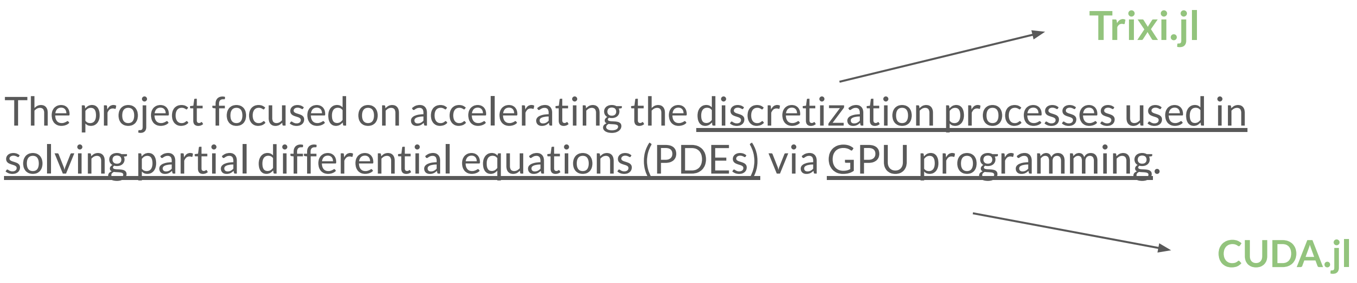
GPU Programming Optimization - GSoC 2023 (Cont’d)

Huiyu Xie
huiyuxie.sde@gmail.com

GSoC 2023 Project Brief Review



Trixi.jl

The project focused on accelerating the discretization processes used in solving partial differential equations (PDEs) via GPU programming.

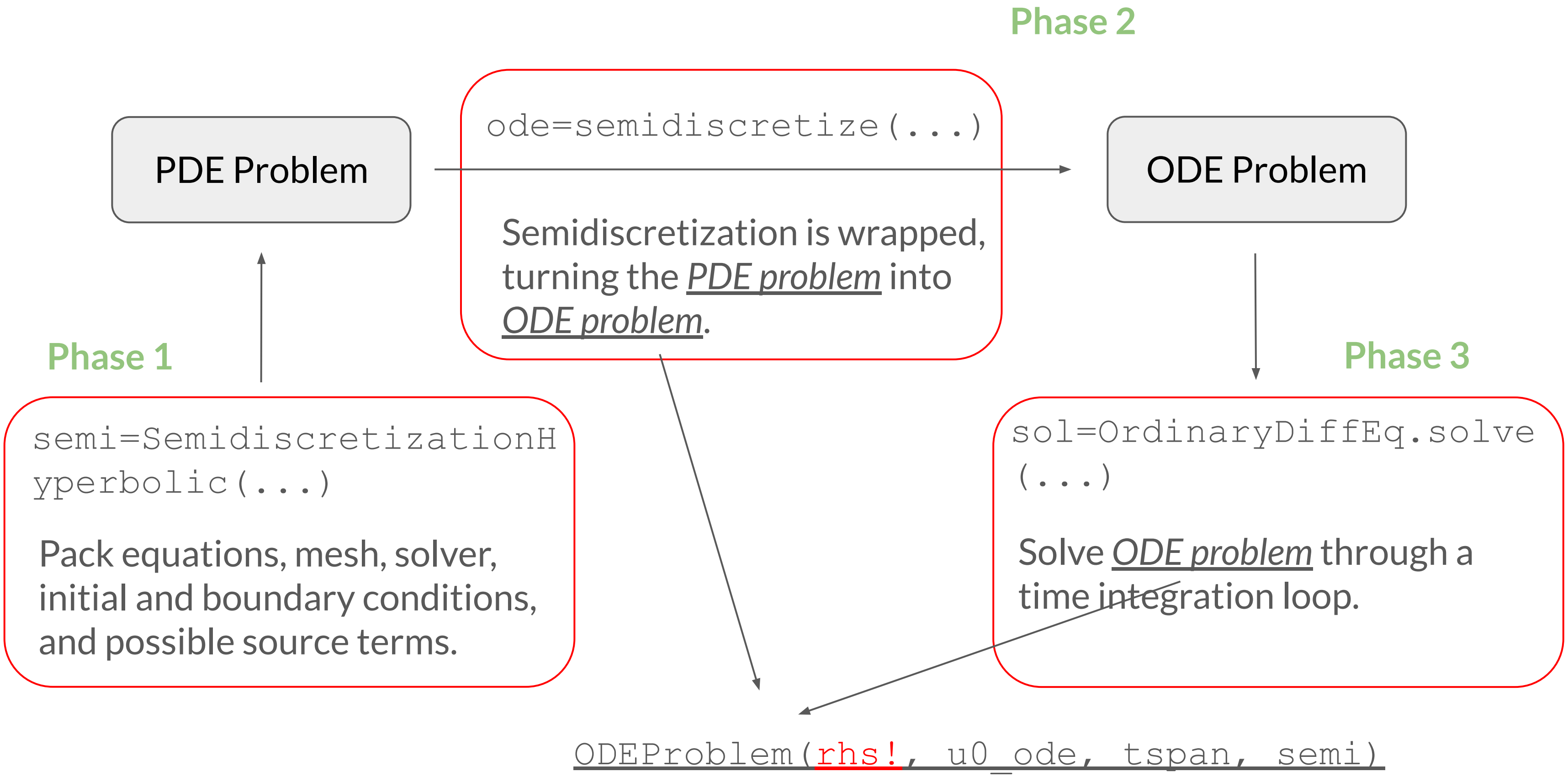
CUDA.jl

Note: The GPU type selected for this project is the *NVIDIA Tesla V100*, which features *5120 CUDA Cores* and *640 Tensor Cores*. The GPU was set up in the cloud on AWS.

Links to some relevant resources:

- Trixi.jl (Docs, GitHub)
- CUDA.jl (Docs, GitHub)
- GSoC 2023 Project (Google, Code, Report)
- CUDA (Docs)
- AWS (Docs)

GSoC 2023 Project Brief Review Cont’d



PDE Problem

Phase 2

```
ode=semidiscretize(...)
```

Semidiscretization is wrapped, turning the *PDE problem* into *ODE problem*.

ODE Problem

Phase 1

```
semi=SemidiscretizationHyperbolic(...)
```

Pack equations, mesh, solver, initial and boundary conditions, and possible source terms.

Phase 3

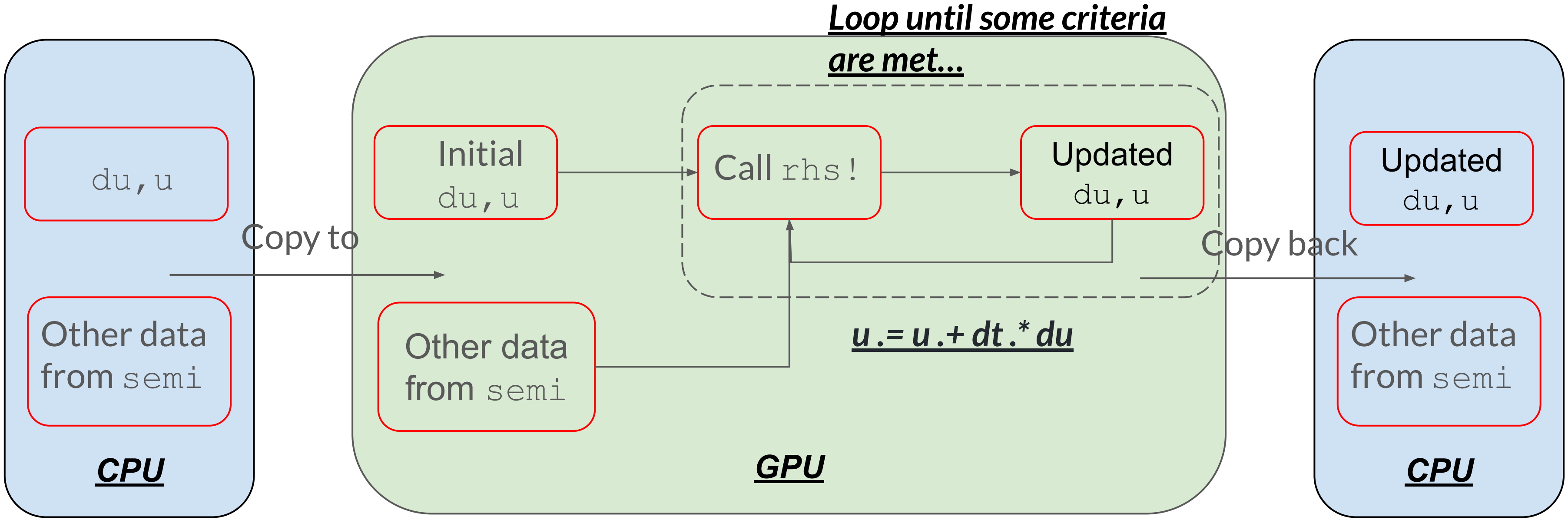
```
sol=OrdinaryDiffEq.solve(...)
```

Solve *ODE problem* through a time integration loop.

```
ODEProblem(rhs!, u0_ode, tspan, semi)
```

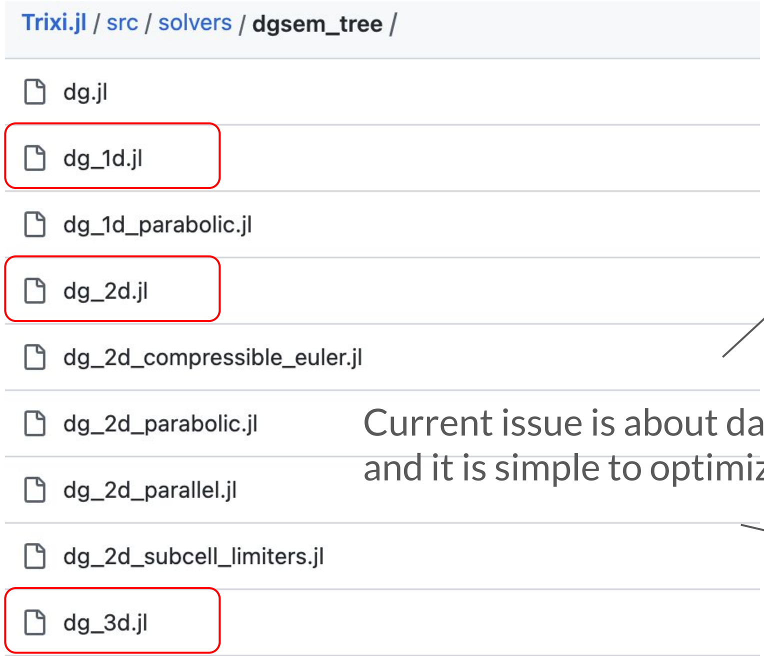
GSoC 2023 Project Brief Review Cont’d

When solving the ODE problem…



GSoC 2023 Project Brief Review Cont’d

Implemented GPU solvers for 1D, 2D and 3D equation problems based on DGSEM with tree mesh.



