2-1
 i1 = i2
(i2,j2)-(i1,j1)=[=,<]
For S2→S2: j1 = j2-1
 i1 = i2-1
(i2,j2)-(i1,j1)=[<,<]

```
DO i = 2, N
  DO j = 1, N
S1    A[i,j]=A[i-1,j+1]..
      ENDDO
    ENDDO
```

For S1→S1: i1 = i2-1
 j1 = j2+1
(i2,j2)-(i1,j1)=
[<,>]

Interchange is safe for the 1st and 2nd nests, but not for the 3rd!

e.g., [=,<] → [<,<] (for the second loop nest)
 [<,<] [<,<]

After interchange, loop j of the second loop nest is parallel.

Corollary: A parallel loop can be always interchanged inwards.

Dependency Graph and Loop Distribution

Def. Dependency Graph: edges from the source of the dependency, i.e., early iteration, to the sink, i.e., later iteration.

Th. Loop Distribution: Statements that are in a dependence cycle remain in one (sequential) loop. The others are distributed to separate loops in graph order; if no cycle then parallel loops.

Vectorization Example

```
DO i = 3, N
S1  A[i] = B[i-2] ...
S2  B[i] = B[i-1] ...
ENDDO
```

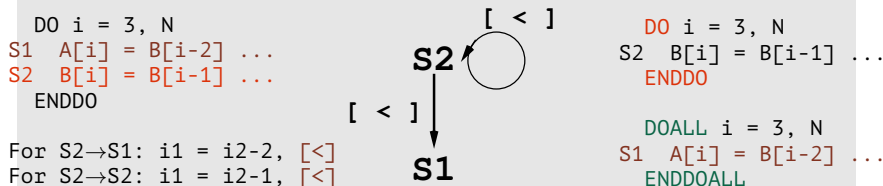
```
For S2→S1: i1 = i2-2, [<]
For S2→S2: i1 = i2-1, [<]
```

Dependency Graph and Loop Distribution

Def. Dependency Graph: edges from the source of the dependency, i.e., early iteration, to the sink, i.e., later iteration.

Th. Loop Distribution: Statements that are in a dependence cycle remain in one (sequential) loop. The others are distributed to separate loops in graph order; if no cycle then parallel loops.

Vectorization Example



Corollary: It is always legal to distribute a parallel loop;
but requires array expansion for local variables or if output
dependencies are present.

Loop Distribution May Require Array Expansion

```
float tmp;  
for(i=2; i<N; i++) {  
    tmp = 2*B[i-2];  
    A[i] = tmp;  
    B[i] = tmp+B[i-1]  
}
```

```
float tmp[N];  
for(int i=2; i<N; i++) {  
    tmp[i] = 2*B[i-2];  
    B[i] = tmp[i]+B[i-1];  
}  
  
forall(int i=2; i<N; i++) {  
    A[i] = tmp[i];  
}
```

No matter where tmp is declared (inside or outside the loop) it needs to be expanded into an array in order to do loop distribution.

If tmp is declared outside the loop then requires **privatization**,

Loop Distribution May Require Array Expansion

```
float tmp;  
for(i=2; i<N; i++) {  
    tmp = 2*B[i-2];  
    A[i] = tmp;  
    B[i] = tmp+B[i-1]  
}
```

```
float tmp[N];  
for(int i=2; i<N; i++) {  
    tmp[i] = 2*B[i-2];  
    B[i] = tmp[i]+B[i-1];  
}  
  
forall(int i=2; i<N; i++) {  
    A[i] = tmp[i];  
}
```

No matter where tmp is declared (inside or outside the loop) it needs to be expanded into an array in order to do loop distribution.

If tmp is declared outside the loop then requires **privatization**, because it actually causes frequent waw dependencies. However its value is written before being used within the same iteration. Hence it is semantically equivalent to a locally declared variable, which will remove the output (waw) dependency.

Distribution requires array expansion of the scalar tmp.

False Dependencies (WAR/WAW)

- ▶ **Cross-Iteration Anti Dependencies (WAR)** correspond to a read from the array as it was before the loop \Rightarrow can be eliminated by reading from a copy of the array.
- ▶ **Cross-Iteration WAW Dependencies (WAW):**
If they correspond to the case in which every **read** from a scalar or array location is covered by a **previous same-iteration write** \Rightarrow can be eliminated **privatization (renaming)**, which semantically moves the declaration of the variable (scalar or array) inside the loop.
- ▶ Direction-vectors reasoning is limited to relatively simple loop nests, e.g., difficult to reason about privatization in such a way.

