Programming GPUs with PyCUDA and Julia

1. Data Parallelism
   - the programming model
   - host (CPU) and device (GPU)
   - matrix matrix multiplication
2. PyCUDA
   - about PyCUDA
   - using PyCUDA for matrix matrix multiplication
   - the kernel explained
3. GPU Programming in Julia
   - about CUDA.jl
   - a first kernel
   - vendor agnostic GPU computing

MCS 572 Lecture 17
Introduction to Supercomputing
Jan Verschelde, 4 October 2024

Programming GPUs with PyCUDA and Julia

1. Data Parallelism
   - the programming model
   - host (CPU) and device (GPU)
   - matrix matrix multiplication
2. PyCUDA
   - about PyCUDA
   - using PyCUDA for matrix matrix multiplication
   - the kernel explained
3. GPU Programming in Julia
   - about CUDA.jl
   - a first kernel
   - vendor agnostic GPU computing

the programming model

Programming model: Single Instruction Multiple Data (SIMD).

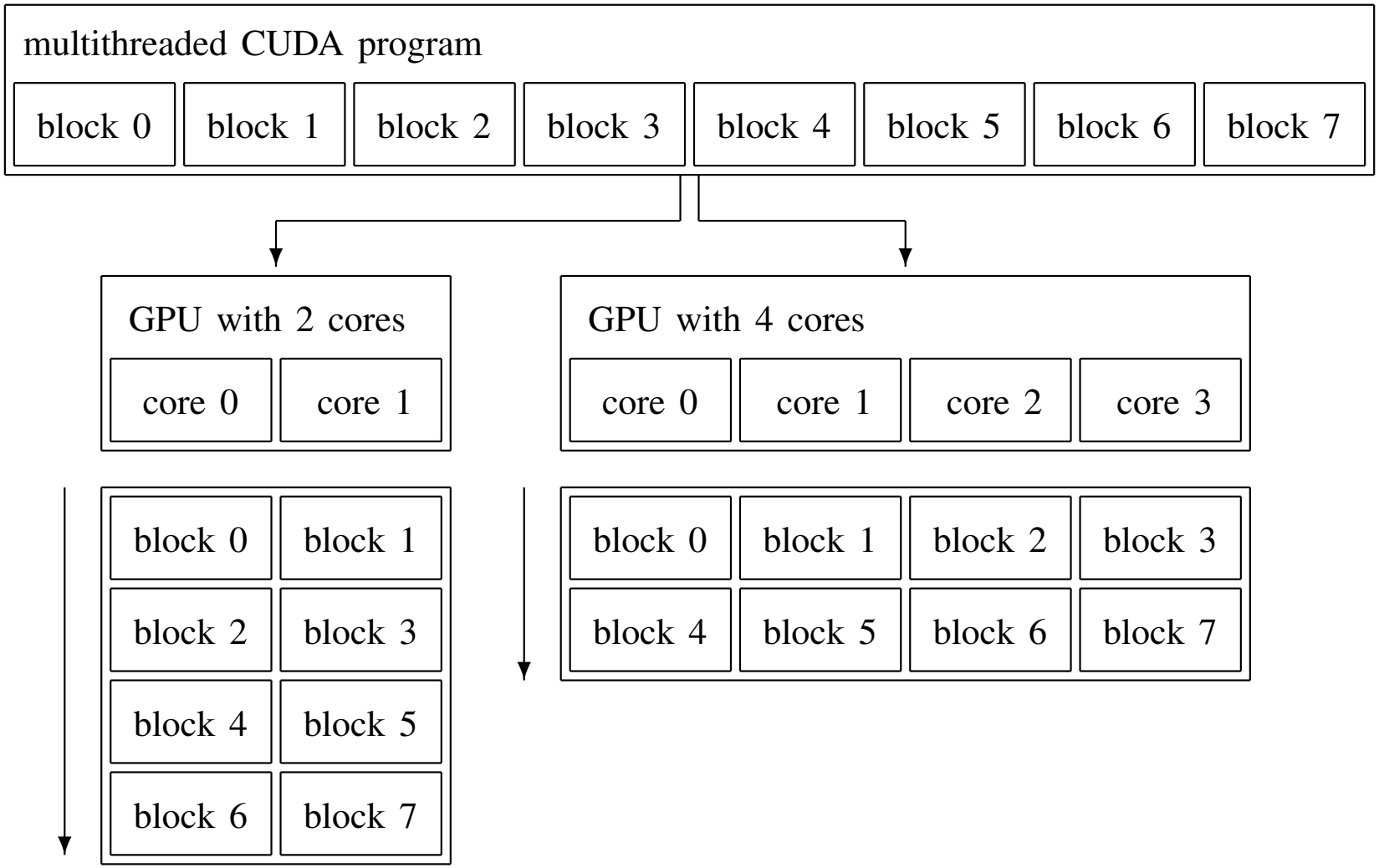
- Data parallelism: blocks of threads read from memory, execute the same instruction(s), write to memory.
- Massively parallel: need 10,000 threads for full occupancy.