Trixi.jl / src / solvers / dgsem_tree /

- dg.jl
- dg_1d.jl
- dg_1d_parabolic.jl
- dg_2d.jl
- dg_2d_compressible_euler.jl
- dg_2d_parabolic.jl
- dg_2d_parallel.jl
- dg_2d_subcell_limiters.jl
- dg_3d.jl

Take 2D problem for example!

```
rhs_gpu!(...)

    copy_to_gpu!(...)
    cuda_volume_integral!(...)
    cuda_prolong2interfaces!(...)
    cuda_interface_flux!(...)
    cuda_prolong2boundaries!(...)
    cuda_boundary_flux!(...)
    cuda_surface_integral!(...)
    cuda_jacobian!(...)
    cuda_sources!(...)
    copy_to_cpu!(...)

end
```

Current issue is about data transfer and it is simple to optimize.

Remaining Precision Problems (32-bit v.s. 64-bit)

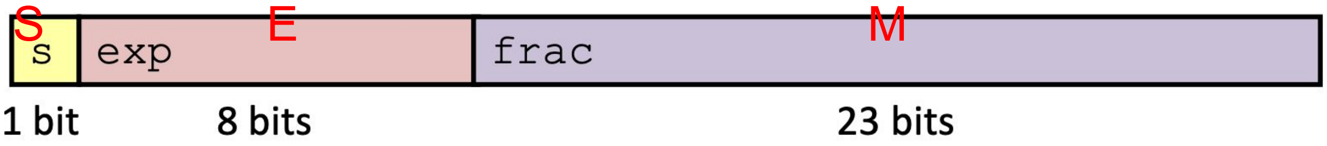
Tested both CPU and GPU implementation using example elixir_advection_basic.jl (2D equation problem), relative errors are shown as below:

```
julia> extrema(du_gpu -du_cpu)./ maximum(abs, du_cpu) # Based on Float32

(-1.707497444828748e-5, 1.7006649665921008e-5)

julia> extrema(du_gpu -du_cpu)./ maximum(abs, du_cpu) # Based on Float64

(-3.3923170749628214e-14, 3.61847154662701e-14)
```

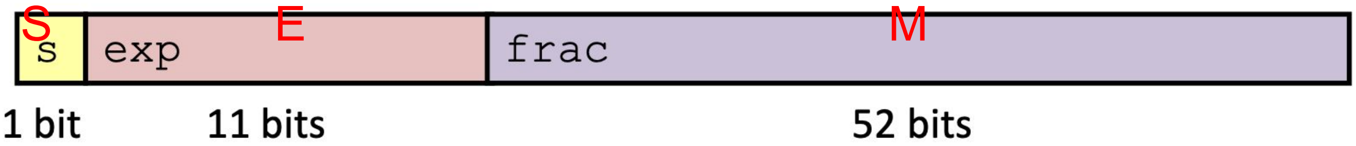
While using `Float64` is much better than `Float32`, both cases caused accuracy issues. Why?

IEEE Floating-Point Standard (IEEE 754)

Single precision (`float/Float32`): 32 bits



Double precision (`double/Float64`): 64 bits



$$\text{bias } = 2^{(n-1)} - 1$$

$$(-1)^S \times 1.M \times 2^{(E-\text{ bias })}$$

$$(-1)^S \times 0.M \times 2^{(-2^{(n-1)}+2)}$$

- Bias: Allows both `positive(+)` and `negative(-)` representation.
- Normalization (1.M): Maximizes number precision.
- Denormalization (0.M): Represents numbers close to `zero`. Use when n-bit exponent is `zero`.

Remaining Precision Problems (32-bit v.s. 64-bit)

Case of 32-bit floating point (`single/Float32`)

- `Float64` is converted to `Float32` (CPU-GPU).
- The 52-bit mantissa (i.e., fraction) of the `Float64` is truncated (or rounded) to fit into the 23-bit mantissa of the `Float32`.

Case of 64-bit floating point (`double/Float64`)

- `Float64` is NOT converted to `Float32` (CPU-GPU).
- But why are there still accuracy errors?

➡ With finite precision, the order of number operation affect the accuracy of the final result. (See one toy example in the next slide!)

Remaining Precision Problems (Example of simple 5-bit system)

```
Sign: 1-bit
Exponent: 2-bit
Fraction: 2-bit
```

Toy example - **5-bit** number representation system

- The smallest positive number can be represented (i.e., precision) is `1*2^(-2)=0.25` (Can be easily verified!), and thus any number less than `0.25` will underflow to zero .
- Consider the following two calculations:

(1) `1.00*2^0 + 1.00*2^0 + 1.00*2^(-2) + 1.00*2^(-2)=`
`1.00*2^1 + 1.00*2^(-2) + 1.00*2^(-2) = 1.00*2^1`

(2) `(1.00*2^0 + 1.00*2^0) + (1.00*2^(-2) + 1.00*2^(-2))=`
`1.01*2^1`

Both (1) and (2) are using binary!

Remaining Precision Problems (How to deal with?)

Trade-off: Speed v.s. Precision

Reasons have been identified!

- GPU programming (i.e., parallel programming) adopt parallel computing strategy, that is, the order of operations may not be sequential (like summing up an array of numbers).

How to deal with this issue?

(1) Easy approach: Sorting is frequently used in parallel numerical algorithms

- Group numerical values close to each other in the same group.
- Perform operation (e.g., addition) in each group, more likely to get accurate result.
- Sign of numbers should be taken into account.

(2) Advanced approach: Kahan's Summation Algorithm (i.e., Compensated Summation Algorithm) - Only for accurate summation.

Remaining Precision Problems (Q&A)

Typically, there is a trade-off between speed and precision

- Use `Float32` /`float`, low precision but high speed
- Use `Float64` /`double`, high precision but low speed -

  Further nested trade-off problem (can be applied to both 32-bit and 64-bit)

  - Apply algorithm, improve precision but affect speed
  - No algorithm, keep errors and speed

Which one is more preferable, and what level of accuracy error is acceptable? (Given the context of solving PDE problems.)

Relative Error Formula →

$$\boxed{\delta = \left| \frac{v_A - v_E}{v_E} \right| \cdot 100\%}$$

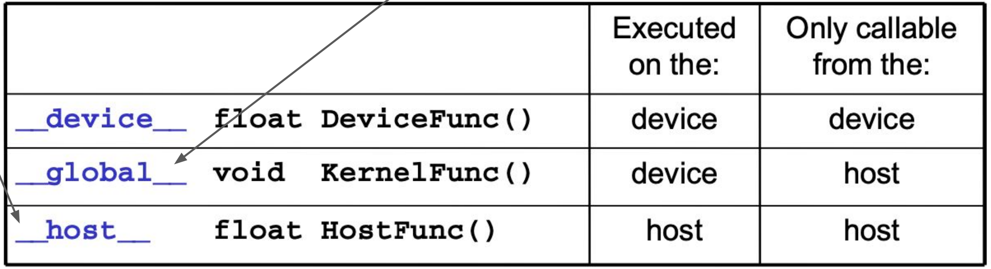
Optimization of Existing CUDA Kernels (Some Reviews)

Recall what I have said in the GSoC project report…

- Implement custom CUDA kernels instead of using existing types (e.g., `CuArray` type).
- Avoid using conditional statement (e.g., `if-then-else` branches) in CUDA kernels.
- Decrease the number of CUDA kernel calls (i.e., calls to `__global__` functions) within a certain function (i.e., `__host__` function).