Example: Eliminating WAW Dependencies

```
float A[M];  
for(i=0; i<N; i++){  
    for(int j=0, j<M; j++)  
        A[j] = (4*i+4*j) % M;  
    for(int k=0; k<N; k++)  
        X[i][k] = X[i][k-1] *  
            A[A[(2*i+k)%M]%M];  
}
```

Example: Eliminating WAW Dependencies

// The write to A[j] causes many WAWs,

```
float A[M];
for(i=0; i<N; i++){
    for(int j=0, j<M; j++)
        A[j] = (4*i+4*j) % M;
    for(int k=0; k<N; k++)
        X[i][k] = X[i][k-1] *
            A[A[(2*i+k)%M]%M];
}
```

Example: Eliminating WAW Dependencies

```
float A[M];
for(i=0; i<N; i++){
    for(int j=0, j<M; j++)
        A[j] = (4*i+4*j) % M;
    for(int k=0; k<N; k++)
        X[i][k] = X[i][k-1] *
            A[A[(2*i+k)%M]%M];
}

// The write to A[j] causes many WAWs,
// but A is fully written in loop j
// float A[N][M];
forall(int i=0; i<N; i++){
    float A[M];
    for(int j=0, j<M; j++)
        A[j] = (4*i+4*j) % M; // A[i][j]
    for(int k=0; k<N; k++)
        X[i][k]=X[i][k-1] *
            A[A[(2*i+k)%M]%M];
        //A[i][A[i,...]]
    }
```

For OpenCL programming you would likely use array expansion (`float A[N,M]`) because the size M is likely unknown at compile time.

Putting Everything Together

```
float X[M][N][N]; /* ... compute X ... */
float A[N][N];
for(int i = 0; i < M; i++) {
    //initialize A
    for(int k = 0; k < N; k++) {
        for(int j = 0; j < N; j++) {
            float x = X[i][j][k];
            A[k][j] = x * x;
        }
    }
    // convergence loop
    for(int t = 0; t < T; t++) {
        for(int j = 0; j < N; j++) {
            for(int k = 0; k < N; k++) {
                A[k][j] = fsqrt(A[k][j] * A[k-1][j]);
                X[i][k][j] += fsqrt(B[i][k-1][j-1]);
            }
        }
    }
}
```

Exercise: Assume the code above runs on a dataset having $M = 128$ and $N = 512$ and, less important, say $T = 256$.

- (1) Is this code suitable for GPU execution?
- (2) Transform the code to one suitable for GPU execution.

Content of Day2 Lecture:

Dependence Analysis

Problem Statement, Dependency Definition

Dependency Matrix, Loop Parallelism and Interchange

Loop Distribution, Array Expansion/Privatization

Exercise: Putting it all together

Optimizing Spatial Locality (Coalesced Memory)

Motivating Example on A Contrived Program

Uncoalesced Matrix Transposition Kernel

Coding Exercise: Entirely-Coalesced Transposition

Program Optimized By Transposition + Coding Exercise

Fully Fused Program + Coding Exercise

Optimizing Temporal Locality

Naive Matrix-Matrix Multiplication

Block-Tiled Matrix Multiplication + Coding Exercise

Register-Tiled Matrix Multiplication + Coding Exercise

Motivation: Coalesced Accesses to Global Memory

Which are the parallel/sequential loops?

```
real A[height][width];
real B[height][width];
// Non-Coalesced Access
for(i=0; i<height; i++) {
    real accum = 0.0;

    for(j=0; j<width; j++) {
        real tmpA = A[i][j];
        accum = tmpA*tmpA - accum;
        B[i][j] = accum;
    }
}
```

Motivation: Coalesced Accesses to Global Memory

Which are the parallel/sequential loops?

```
real A[height][width];
real B[height][width];
// Non-Coalesced Access
for(i=0; i<height; i++) {
    real accum = 0.0;

    for(j=0; j<width; j++) {
        real tmpA = A[i][j];
        accum = tmpA*tmpA - accum;
        B[i][j] = accum;
    }
}

A' = transpose(A);
// Coalesced Accesses
for(i=0; i<height; i++) {
    real accum = 0.0;

    for(int j=0; j<width; j++) {
        real tmpA = A'[j][i];
        accum = tmpA*tmpA - accum;
        B'[j][i] = accum;
    }
}
B = transpose(B');
```

- ▶ Loop *i* is parallel;
- ▶ loop *j* is sequential because of the RAW cross-iteration dependencies on accum.
- ▶ The transformed program performs about $3\times$ the number of global-memory accesses than the original.

Motivation: Coalesced Accesses to Global Memory

Which are the parallel/sequential loops?

```
real A[height][width];
real B[height][width];
// Non-Coalesced Access
for(i=0; i<height; i++) {
    real accum = 0.0;

    for(j=0; j<width; j++) {
        real tmpA = A[i][j];
        accum = tmpA*tmpA - accum;
        B[i][j] = accum;
    }
}

A' = transpose(A);
// Coalesced Accesses
for(i=0; i<height; i++) {
    real accum = 0.0;

    for(int j=0; j<width; j++) {
        real tmpA = A'[j][i];
        accum = tmpA*tmpA - accum;
        B'[j][i] = accum;
    }
}
B = transpose(B');
```

- ▶ Loop *i* is parallel;
- ▶ loop *j* is sequential because of the RAW cross-iteration dependencies on accum.
- ▶ The transformed program performs about $3\times$ the number of global-memory accesses than the original.
- ▶ But it is significantly faster than the original (on GPUs).