The code that runs on the GPU is defined in a function, the *kernel*.

What makes the programming model scalable?

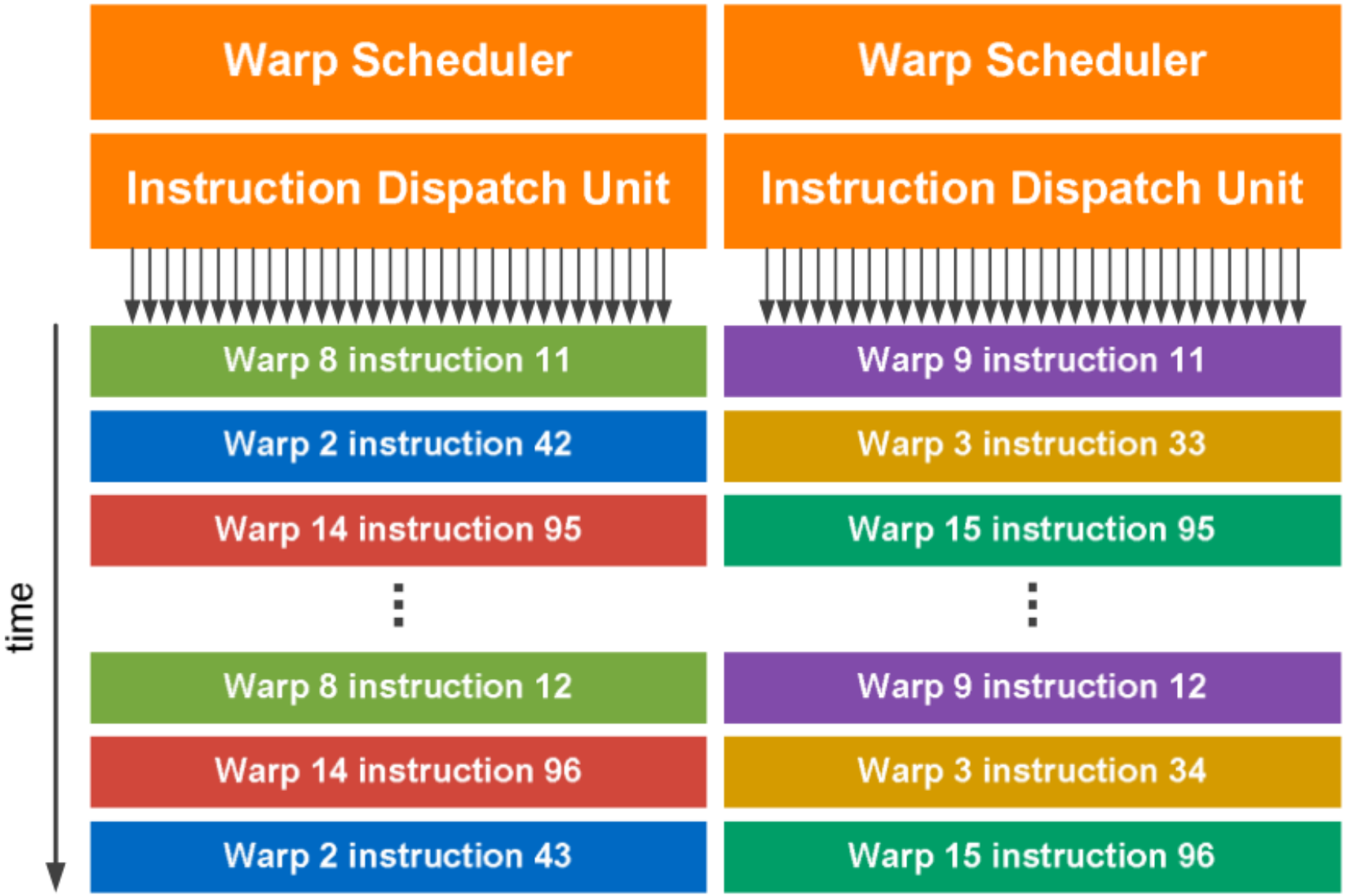
a scalable programming model

each core represents a streaming multiprocessor



dual warp scheduler

a warp consists of 32 threads



launching kernels

A kernel launch