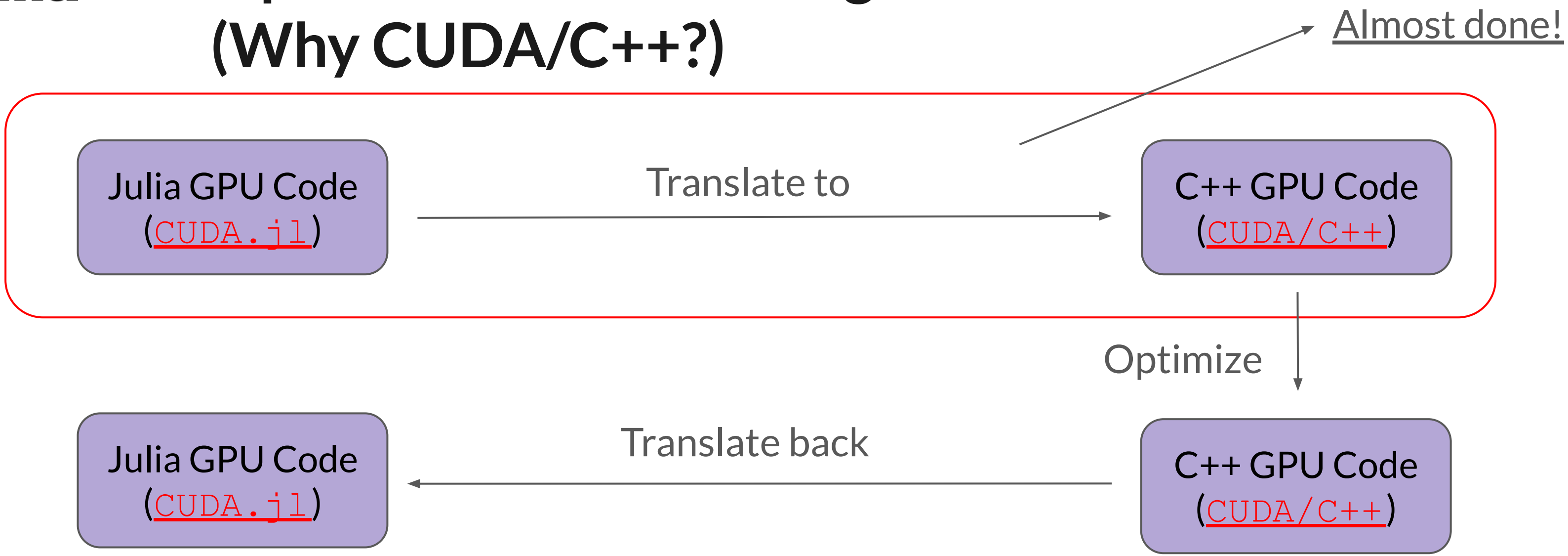
But these are not enough…

- We have to further optimize CUDA kernels!



	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

Optimization of Existing CUDA Kernels (Why CUDA/C++?)



Choose to optimize existing kernels based on CUDA/C++. Why?

- `CUDA/C++` is more mature than `cuda.jl` (i.e., more runtime APIs).
- `CUDA/C++` has more documentations and examples.
- Existing optimization examples are demonstrated with `CUDA/C++`.

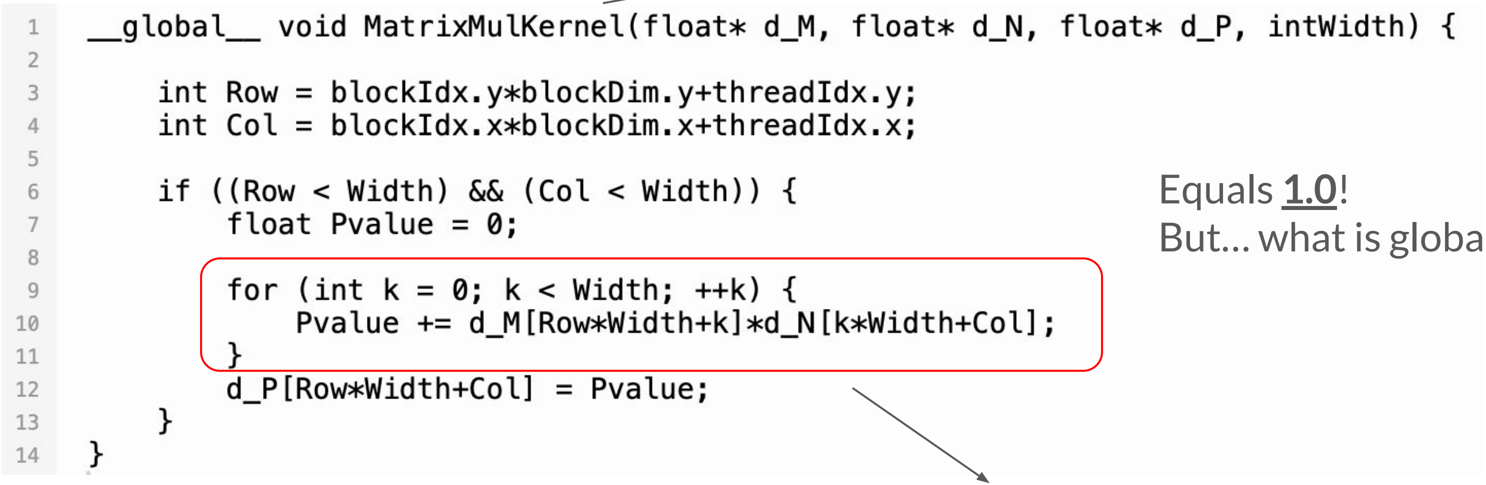
C++: 0-indexed and row-based

Julia: 1-indexed and column-based

Compute-to-Global-Memory-Access Ratio (Kernel Optimization)

Let’s look at this example!

Classic & Simple:
Square matrix multiplication



```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, intWidth) {

    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;

        for (int k = 0; k < Width; ++k) {
            Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
        }
        d_P[Row*Width+Col] = Pvalue;
    }
}
```

Equals **1.0**!
But… what is global memory?

Compute-to-Global-Memory-Access Ratio = #(times of floating-point calculation performed) / #(times of accessing to global memory) within a program region

GPU Memory Architecture/Layout (Kernel Optimization)

Good Performance

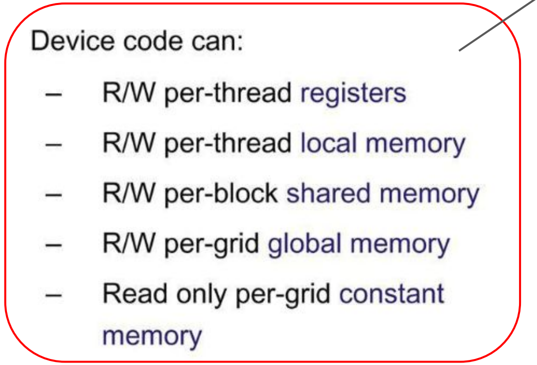
Registers - Thread scope - Fastest - Very limited

Shared memory - Block scope - Fast - Limited

Global memory - Grid scope (GPU device) - Slow - Not limited

Bad Performance

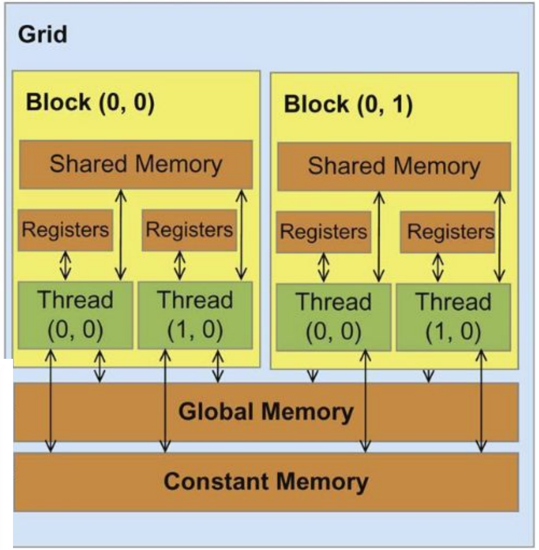
And `cudaMalloc()`

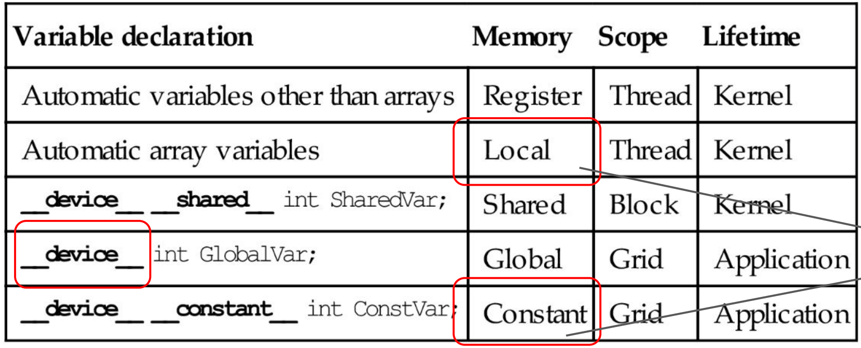


Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Complicated... but...



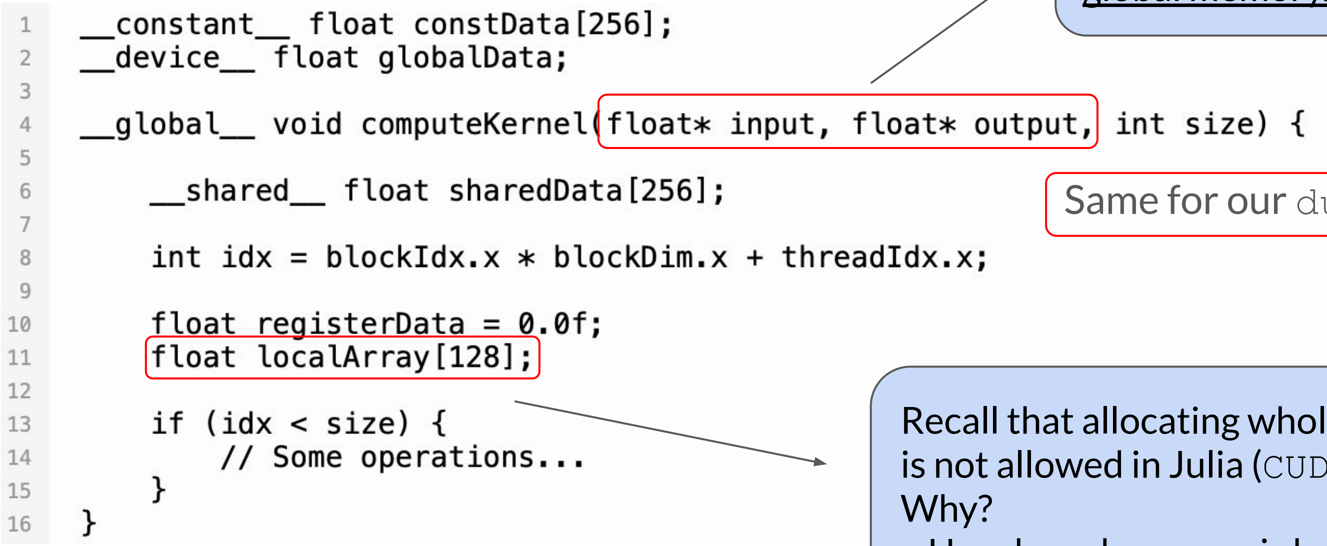


Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
__device__ __shared__ int SharedVar;	Shared	Block	Kernel
__device__ int GlobalVar;	Global	Grid	Application
__device__ __constant__ int ConstVar;	Constant	Grid	Application

Reside in global memory!