Motivation: What Are Coalesced Accesses?

Coalesced access: a (quarter) wave accesses in a SIMD instruction consecutive words in global-memory.

SIMD Instruction: *Thread Id:*

LW Rx off(Rb)

$16*k+0$	$16*k+1$	\dots	$16*k+15$
----------------------------	----------------------------	---------------------------	-----------------------------

Global Memory:

	word $q+0$	word $q+1$	\dots	word $q+15$	
--	----------------------------------	----------------------------------	---------------------------	-----------------------------------	--

Uncoalesced Transposition Kernel

- Semantically we chunk the matrix into $T \times T$ blocks;
- Each block is processed by one workgroup, e.g., $T=16$.

```
// Initial                                // Blocked
for(i=0; i<height; i++) {                for(ii=0; ii<height; ii+=T) {
  for(j=0; j<width; j++) {                for(jj=0; jj<width; jj+=T) {
    B[j*height+i] =                      for(i=ii; i<min(ii+T,height); i++) {
      A[i*width+j];                      for(j=jj; j<min(jj+T,width); j++) {
    }                                     B[j*height+i] = A[i*width+j];
  }                                     } } } }
}
```

- Local workgroup: $T \times T$;
- Global(1): $\lceil \text{width}/T \rceil * T$; Global 1: $\lceil \text{height}/T \rceil * T$.

Uncoalesced Transposition Kernel

- Semantically we chunk the matrix into $T \times T$ blocks;
- Each block is processed by one workgroup, e.g., $T=16$.

```
// Initial                                // Blocked
for(i=0; i<height; i++) {                for(ii=0; ii<height; ii+=T) {
    for(j=0; j<width; j++) {              for(jj=0; jj<width; jj+=T) {
        B[j*height+i] =                  for(i=ii; i<min(ii+T,height); i++) {
            A[i*width+j];                  for(j=jj; j<min(jj+T,width); j++) {
        }                                  B[j*height+i] = A[i*width+j];
    } } } }                               }
```

- Local workgroup: $T \times T$;
- Global(1): $\lceil \text{width}/T \rceil * T$; Global 1: $\lceil \text{height}/T \rceil * T$.

```
__kernel void naiveTransp( __global real* A, __global real* B
                          , int height, int width ) {
    unsigned int i = get_global_id(1);
    unsigned int j = get_global_id(0);
    if( (j >= width) || (i >= height) ) return;
    B[j*height + i] = A[i*width + j]; // uncoalesced access!
}
```


Exercise 1: Fill In Coalesced Transposition Kernel

```
// verifies that workgroup size is: TILE*TILE
__kernel __attribute__((reqd_work_group_size(TILE, TILE, 1)))
__kernel void coalsTransp( __global real* A,
    __global real* B, uint32_t height, uint32_t width ) {
    __local float tile[TILE][TILE];                // local memory
    int gidx = get_global_id(0); int gidy = get_global_id(1);
    int lidx = get_local_id(0); int lidy = get_local_id(1);

    // 1. read current element from global to local memory.
    //    make sure to NOT access arrays out of bounds!
    // 2. current thread loads from local memory the transposed element
    //    at workgroup level, and writes it in the proper position in B.
    //    Useful functions: get_group_id(0), get_local_size(0); or 1.
    // 3. Insert necessary workgroup-level synchronization
    //    barrier(CLK_LOCAL_MEM_FENCE);
}
```

Transposing a (blocked) matrix \equiv moving each block to the transposed position + transposing each block!

Exercise 1: Fill In Coalesced Transposition Kernel

```
// verifies that workgroup size is: TILE*TILE
__kernel __attribute__((reqd_work_group_size(TILE, TILE, 1)))
__kernel void coalsTransp( __global real* A,
    __global real* B, uint32_t height, uint32_t width ) {
    __local float tile[TILE][TILE]; // local memory
    int gidx = get_global_id(0); int gidy = get_global_id(1);
    int lidx = get_local_id(0); int lidy = get_local_id(1);

    // 1. read current element from global to local memory.
    //    make sure to NOT access arrays out of bounds!
    // 2. current thread loads from local memory the transposed element
    //    at workgroup level, and writes it in the proper position in B.
    //    Useful functions: get_group_id(0), get_local_size(0); or 1.
    // 3. Insert necessary workgroup-level synchronization
    //    barrier(CLK_LOCAL_MEM_FENCE);
}
```

Transposing a (blocked) matrix \equiv moving each block to the transposed position + transposing each block!