- creates a grid of blocks, and
- each block has one or more threads.

The organization of the grids and blocks can be 1D, 2D, or 3D.

During the running of the kernel:

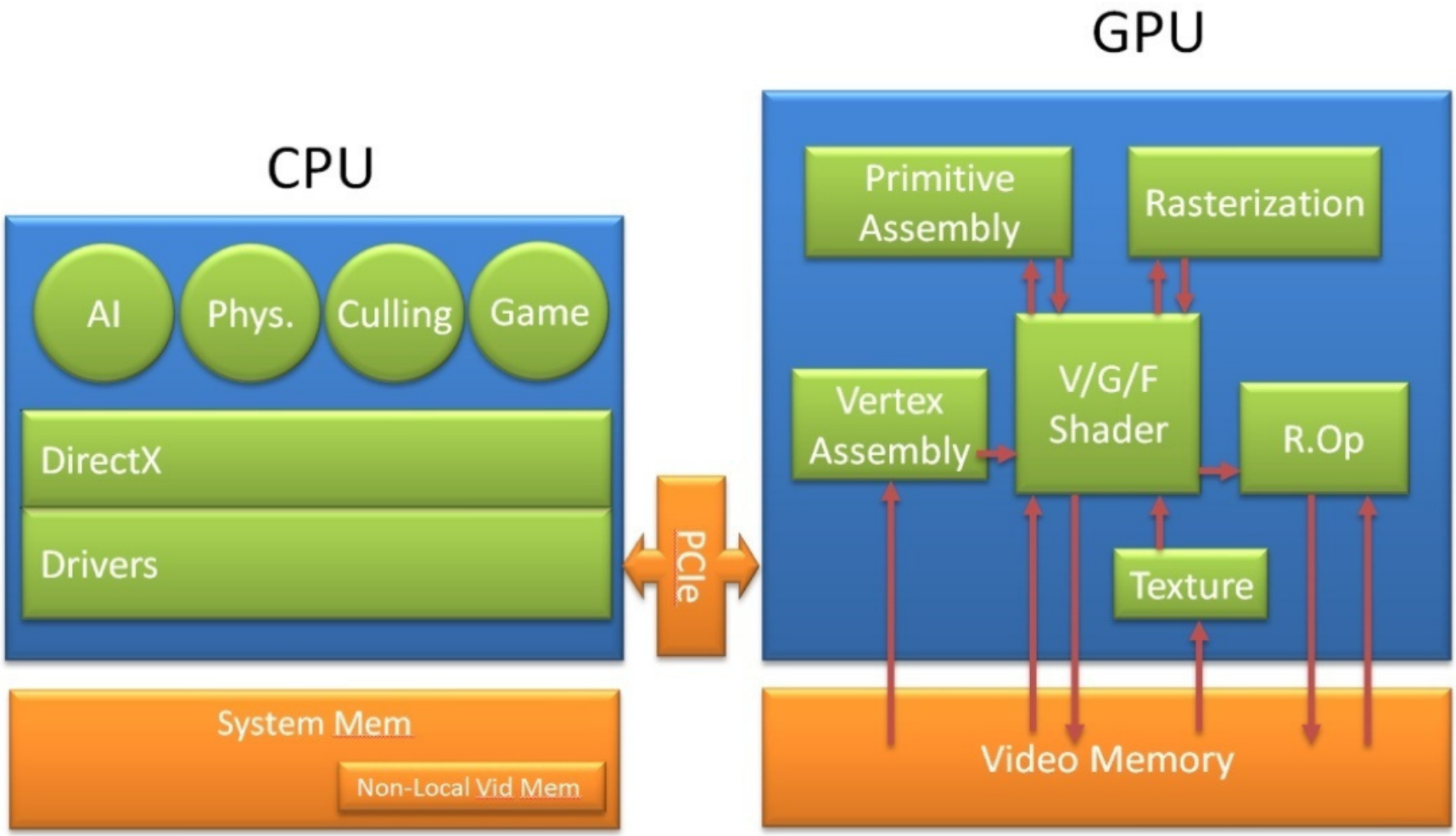
- Threads in the same block are executed simultaneously.
- Blocks are scheduled by the streaming multiprocessors.