GPU Memory Architecture/Layout Cont’d (Kernel Optimization)

Let’s use this example to figure out memory types!



```
__constant__ float constData[256];
__device__ float globalData;

__global__ void computeKernel(float* input, float* output, int size) {

    __shared__ float sharedData[256];

    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    float registerData = 0.0f;
    float localArray[128];

    if (idx < size) {
        // Some operations...
    }
}
```

Normally, allocated by calling `cudaMalloc()` , thus they are in <u>global memory.</u>

Same for our `du` and `u` variables!

Recall that allocating whole arrays is not allowed in Julia (`CUDA.jl`). Why?
 - Use shared memory is better.

Optimization of Existing CUDA Kernels (Insights from Ratio…)

Check the formula again:

Compute-to-Global-Memory-Access Ratio = #(times of floating-point calculation performed) / #(times of accessing to global memory) within a program region

- Higher ratio implies better performance
- Lower ratio implies worse performance

Have to increase the ratio! But how?

Generally there are two ways (straightforward!)

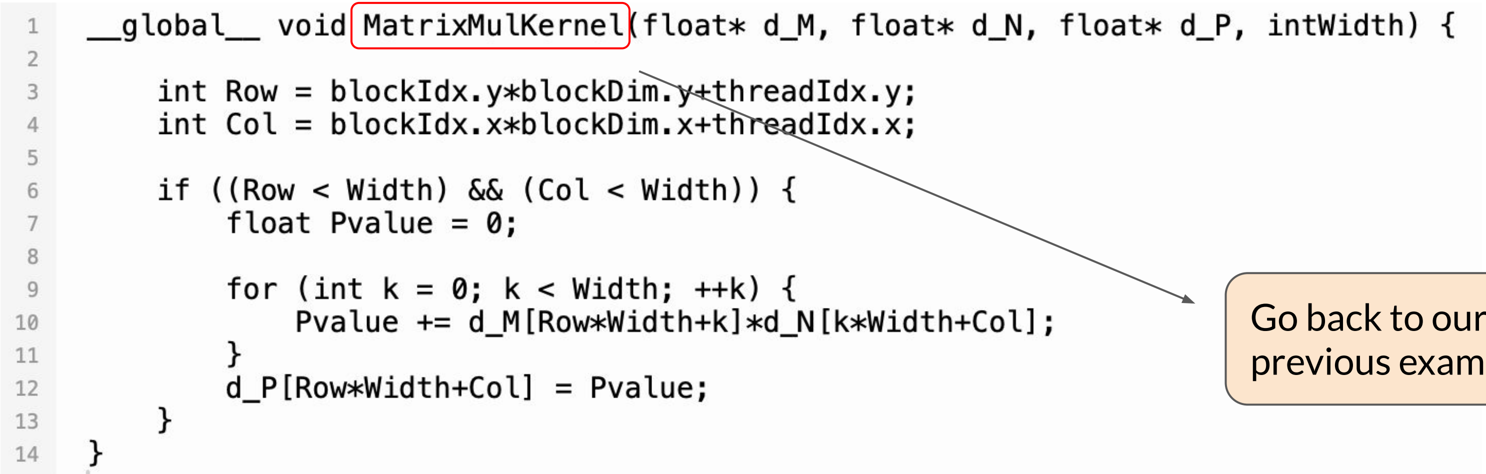
- Increase numerator (But almost fixed…)
- Decrease denominator - (1) Use registers - (2) Use shared memory - (3) Keep in global memory

Common approach by programmers!

Optimization of Existing CUDA Kernels (Insights from Ratio…)

How to decrease denominator (i.e., decrease the times of accessing global memory):

- Use registers (Very limited! Cannot allocate large amounts of data…)
- Use shared memory (Viable!)
- Keep in global memory (Viable!)



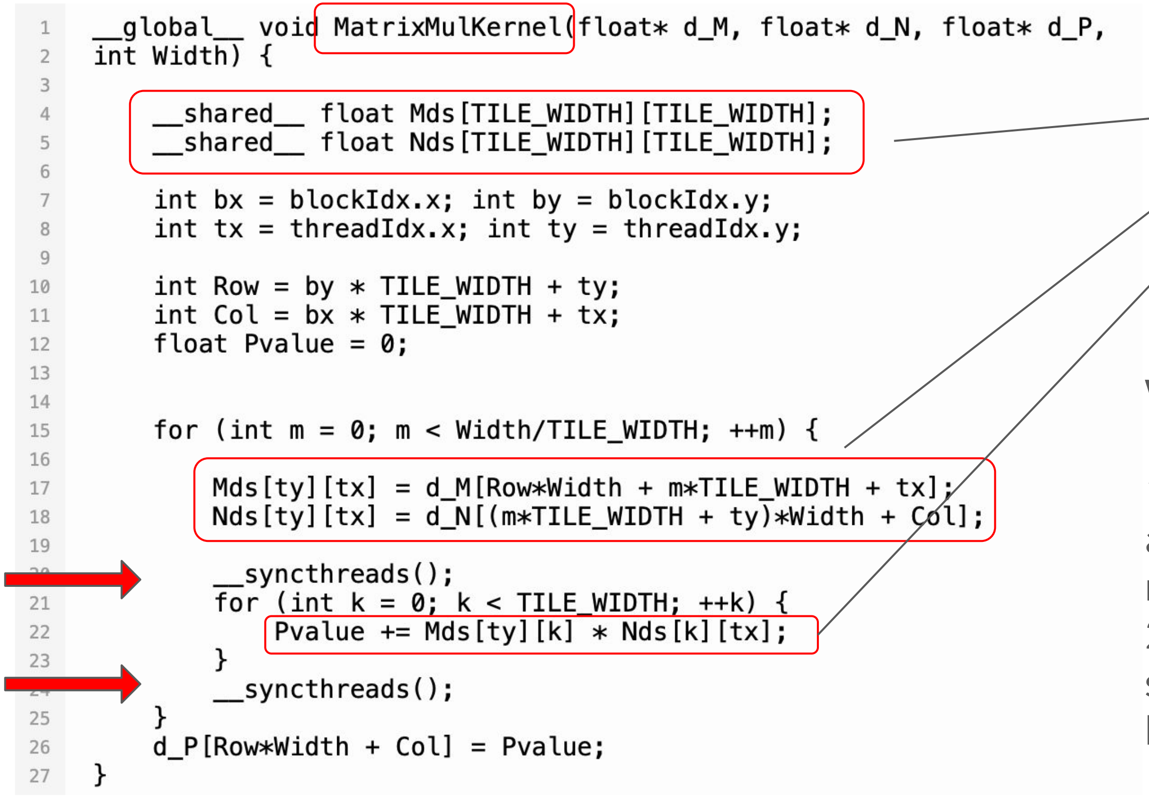
```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, intWidth) {

    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;

        for (int k = 0; k < Width; ++k) {
            Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
        }
        d_P[Row*Width+Col] = Pvalue;
    }
}
```

Go back to our previous example!

Optimization of Existing CUDA Kernels (Shared Memory and Tiling)

Still confused?
No worries!

```
#define TILE_WIDTH …
```



```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
int Width) {

    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float Pvalue = 0;


    for (int m = 0; m < Width/TILE_WIDTH; ++m) {

        Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
        Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];

        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; ++k) {
            Pvalue += Mds[ty][k] * Nds[k][tx];
        }
        __syncthreads();
    }
    d_P[Row*Width + Col] = Pvalue;
}
```

(1) Declare shared memory
(2) Copy from global to shared
(3) Access to shared memory

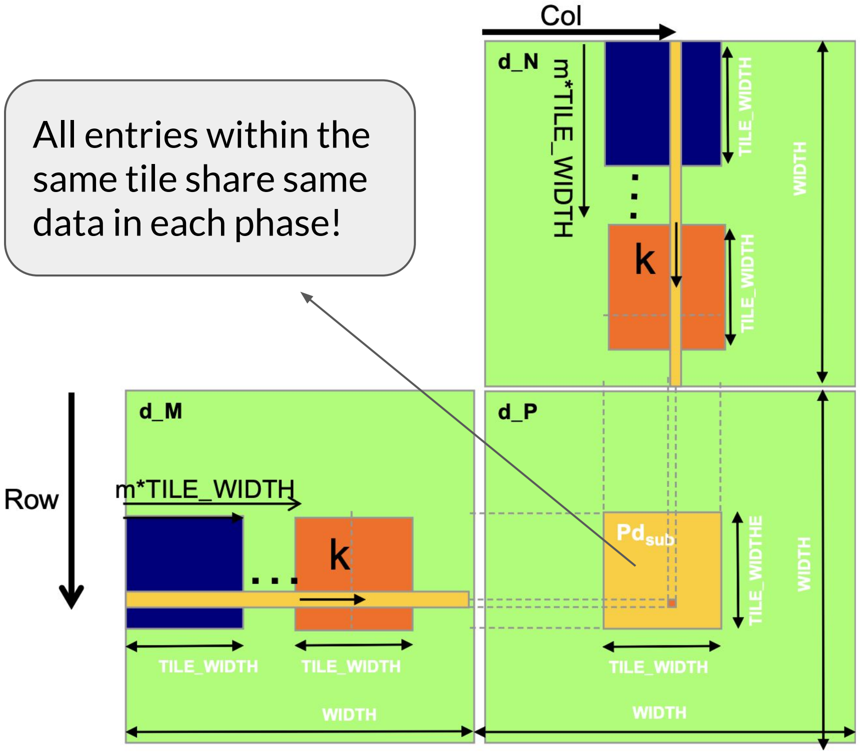
What is `__syncthreads()`?