- Implement in `Day2-Exercises/Transp/kernels.cl`
- Are the read and write accesses to global memory coalesced?
- Why/Where should you place the barrier?

Exercise 2: Coalescing By Transposition

```
real A[height][width];
real B[height][width];
// Non-Coalesced Access
for(i=0; i<height; i++) {
    real accum = 0.0;

    for(j=0; j<width; j++) {
        real tmpA = A[i][j];
        accum = tmpA*tmpA - accum;
        B[i][j] = accum;
    }
}
```

Exercise 2: Coalescing By Transposition

```
real A[height][width];
real B[height][width];
// Non-Coalesced Access
for(i=0; i<height; i++) {
    real accum = 0.0;

    for(j=0; j<width; j++) {
        real tmpA = A[i][j];
        accum = tmpA*tmpA - accum;
        B[i][j] = accum;
    }
}

A' = transpose(A);
// Coalesced Accesses
for(i=0; i<height; i++) {
    real accum = 0.0;

    for(int j=0; j<width; j++) {
        real tmpA = A'[j][i];
        accum = tmpA*tmpA - accum;
        B'[j][i] = accum;
    }
}
B = transpose(B');
```

- ▶ Open File "Day2-Exercises/Transp/coalescing.c", function named "runGPUcoalsProgram".
- ▶ Fill the correct sizes for the "coalsProgrm" kernel call.
- ▶ Open File "Day2-Exercises/Transp/kernels.cl" and implement kernel named "coalsProgrm".
- ▶ Fix bugs until it validates! :-)
- ▶ What is the speedup with respect to the naive program?
- ▶ What fraction of the memcopy bandwidth is achieved?

Exercise 3: Fused Program With Coalesced Accesses

- ▶ Open File "Day2-Exercises/Transp/kernels.cl" and implement the missing pieces from kernel named "optimProgram";
- ▶ The idea is to have coalesced accesses "fused" into the kernel by each thread processing CHUNK elements at a time by:
 - ▶ collectively copying $\text{CHUNK} * \text{WORKGROUP_SIZE}$ input elements from global to local memory in coalesced fashion;
 - ▶ each thread now processes CHUNK elements from its row;
 - ▶ collectively copying $\text{CHUNK} * \text{WORKGROUP_SIZE}$ result elements from local to global memory in coalesced fashion;
 - ▶ repeat in a loop (inside the kernel) until all row's elements have been processed!
- ▶ Fix bugs until it validates! :-)
- ▶ What is the speedup with respect to the transposition-optimized program?
- ▶ What fraction of the memcpy bandwidth is achieved?

Content of Day2 Lecture:

Dependence Analysis

- Problem Statement, Dependency Definition

- Dependency Matrix, Loop Parallelism and Interchange

- Loop Distribution, Array Expansion/Privatization

- Exercise: Putting it all together

Optimizing Spatial Locality (Coalesced Memory)

- Motivating Example on A Contrived Program

- Uncoalesced Matrix Transposition Kernel

- Coding Exercise: Entirely-Coalesced Transposition

- Program Optimized By Transposition + Coding Exercise

- Fully Fused Program + Coding Exercise

Optimizing Temporal Locality

- Naive Matrix-Matrix Multiplication

- Block-Tiled Matrix Multiplication + Coding Exercise

- Register-Tiled Matrix Multiplication + Coding Exercise

Matrix Multiplication: Loop Strip Mining

```
for (int i = 0; i < M; i++) { // Parallel
    for (int j = 0; j < N; j++) { // Parallel
        float tmp = 0.0
        for(int k = 0; k < U; k++) { // Reduction
            tmp += A[i,k]*B[k,j]
        }
        C[i,j] = tmp;
    }
}
```

Matrices:

► $A \in \mathcal{M}^{M \times U}$

► $B \in \mathcal{M}^{U \times N}$

► $C \in \mathcal{M}^{M \times N}$

Loops of indices i
and j are parallel.

Accesses to A and B invariant to loops i and $j \Rightarrow$

The idea is to apply Block Tiling to optimize temporal locality!

Loop Strip Mining

```
for(int i = 0; i < N; i++) {  
    loop_body(i)  
}  
⇒  
for(int ii = 0; ii < N; ii += T) {  
    for(int i = ii; i < MIN(ii+T, N); i++) {  
        loop_body(i)  
    }  
}
```

Strip Mining is always safe: the transformed loop executes the same instructions in the same order as the original loop!

Strip mining all loops by some tile T result in:

Loop Strip Mining

```
for(int i = 0; i < N; i++) {  
    loop_body(i)  
}  
⇒  
for(int ii = 0; ii < N; ii += T) {  
    for(int i = ii; i < MIN(ii+T,N); i++) {  
        loop_body(i)  
    }  
}
```

Strip Mining is always safe: the transformed loop executes the same instructions in the same order as the original loop!