CUDA = Compute Unified Device Architecture, by NVIDIA.

Programming GPUs with PyCUDA and Julia

1. Data Parallelism
   - the programming model
   - host (CPU) and device (GPU)
   - matrix matrix multiplication
2. PyCUDA
   - about PyCUDA
   - using PyCUDA for matrix matrix multiplication
   - the kernel explained
3. GPU Programming in Julia
   - about CUDA.jl
   - a first kernel
   - vendor agnostic GPU computing

displaying images with programmable GPUs



from the GeForce 8 and 9 Series GPU Programming Guide (NVIDIA)

Programming GPUs with PyCUDA and Julia

1. Data Parallelism
   - the programming model
   - host (CPU) and device (GPU)
   - matrix matrix multiplication
2. PyCUDA
   - about PyCUDA
   - using PyCUDA for matrix matrix multiplication
   - the kernel explained
3. GPU Programming in Julia
   - about CUDA.jl
   - a first kernel
   - vendor agnostic GPU computing

data parallelism

Many applications process large amounts of data.

Data parallelism refers to the property where many arithmetic operations can be safely performed on the data simultaneously.

Consider the multiplication of matrices A and B: $C = A \cdot B$, with

$$A = [a_{i,j}] \in \mathbb{R}^{n \times m}, \quad B = [b_{i,j}] \in \mathbb{R}^{m \times p}, \quad C = [c_{i,j}] \in \mathbb{R}^{n \times p}.$$

$c_{i,j}$ is the inner product of the ith row of A with the jth column of B:

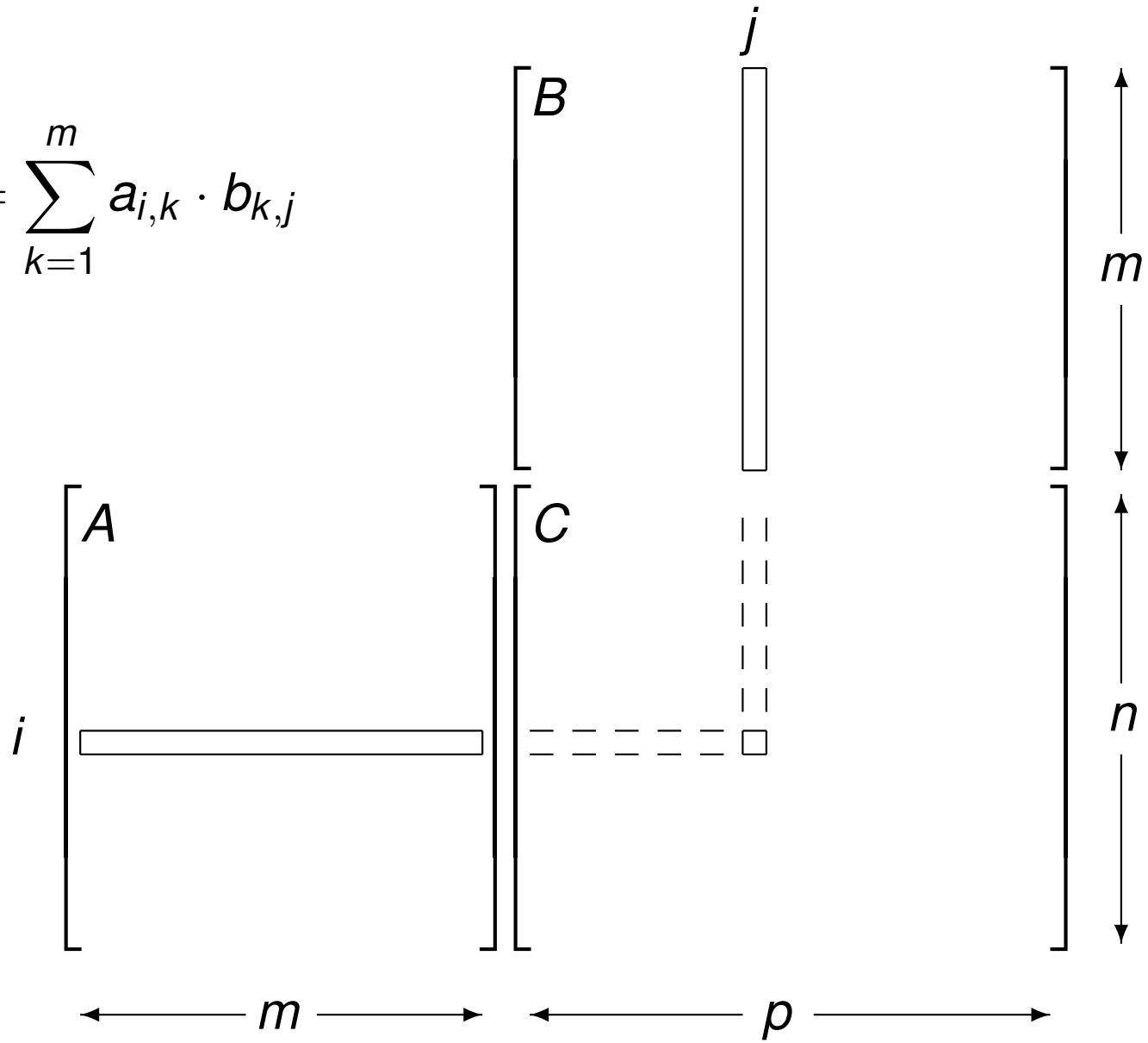
$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}.$$

All $c_{i,j}$'s can be computed independently from each other.

For $n = m = p = 1,024$ we have 1,048,576 inner products.