1st sync: Make sure all necessary data are loaded into a single tile (of shared memory).
2nd sync: Make sure all data from a single tile (of shared memory) have been used before being replacing.

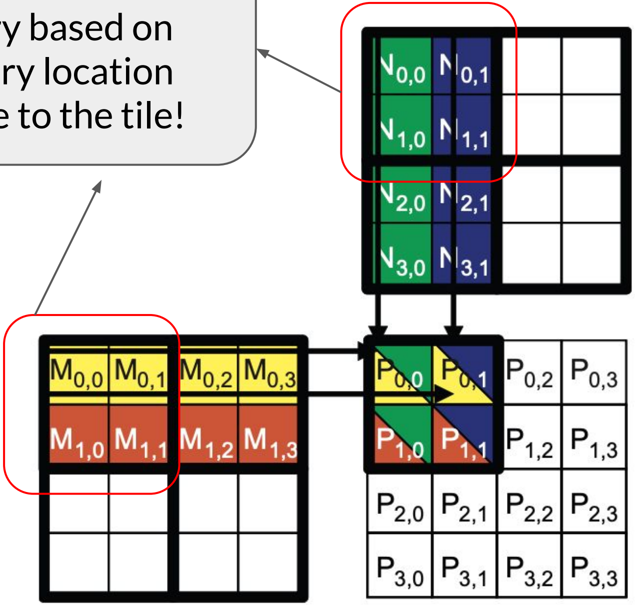
Optimization of Existing CUDA Kernels (Shared Memory and Tiling)

Do some visualization!

All entries within the same tile share same data in each phase!

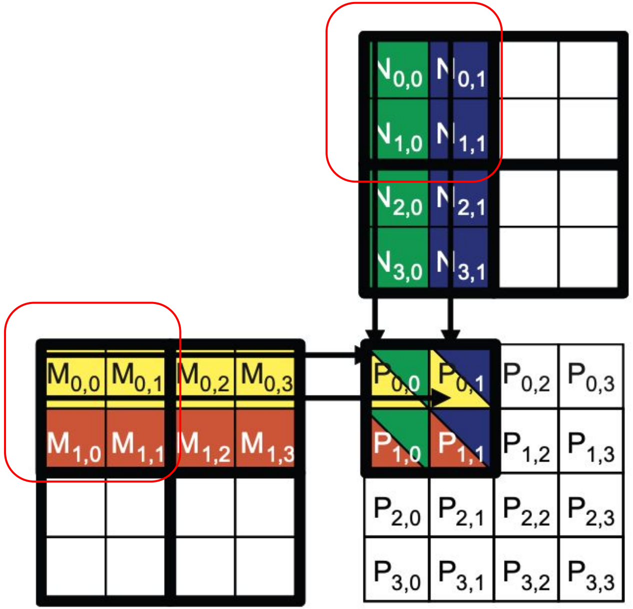


Load data to shared memory based on the entry location relative to the tile!



Optimization of Existing CUDA Kernels (Shared Memory and Tiling)

Do some calculation!



TILE_WIDTH * #(new times of accessing global memory)
= #(original times of accessing global memory)

#(new times of accessing global memory)
= #(original times of accessing global memory)/TILE_WIDTH

Global memory accessing be reduced by times!
But what is the best size for tiling?

Optimization of Existing CUDA Kernels (About Size for Tiling...)

Within the limit, the larger the better? Not really... Need performance tuning.

Consider the following extreme cases:

- If size for tiling is too large, we will exceed the limit of shared memory per block.
- If size for tiling is too small, we will have few performance improvement.

The best size actually depends on the matrix size and the GPU architecture. And the tile size often matches block size.

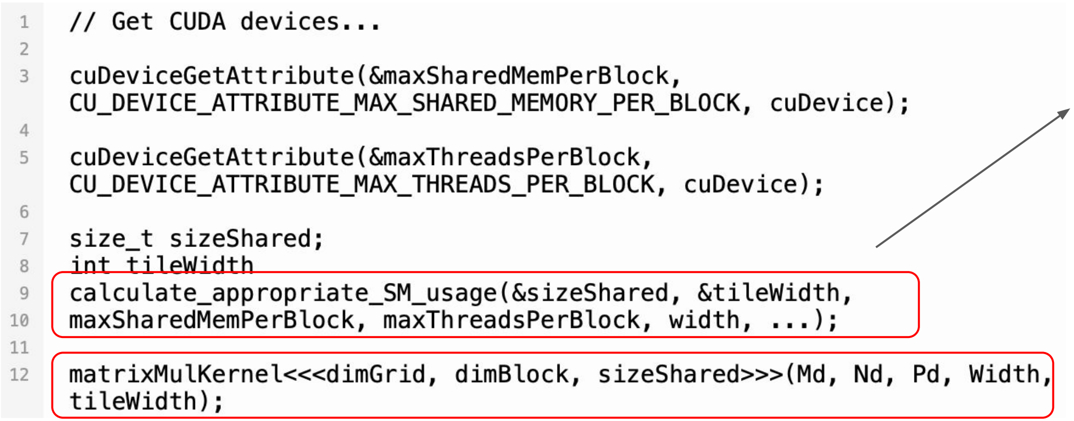
Fortunately, programmers can dynamically allocate shared memory during runtime!

```
cuDeviceGetAttribute()
CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK
CU_DEVICE_ATTRIBUTE_WARP_SIZE
CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK
...
```

Optimization of Existing CUDA Kernels (Dynamic allocation of Shared Memory)

Case dependent!

Let’s start with kernel configuration and execution.



```
// Get CUDA devices...

cuDeviceGetAttribute(&maxSharedMemPerBlock,
CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK, cuDevice);

cuDeviceGetAttribute(&maxThreadsPerBlock,
CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK, cuDevice);

size_t sizeShared;
int tileWidth
calculate_appropriate_SM_usage(&sizeShared, &tileWidth,
maxSharedMemPerBlock, maxThreadsPerBlock, width, ...);

matrixMulKernel<<<dimGrid, dimBlock, sizeShared>>>(Md, Nd, Pd, Width,
tileWidth);
```

Get the <u>tile size</u> (and thus <u>the size of shared memory</u>) based on some constraints:

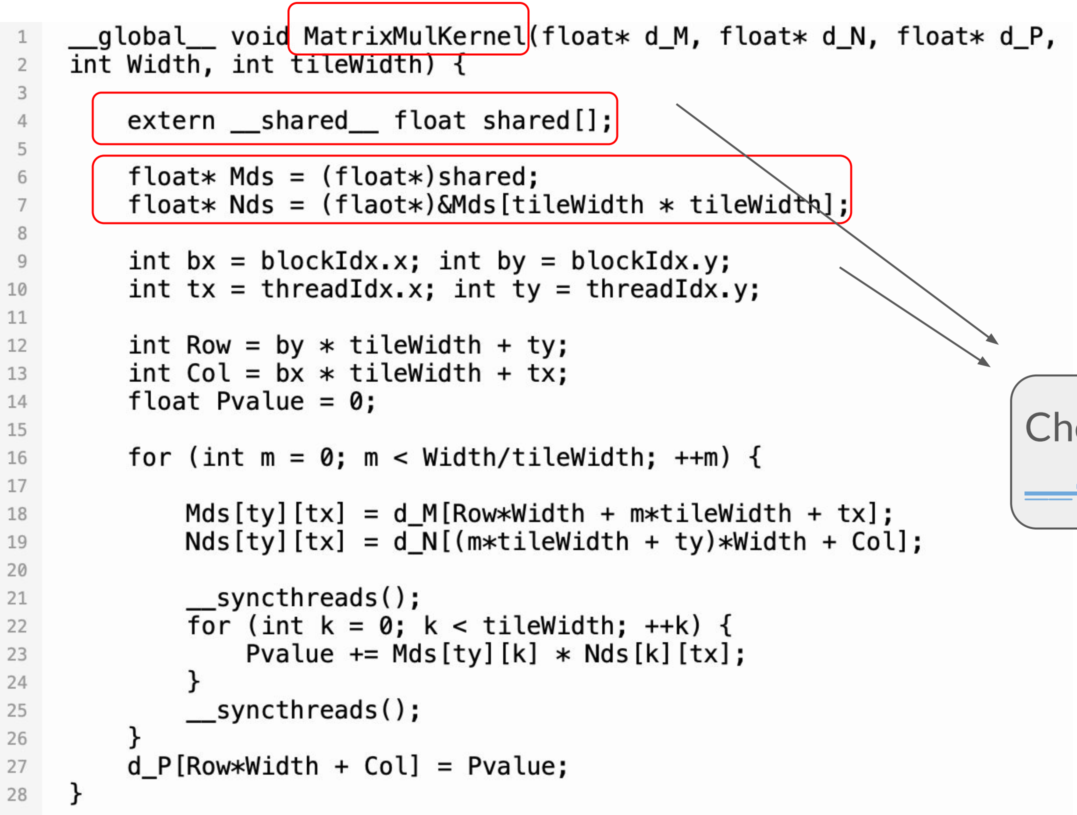
- Tile memory cannot exceed `maxSharedMemPerBlock`.
- Block size cannot exceed `maxThreadsPerBlock`.
- Tile size (i.e., `tileWidth`) should be smaller than matrix size (i.e., `width`).