Strip mining all loops by some tile T result in:

```
for (int ii = 0; ii < M; ii += T) { // Parallel  
    for (int i = ii; i < MIN(ii+T,M); i++) { // Parallel  
        for (int jj = 0; jj < N; jj += T) { // Parallel  
            for (int j = jj; j < MIN(jj+T,N); j++) { // Parallel  
                float tmp = 0.0;  
                for(int kk = 0; kk < U; k += T) { // Reduction  
                    for(int k=kk; k<MIN(kk+T,U); k++) { // Reduction  
                        tmp += A[i,k]*B[k,j];  
                    }  
                }  
                C[i,j] = tmp;  
            }  
        }  
    }  
}
```

Block Tiling: Strip Mining + Interchange

Now interchange the loops if index i and jj . Why is this safe?

```
for (int ii = 0; ii < M; ii += T) {           // Global 1
    for (int jj = 0; jj < N; jj += T) {       // Global 0
        for (int i = ii; i < MIN(ii+T,M); i++) { // Local 1
            for (int j = jj; j < MIN(jj+T,N); j++) { // Local 0
                float tmp = 0.0;
                for(int kk = 0; kk < U; k += T) { // Sequential

                    //What slices of A and B are used in the loop below?

                    for(int k=kk; k<MIN(kk+T,U); k++) { // Sequential
                        tmp += A[i,k]*B[k,j];
                    }
                }
                C[i,j] = tmp;
            }
        }
    }
}
```

Block Tiling: Strip Mining + Interchange

Now interchange the loops if index i and jj . Why is this safe?

```
for (int ii = 0; ii < M; ii += T) {           // Global 1
    for (int jj = 0; jj < N; jj += T) {       // Global 0
        for (int i = ii; i < MIN(ii+T,M); i++) { // Local 1
            for (int j = jj; j < MIN(jj+T,N); j++) { // Local 0
                float tmp = 0.0;
                for(int kk = 0; kk < U; k += T) { // Sequential

                    //What slices of A and B are used in the loop below?

                    for(int k=kk; k<MIN(kk+T,U); k++) { // Sequential
                        tmp += A[i,k]*B[k,j];
                    }
                }
                C[i,j] = tmp;
            } } } }
```

$A[ii : \text{MIN}(ii+T,M)][kk : \text{MIN}(kk+T,U)]$
 $B[kk : \text{MIN}(kk+T,U)][jj : \text{MIN}(jj+T,N)]$

Each array slice has $T \times T$ elements and there are $T \times T$ threads in a workgroup; so each thread can copy one element to local memory. Thus, we cut the number of global-memory accesses by a T factor.

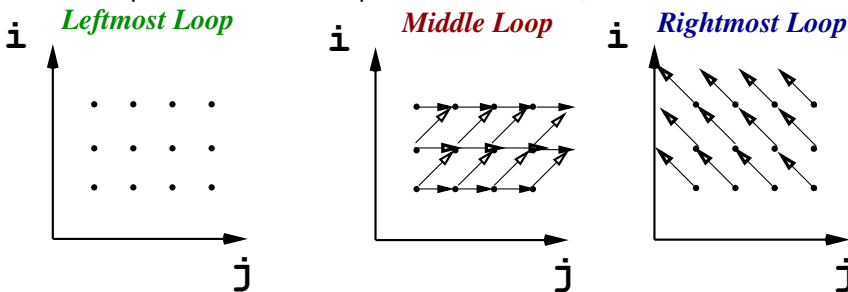
Exercise 4: Implement Block-Tiled Matrix Multiplication

- ▶ Open File "Day2-Exercises/MMMult/mmm.c", function named "runGPUverMMM".
- ▶ Place there the correct sizes for the matrix multiplication kernels. That would make the naive version work correctly.
- ▶ Open File "Day2-Exercises/MMMult/mmm.cl" and implement kernel named "blockMMM".
- ▶ Fix bugs until it validates! :-)

Exercise 4: Implement Block-Tiled Matrix Multiplication

- ▶ Open File "Day2-Exercises/MMMult/mmm.c", function named "runGPUverMMM".
- ▶ Place there the correct sizes for the matrix multiplication kernels. That would make the naive version work correctly.
- ▶ Open File "Day2-Exercises/MMMult/mmm.cl" and implement kernel named "blockMMM".
- ▶ Fix bugs until it validates! :-)

The parallel iteration space is block tiled, not the matrices:



Exercise 4: Implement Block-Tiled Matrix Multiplication

Copy $A[ii:MIN(ii+T,M)][kk:MIN(kk+T,U)]$ in A_loc of size $[T][T]$

Copy $B[kk:MIN(kk+T,U)][jj:MIN(jj+T,N)]$ in B_loc of size $[T][T]$

```
for (int ii = 0; ii < M; ii += T) { // Global 1
    for (int jj = 0; jj < N; jj += T) { // Global 0
        for (int i = ii; i < MIN(ii+T,M); i++) { // Local 1
            for (int j = jj; j < MIN(jj+T,N); j++) { // Local 0
                float tmp = 0.0;
                for(int kk = 0; kk < U; k += T) { // Sequential

                    // Collectively copy to local memory.
                    // Loop of index k should only use local memory.
                    // Add barrier here. Why?

                    for(int k=kk; k<MIN(kk+T,U); k++) { // Sequential
                        tmp += A_loc[i-ii,k-kk] * B_loc[k-kk,j-jj];
                    } // Add barrier here. Why?
                } C[i,j] = tmp;
            }
        }
    }
}
```

Modify loops of index i , j and k to go from 0 to T with stride 1 . Unroll loop k .

OpenCL equivalences:

Global workgroup size:

Exercise 4: Implement Block-Tiled Matrix Multiplication

Copy $A[ii:MIN(ii+T,M)][kk:MIN(kk+T,U)]$ in A_loc of size $[T][T]$

Copy $B[kk:MIN(kk+T,U)][jj:MIN(jj+T,N)]$ in B_loc of size $[T][T]$

```
for (int ii = 0; ii < M; ii += T) { // Global 1
    for (int jj = 0; jj < N; jj += T) { // Global 0
        for (int i = ii; i < MIN(ii+T,M); i++) { // Local 1
            for (int j = jj; j < MIN(jj+T,N); j++) { // Local 0
                float tmp = 0.0;
                for(int kk = 0; kk < U; k += T) { // Sequential

                    // Collectively copy to local memory.
                    // Loop of index k should only use local memory.
                    // Add barrier here. Why?

                    for(int k=kk; k<MIN(kk+T,U); k++) { // Sequential
                        tmp += A_loc[i-ii,k-kk] * B_loc[k-kk,j-jj];
                    } // Add barrier here. Why?
                }
                C[i,j] = tmp;
            }
        }
    }
}
```

Modify loops of index i , j and k to go from 0 to T with stride 1. Unroll loop k .

OpenCL equivalences:

Global workgroup size: $(\lceil M/T \rceil * T) \times (\lceil N/T \rceil * T)$. Local Workgroup size: $T \times T$.

Local ids:

Exercise 4: Implement Block-Tiled Matrix Multiplication

Copy A[ii:MIN(ii+T,M)][kk:MIN(kk+T,U)] in A_loc of size [T][T]

Copy B[kk:MIN(kk+T,U)][jj:MIN(jj+T,N)] in B_loc of size [T][T]

```
for (int ii = 0; ii < M; ii += T) { // Global 1
    for (int jj = 0; jj < N; jj += T) { // Global 0
        for (int i = ii; i < MIN(ii+T,M); i++) { // Local 1
            for (int j = jj; j < MIN(jj+T,N); j++) { // Local 0
                float tmp = 0.0;
                for(int kk = 0; kk < U; k += T) { // Sequential

                    // Collectively copy to local memory.
                    // Loop of index k should only use local memory.
                    // Add barrier here. Why?

                    for(int k=kk; k<MIN(kk+T,U); k++) { // Sequential
                        tmp += A_loc[i-ii,k-kk] * B_loc[k-kk,j-jj];
                    } // Add barrier here. Why?
                } C[i,j] = tmp;
            } } } }
```

Modify loops of index i, j and k to go from 0 to T with stride 1. Unroll loop k.

OpenCL equivalences:

Global workgroup size: $(\lceil M/T \rceil * T) \times (\lceil N/T \rceil * T)$. Local Workgroup size: $T \times T$.