What else...?

Launch up the kernel by passing the size of shared memory...

Warning: Shared memory configurator <u>should not be too complex</u>! Bad performance again!

Optimization of Existing CUDA Kernels (Dynamic allocation of Shared Memory)



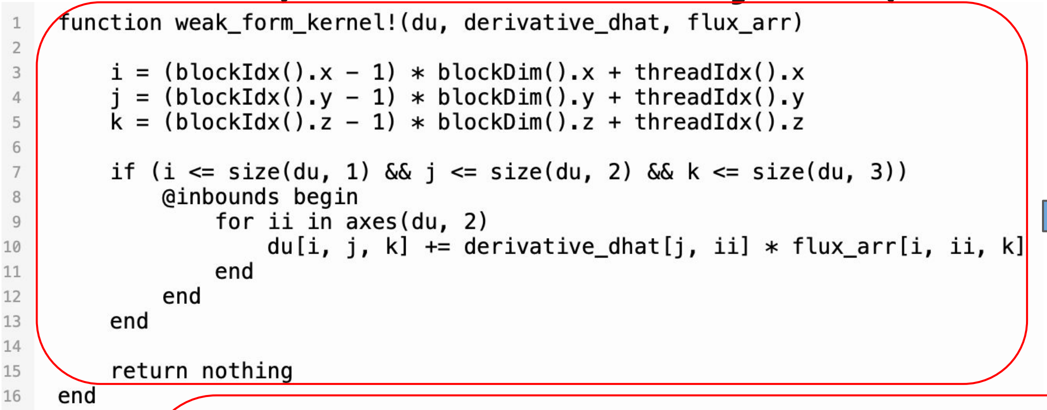
```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
int Width, int tileWidth) {

    extern __shared__ float shared[];

    float* Mds = (float*)shared;
    float* Nds = (flaot*)&Mds[tileWidth * tileWidth];

    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * tileWidth + ty;
    int Col = bx * tileWidth + tx;
    float Pvalue = 0;

    for (int m = 0; m < Width/tileWidth; ++m) {

        Mds[ty][tx] = d_M[Row*Width + m*tileWidth + tx];
        Nds[ty][tx] = d_N[(m*tileWidth + ty)*Width + Col];

        __syncthreads();
        for (int k = 0; k < tileWidth; ++k) {
            Pvalue += Mds[ty][k] * Nds[k][tx];
        }
        __syncthreads();
    }
    d_P[Row*Width + Col] = Pvalue;
}
```

Then go quick with the kernel implementation.

Check CUDA programming __shared__ references here!

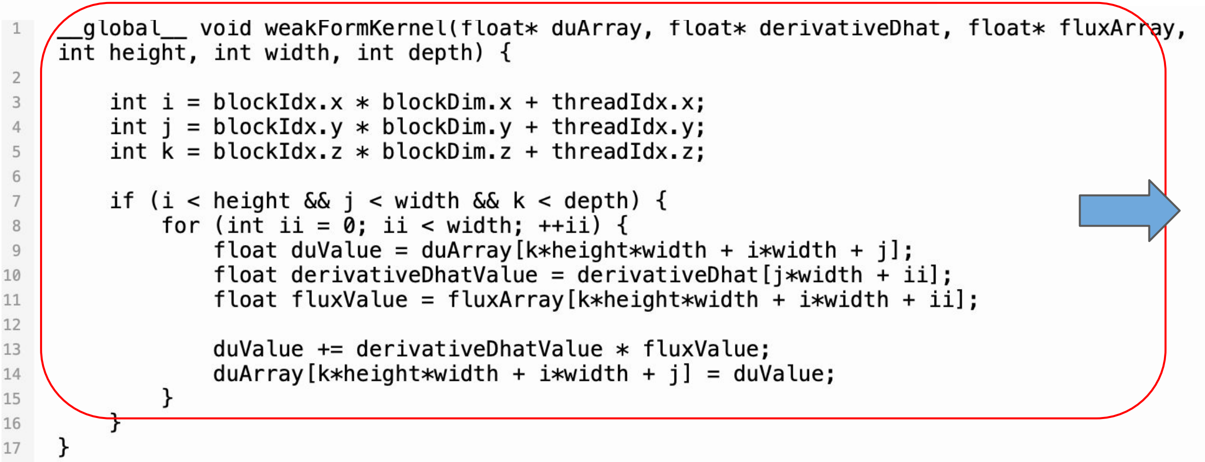
Optimization of Existing CUDA Kernels (Back to Our Project...)

Let's take `weak_form_kernel!()` from 1D problem as an example!



```julia
function weak_form_kernel!(du, derivative_dhat, flux_arr)

    i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    j = (blockIdx().y - 1) * blockDim().y + threadIdx().y
    k = (blockIdx().z - 1) * blockDim().z + threadIdx().z

    if (i <= size(du, 1) && j <= size(du, 2) && k <= size(du, 3))
        @inbounds begin
            for ii in axes(du, 2)
                du[i, j, k] += derivative_dhat[j, ii] * flux_arr[i, ii, k]
            end
        end
    end

    return nothing
end
```

Recall: `Julia` is 1-indexed and column-based.



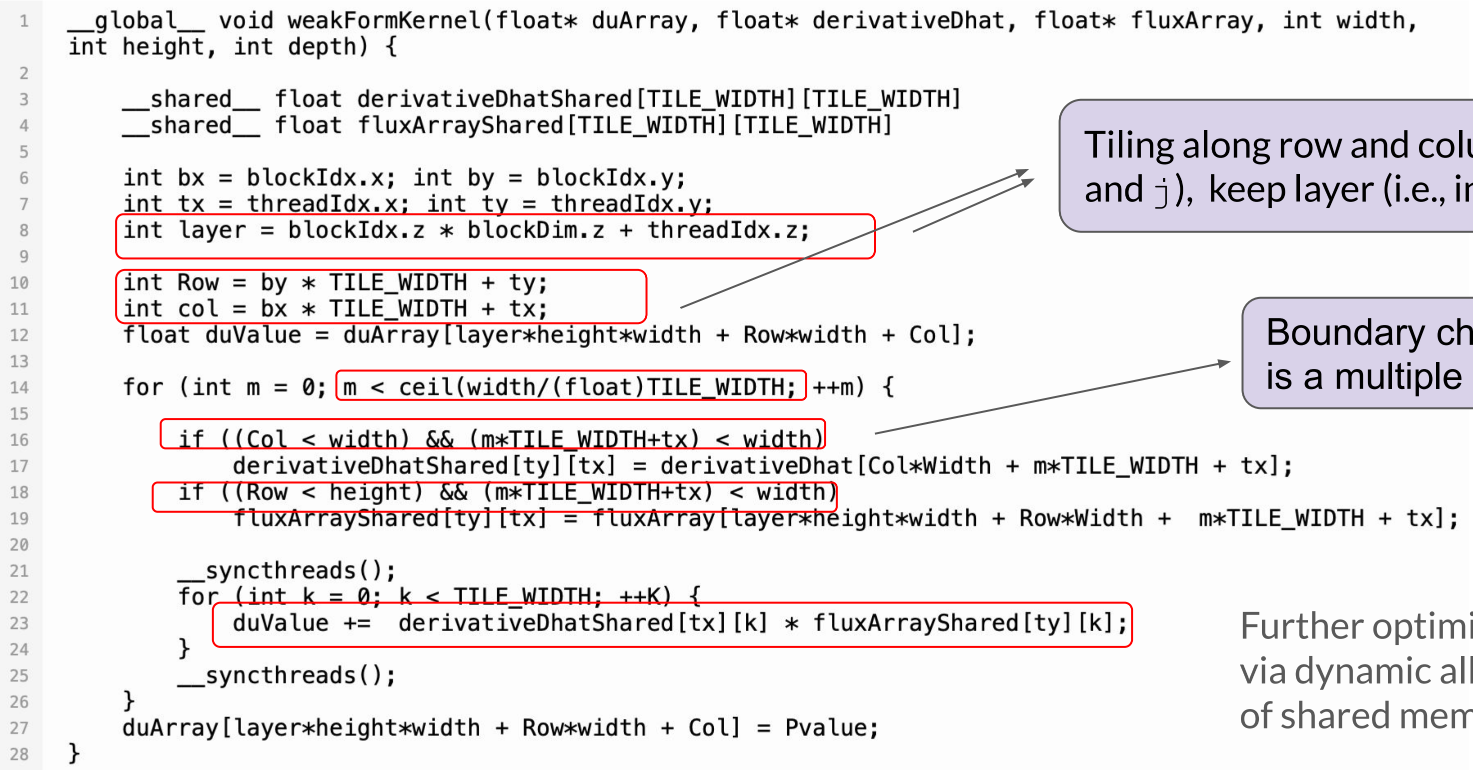
```cpp
__global__ void weakFormKernel(float* duArray, float* derivativeDhat, float* fluxArray,
int height, int width, int depth) {

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

    if (i < height && j < width && k < depth) {
        for (int ii = 0; ii < width; ++ii) {
            float duValue = duArray[k*height*width + i*width + j];
            float derivativeDhatValue = derivativeDhat[j*width + ii];
            float fluxValue = fluxArray[k*height*width + i*width + ii];

            duValue += derivativeDhatValue * fluxValue;
            duArray[k*height*width + i*width + j] = duValue;
        }
    }
}
```

Recall: `C++` is 0-indexed and row-based.

Optimization of Existing CUDA Kernels (Back to Our Project…)

Do some optimization (tiling via shared memory)!

```
#define TILE_WIDTH …
```



```
__global__ void weakFormKernel(float* duArray, float* derivativeDhat, float* fluxArray, int width,
int height, int depth) {

    __shared__ float derivativeDhatShared[TILE_WIDTH][TILE_WIDTH]
    __shared__ float fluxArrayShared[TILE_WIDTH][TILE_WIDTH]

    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    int layer = blockIdx.z * blockDim.z + threadIdx.z;

    int Row = by * TILE_WIDTH + ty;
    int col = bx * TILE_WIDTH + tx;
    float duValue = duArray[layer*height*width + Row*width + Col];

    for (int m = 0; m < ceil(width/(float)TILE_WIDTH; ++m) {

        if ((Col < width) && (m*TILE_WIDTH+tx) < width)
            derivativeDhatShared[ty][tx] = derivativeDhat[Col*Width + m*TILE_WIDTH + tx];
        if ((Row < height) && (m*TILE_WIDTH+tx) < width)
            fluxArrayShared[ty][tx] = fluxArray[layer*height*width + Row*Width +  m*TILE_WIDTH + tx];

        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; ++K) {
            duValue +=  derivativeDhatShared[tx][k] * fluxArrayShared[ty][k];
        }
        __syncthreads();
    }
    duArray[layer*height*width + Row*width + Col] = Pvalue;
}
```

Tiling along row and column (i.e., index `i` and `j`), keep layer (i.e., index `k`) intact.

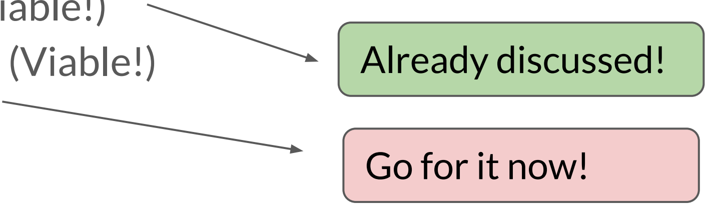
Boundary check! Matrix size is a multiple of tile size?

Further optimization via dynamic allocation of shared memory…

Optimization of Existing CUDA Kernels (Memory Coalescing)

How to decrease denominator (i.e., decrease the times of accessing global memory):

- Use registers (Very limited! Cannot allocate large amounts of data...)
- Use shared memory (Viable!)
- Keep in global memory (Viable!)



Already discussed!

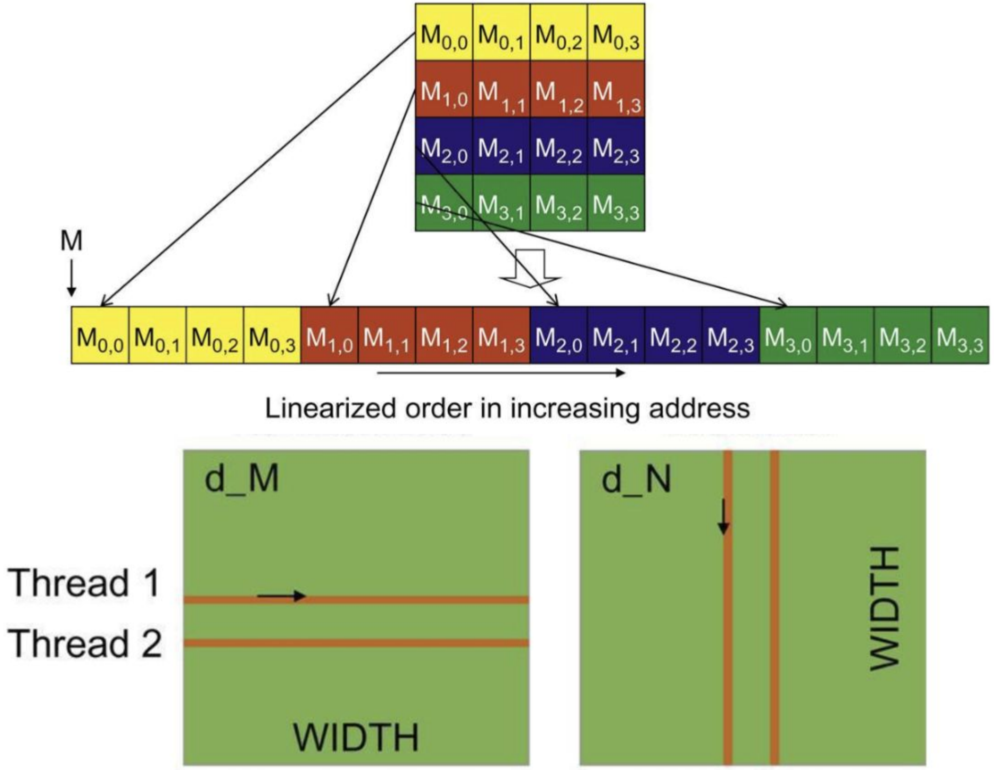
Go for it now!

Good prerequisite for memory coalescing!

The global memory of a CUDA device is implemented with DRAMs (Dynamic Random-Access Memory).

Each time a DRAM location is accessed, a range of consecutive locations that includes the requested location are actually accessed (i.e., DRAM burst).

Optimization of Existing CUDA Kernels (Memory Coalescing)



$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
Linearized order in increasing address
d_M
Thread 1
Thread 2
WIDTH
d_N
WIDTH

Array elements in `C(C++)` and `CUDA` are placed into linearly addressed memory space based on row-major convention.

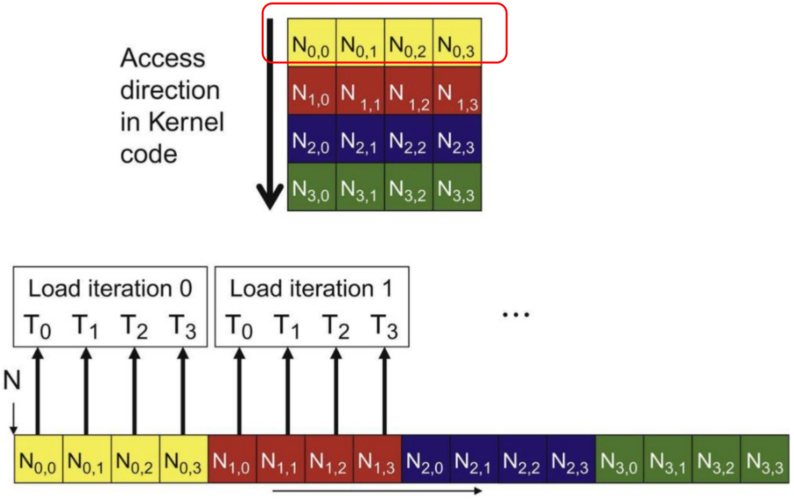
What about `Julia` and `CUDA.jl`?

Column-major!

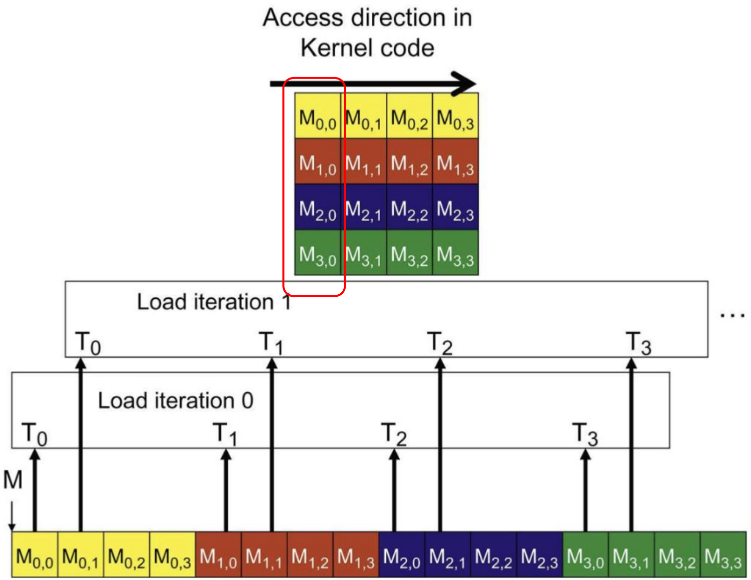
- Left is coalesced in `CUDA.jl` (`Julia`).
- Right is coalesced in `CUDA` (`C++`).

Why?

Optimization of Existing CUDA Kernels (Memory Coalescing Cont'd)


Access direction in Kernel code
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
Load iteration 0
T_0 T_1 T_2 T_3
Load iteration 1
T_0 T_1 T_2 T_3
...
N
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$ $N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$ $N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$ $N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$

In a row-based layout, consecutive data is accessed at each iteration.


Access direction in Kernel code
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
Load iteration 1
T_0 T_1 T_2 T_3
...
Load iteration 0
T_0 T_1 T_2 T_3
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

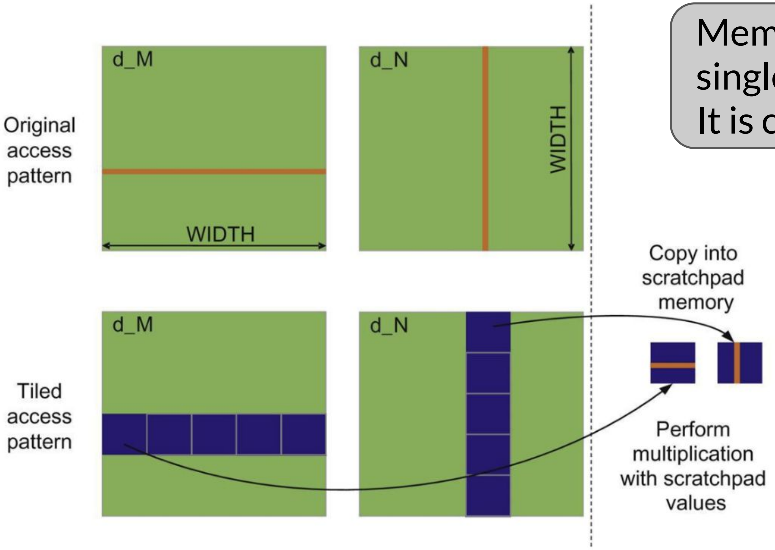
In a column-based layout, consecutive data is accessed at each iteration.