Local ids: $i - ii = \text{get_local_id}(1)$, $j - jj = \text{get_local_id}(0)$

Global ids: $i = \text{get_global_id}(1)$, $j = \text{get_global_id}(0)$

Similar: $ii = \text{get_group_id}(1)*T$, $jj = \text{get_group_id}(0)*T$

Local memory declared with: `__local float A_loc[T][T];`

Exercise 4: Implement Block-Tiled Matrix Multiplication

Loops of index *i* and *j* form the local workgroup. Interchange them innermost and distribute them:

```
for (int ii = 0; ii < M; ii += T) { // Global 1
    for (int jj = 0; jj < N; jj += T) { // Global 0
        float A_loc[T][T], B_loc[T][T]; // local memory
        float tmp_reg[T][T]; // register memory: still represented as a scalar for a thread

        for (int i = ii; i < ii+T; i++) { // Local 1
            for (int j = jj; j < jj+T; j++) { // Local 0
                tmp[i-ii][j-jj] = 0.0;
            }
        }
        for (int kk = 0; kk < U; k += T) { // Sequential
            for (int i = ii; i < ii+T; i++) { // Local
                for (int j = jj; j < jj+T; j++) { // Local
                    A_loc[i-ii][k-kk] = (i < M && k < U)? A[i][kk+j-jj] : 0.0; // copy to local mem
                    B_loc[k-kk][j-jj] = (j < N && k < U)? B[kk+i-ii][j] : 0.0; // copy to local mem
                } // needs barrier here, why?
            }
            for (int i = ii; i < ii+T; i++) { // Local 1
                for (int j = jj; j < jj+T; j++) { // Local 0
                    for (int k=kk; k < kk+T; k++) { // Sequential
                        tmp[i][j] += A_loc[i-ii,k-kk] * B[k-kk,j-jj]; // use local memory
                    }
                }
            }
            for (int i = ii; i < ii+T; i++) { // Local 1
                for (int j = jj; j < jj+T; j++) { // Local 0
                    C[i,j] = tmp;
                } // needs barrier here, why?
            }
        }
    }
}
```

Modify loops of index *i*, *j* and *k* to go from 0 to T with stride 1. Unroll loop *k*.

OpenCL equivalences:

Global workgroup size:

Exercise 4: Implement Block-Tiled Matrix Multiplication

Loops of index i and j form the local workgroup. Interchange them innermost and distribute them:

```
for (int ii = 0; ii < M; ii += T) { // Global 1
    for (int jj = 0; jj < N; jj += T) { // Global 0
        float A_loc[T][T], B_loc[T][T]; // local memory
        float tmp_reg[T][T]; // register memory: still represented as a scalar for a thread

        for (int i = ii; i < ii+T; i++) { // Local 1
            for (int j = jj; j < jj+T; j++) { // Local 0
                tmp[i-ii][j-jj] = 0.0;
            }
        }
        for (int kk = 0; kk < U; k += T) { // Sequential
            for (int i = ii; i < ii+T; i++) { // Local
                for (int j = jj; j < jj+T; j++) { // Local
                    A_loc[i-ii][k-kk] = (i < M && k < U)? A[i][kk+j-jj] : 0.0; // copy to local mem
                    B_loc[k-kk][j-jj] = (j < N && k < U)? B[kk+i-ii][j] : 0.0; // copy to local mem
                } // needs barrier here, why?
            }
            for (int i = ii; i < ii+T; i++) { // Local 1
                for (int j = jj; j < jj+T; j++) { // Local 0
                    for (int k=kk; k < kk+T; k++) { // Sequential
                        tmp[i][j] += A_loc[i-ii,k-kk] * B[k-kk,j-jj]; // use local memory
                    }
                }
            }
            for (int i = ii; i < ii+T; i++) { // Local 1
                for (int j = jj; j < jj+T; j++) { // Local 0
                    C[i,j] = tmp;
                } // needs barrier here, why?
            }
        }
    }
}
```

Modify loops of index i , j and k to go from 0 to T with stride 1 . Unroll loop k .

OpenCL equivalences:

Global workgroup size: $(\lceil M/T \rceil * T) \times (\lceil N/T \rceil * T)$. Local Workgroup size: $T \times T$.

Local ids:

Exercise 4: Implement Block-Tiled Matrix Multiplication

Loops of index i and j form the local workgroup. Interchange them innermost and distribute them:

```
for (int ii = 0; ii < M; ii += T) { // Global 1
    for (int jj = 0; jj < N; jj += T) { // Global 0
        float A_loc[T][T], B_loc[T][T]; // local memory
        float tmp_reg[T][T]; // register memory: still represented as a scalar for a thread

        for (int i = ii; i < ii+T; i++) { // Local 1
            for (int j = jj; j < jj+T; j++) { // Local 0
                tmp[i-ii][j-jj] = 0.0;
            }
        }
        for(int kk = 0; kk < U; k += T) { // Sequential
            for (int i = ii; i < ii+T; i++) { // Local
                for (int j = jj; j < jj+T; j++) { // Local
                    A_loc[i-ii][k-kk] = (i < M && k < U)? A[i][kk+j-jj] : 0.0; // copy to local mem
                    B_loc[k-kk][j-jj] = (j < N && k < U)? B[kk+i-ii][j] : 0.0; // copy to local mem
                } } // needs barrier here, why?
            for (int i = ii; i < ii+T; i++) { // Local 1
                for (int j = jj; j < jj+T; j++) { // Local 0
                    for(int k=kk; k < kk+T; k++) { // Sequential
                        tmp[i][j] += A_loc[i-ii,k-kk] * B[k-kk,j-jj]; // use local memory
                    }
                }
            }
            for (int i = ii; i < ii+T; i++) { // Local 1
                for (int j = jj; j < jj+T; j++) { // Local 0
                    C[i,j] = tmp;
                } } // needs barrier here, why?
        } } }
```

Modify loops of index i , j and k to go from 0 to T with stride 1 . Unroll loop k .