Optimization of Existing CUDA Kernels (Corner Turning)

Tiling algorithm again!

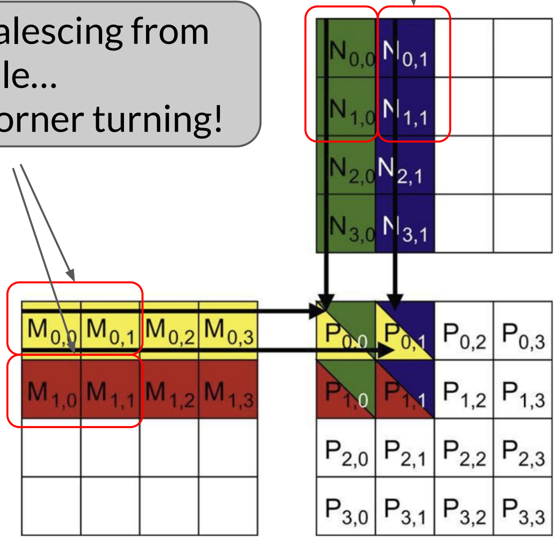
What about `Julia/CUDA.jl`?

In the example of matrix multiplication using `CUDA/C++`, the row interaction is necessary. How can we achieve memory coalescing…?



Original access pattern
d_M
WIDTH
d_N
WIDTH
Tiled access pattern
d_M
d_N
Copy into scratchpad memory
Perform multiplication with scratchpad values

Memory coalescing from single one tile…
It is called corner turning!



Optimization of Existing CUDA Kernels (Short Summary)

To achieve a high compute-to-global-memory-access ratio, we adopt **tiling algorithm**, which bas basically has two advantages:

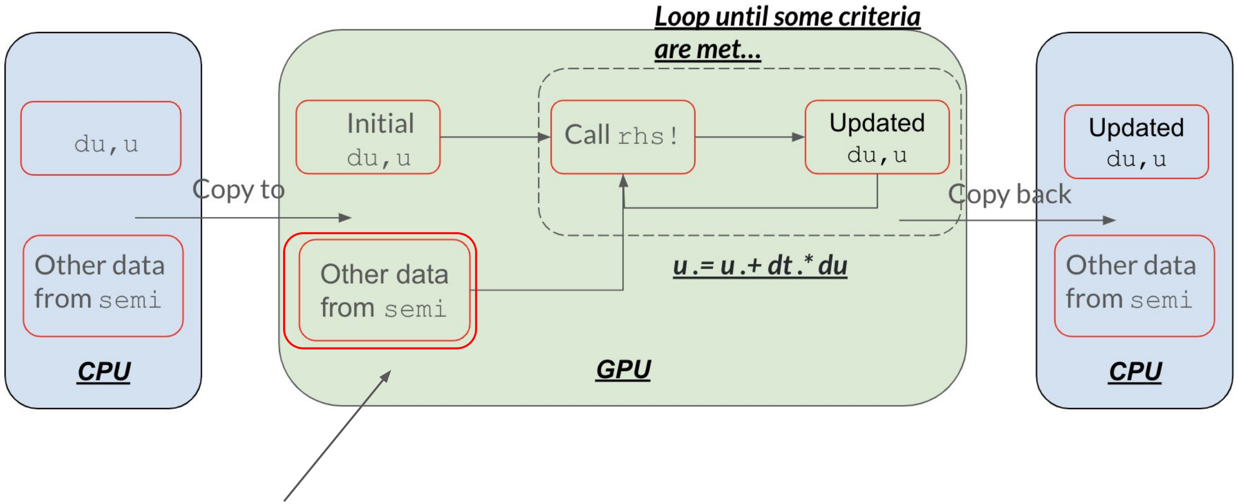
- The number of global memory loads is reduced due to the reuse of data in the shared memory.
- The renaming global memory loads are coalesced so the DRAM bandwidth utilization is further improved.

What else can we do...?

Therefore, the main optimization strategy is to **apply the tiling algorithm** to our existing CUDA kernels.

Note that once the data is in the shared memory, they can be accessed either on a row basis or a column basis with much less performance variation because it is *high-speed on-chip memory*.

Optimization of Existing CUDA Kernels (About Sparse Matrix Computation...)



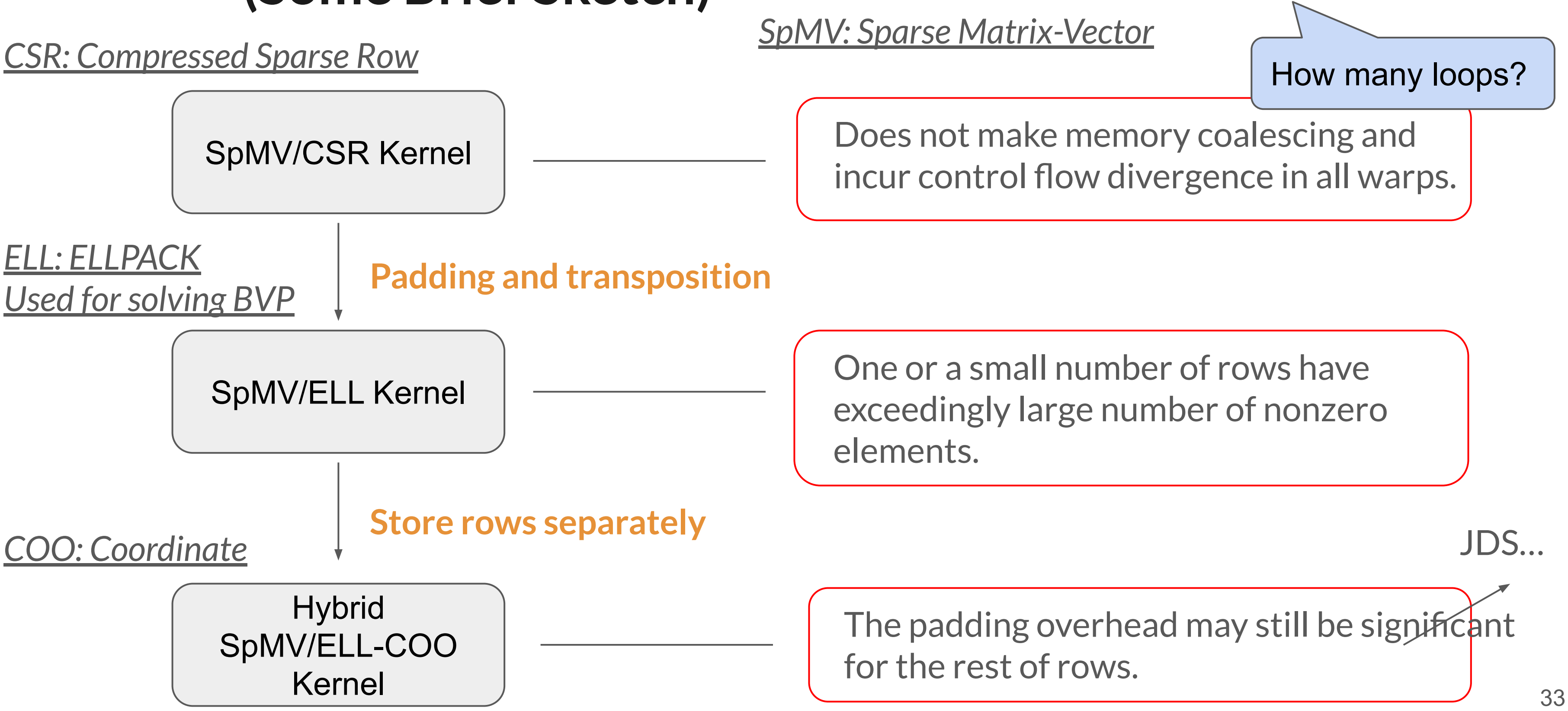
When solving the ODE problem, some data from `semi` (like `cache.xx.xx`) are sparse and remain unchanged during iteration (time integration).

A waste of memory bandwidth if zeros are doing nothing!

Sparse Matrix Parallel Computation (Some Brief Sketch)

Trade-off: Format transformation overhead v.s. Runtime speed

SpMV: Sparse Matrix-Vector



How many loops?

CSR: Compressed Sparse Row

SpMV/CSR Kernel

Does not make memory coalescing and incur control flow divergence in all warps.

ELL: ELLPACK
Used for solving BVP

Padding and transposition

SpMV/ELL Kernel

One or a small number of rows have exceedingly large number of nonzero elements.

COO: Coordinate

Store rows separately

Hybrid SpMV/ELL-COO Kernel

The padding overhead may still be significant for the rest of rows.

JDS...

References

1. Kirk, D. B., & Hwu, W. W. (2016). *Programming Massively Parallel Processors: A Hands-on Approach (3rd ed.)*. Elsevier.
2. NVIDIA. (n.d.). *CUDA C Programming Guide*. Retrieved December 18, 2023, from https://docs.nvidia.com/cuda/cuda-c-programming-guide/contents.html
3. Trixi.jl Developers. (n.d.). *Trixi.jl Documentation*. Retrieved December 18, from https://trixi-framework.github.io/Trixi.jl/stable/
4. CUDA.jl Developers. (n.d.). *CUDA.jl Documentation*. Retrieved December 18, from https://cuda.juliagpu.org/stable/
5. Amazon Web Services, Inc. (n.d.). AWS Documentation. Retrieved December 18, from https://docs.aws.amazon.com/

Q&A

We have discussed a portion of the future work, focusing on the kernel optimization aspect.

Are there any questions so far?