OpenCL equivalences:

Global workgroup size: $(\lceil M/T \rceil * T) \times (\lceil N/T \rceil * T)$. Local Workgroup size: $T \times T$.

Local ids: $i - ii = \text{get_local_id}(1)$, $j - jj = \text{get_local_id}(0)$

Global ids: $i = \text{get_global_id}(1)$, $j = \text{get_global_id}(0)$

Local memory declared with: `__local float A_loc[T][T];`

Exercise 5: Register+Block Tiled Matrix Multiplication

```
for (int i = 0; i < M; i++) { // Parallel
    for (int j = 0; j < N; j++) { // Parallel
        float c = 0.0
        for(int k = 0; k < U; k++) { // Reduction
            c += A[i,k]*B[k,j]
        }
        C[i,j] = c;
    }
}
```

Matrices:

► $A \in \mathcal{M}^{M \times U}$

► $B \in \mathcal{M}^{U \times N}$

► $C \in \mathcal{M}^{M \times N}$

Loops of indices i
and j are parallel.

We can do even better by sacrificing some parallelism in excess:

- stripmine parallel dimension i by a tile T and stride 1;
then move it innermost and sequentialize it;
- stripmine parallel dimension j by a tile $T \times T$ and stride T and
- stripmine the result by a tile T and stride 1;
- the two tiles of parallel dimension j form the local workgroup.

Do not forget to perform array expansion on c , which we will rename cs .

Exercise 5: Register+Block Tiled Matrix Multiplication

```
unsigned int ii, i, jjj, jj, j, kk, k;
forall (ii = 0; ii < M; ii += T ) {           // Parallel Global 1
    forall (jjj = 0; jjj < N; jjj += T*T ) {   // Parallel Global 0
        float cs[T][T][T];

        forall(jj=jjj; jj<min(jjj+T*T,N); jj+=T){ // Parallel Local 1
            forall(j=jj; j<min(jj+T,N); j++) {    // Parallel Local 0
                for (i=ii; i<min(ii+T,M); i++) {    // sequential
                    cs[(jj-jjj)/T][j-jj][i-ii] = 0.0;
                } } }

        for (kk = 0; kk < U; kk += T) {           // sequential
            // here we will insert a collective copy to local
            // memory of the slice: A[ii : ii+T, kk : kk+T]
            for (k = kk; k < min(kk+T,U); k++) {    // sequential
                forall (jj=jjj; jj<min(jjj+T*T,N); jj+=T) { // Parallel Local 1
                    forall (j=jj; j<min(jj+T,N); j++) {    // Parallel Local 0
                        for (i = ii; i < min(ii+T,M); i++) { // sequential
                            // please modify here to read A from local memory
                            // hoist out B[k][j] outside loop of index i (invariant)
                            cs[(jj-jjj)/T][j-jj][i-ii] += A[i][k] * B[k][j];
                        } } } } }
        } } } }

        forall (jj=jjj; jj<min(jjj+T*T,N); jj+=T) { // Parallel Local 1
            forall (j=jj; j<min(jj+T,N); j++) {    // Parallel Local 0
                for (i = ii; i < min(ii+T, M); i++) { // sequential
                    C[i][j] = cs[(jj-jjj)/T][j-jj][i-ii];
                } } }
        } } }
```

Exercise 5: Register+Block Tiled Matrix Multiplication

- ▶ Open File "Day2-Exercises/MMMult/mmm.c", function named "runGPUverMMM".
- ▶ Fill the correct sizes for the "rgblkMMM" kernel.
- ▶ Open File "Day2-Exercises/MMMult/mmm.cl" and implement kernel named "rgblkMMM".
- ▶ Fix bugs until it validates! :-) **Tile size is called RT** and not T!

OpenCL equivalences:

Exercise 5: Register+Block Tiled Matrix Multiplication

- ▶ Open File "Day2-Exercises/MMMult/mmm.c", function named "runGPUverMMM".
 - ▶ Fill the correct sizes for the "rgblkMMM" kernel.
 - ▶ Open File "Day2-Exercises/MMMult/mmm.cl" and implement kernel named "rgblkMMM".
 - ▶ Fix bugs until it validates! :-)
- Tile size is called RT and not T!

OpenCL equivalences:

- ▶ Global worksize 1: $\lceil M/RT \rceil$; Global worksize 0: $\lceil N/RT^2 \rceil * RT^2$;
- ▶ Local worksize 0: RT^2 ; Local worksize 1: 1;
- ▶ `ii = get_group_id(1) * RT;`
`j = jj + get_global_id(0);`
`locx = get_local_id(0) % RT;`
`locy = get_local_id(0) / RT;`
- ▶ Loops of index i should be normalized $[0..RT-1]$ and unrolled (`#pragma unroll`);
- ▶ Array cs should be one dimensional: `float cs[RT];`
- ▶ `__local real A_loc[RT][RT+1];` for bank conflicts.