data parallelism in matrix multiplication

$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}$$



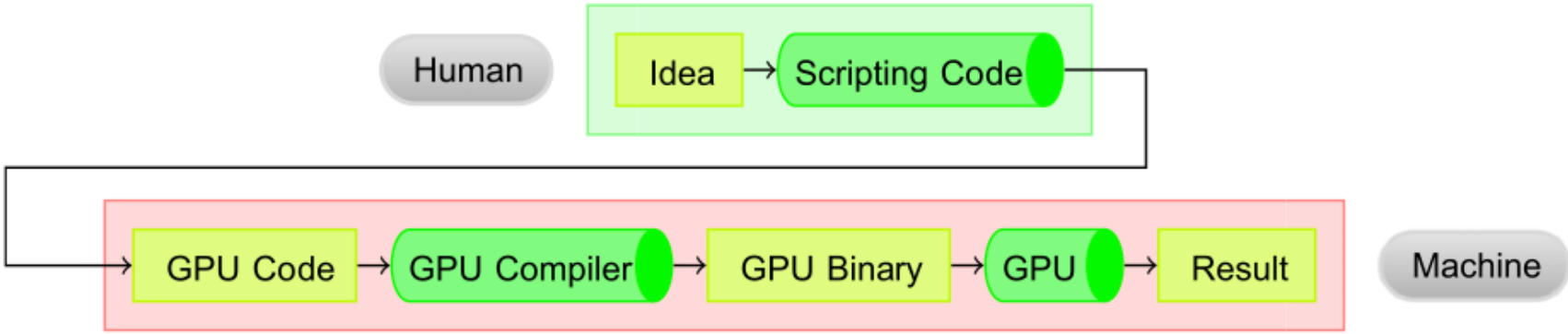
Programming GPUs with PyCUDA and Julia

1. Data Parallelism
   - the programming model
   - host (CPU) and device (GPU)
   - matrix matrix multiplication
2. PyCUDA
   - about PyCUDA
   - using PyCUDA for matrix matrix multiplication
   - the kernel explained
3. GPU Programming in Julia
   - about CUDA.jl
   - a first kernel
   - vendor agnostic GPU computing

about PyCUDA

A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih: **PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation.** *Parallel Computing*, 38(3):157–174, 2012.

The operating principle of GPU code generation:



Benefits:

- takes care of a lot of “boiler plate” code;
- focus on the kernel, with numpy typing.

We focus on PyCUDA. PyOpenCL is for GPUs not by NVIDIA.

checking the installation on `pascal`

```
$ python3
Python 3.6.8 (default, Nov 16 2020, 16:55:22)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-44)] on linux
>>> import pycuda
>>> import pycuda.autoinit
>>> from pycuda.tools import make_default_context
>>> c = make_default_context()
>>> d = c.get_device()
>>> d.name()
'Tesla P100-PCIE-16GB'
>>>
```

checking the installation on a windows laptop

which houses an NVIDIA GPU

```
$ python3
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019,
 23:11:46) [MSC v.1916 64 bit (AMD64)] on win32
>>> import pycuda
>>> import pycuda.autoinit
>>> from pycuda.tools import make_default_context
>>> c = make_default_context()
>>> d = c.get_device()
>>> d.name()
'GeForce RTX 2080 with Max-Q Design'
>>>
```

Programming GPUs with PyCUDA and Julia

1. Data Parallelism
   - the programming model
   - host (CPU) and device (GPU)
   - matrix matrix multiplication
2. PyCUDA
   - about PyCUDA
   - using PyCUDA for matrix matrix multiplication
   - the kernel explained
3. GPU Programming in Julia
   - about CUDA.jl
   - a first kernel
   - vendor agnostic GPU computing

headers and type declarations

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy

(n, m, p) = (3, 4, 5)

n = numpy.int32(n)
m = numpy.int32(m)
p = numpy.int32(p)

a = numpy.random.randint(2, size=(n, m))
b = numpy.random.randint(2, size=(m, p))
c = numpy.zeros((n, p), dtype=numpy.float32)

a = a.astype(numpy.float32)
b = b.astype(numpy.float32)
```

allocation and copy from host to device

```
a_gpu = cuda.mem_alloc(a.size * a.dtype.itemsize)
b_gpu = cuda.mem_alloc(b.size * b.dtype.itemsize)
c_gpu = cuda.mem_alloc(c.size * c.dtype.itemsize)

cuda.memcpy_htod(a_gpu, a)
cuda.memcpy_htod(b_gpu, b)
```

definition of the kernel

```
mod = SourceModule("""
    __global__ void multiply
      ( int n, int m, int p,
        float *a, float *b, float *c )
    {
        int idx = p*threadIdx.x + threadIdx.y;

        c[idx] = 0.0;
        for(int k=0; k<m; k++)
           c[idx] += a[m*threadIdx.x+k]
                     *b[threadIdx.y+k*p];
    }
    """)
```

launching the kernel

```
func = mod.get_function("multiply")
func(n, m, p, a_gpu, b_gpu, c_gpu, \
     block=(int(n), int(p), 1), \
     grid=(1, 1), shared=0)

cuda.memcpy_dtoh(c, c_gpu)

print("matrix A:")
print(a)
print("matrix B:")
print(b)
print("multiplied A*B:")
print(c)
```

running the code

At the command prompt:

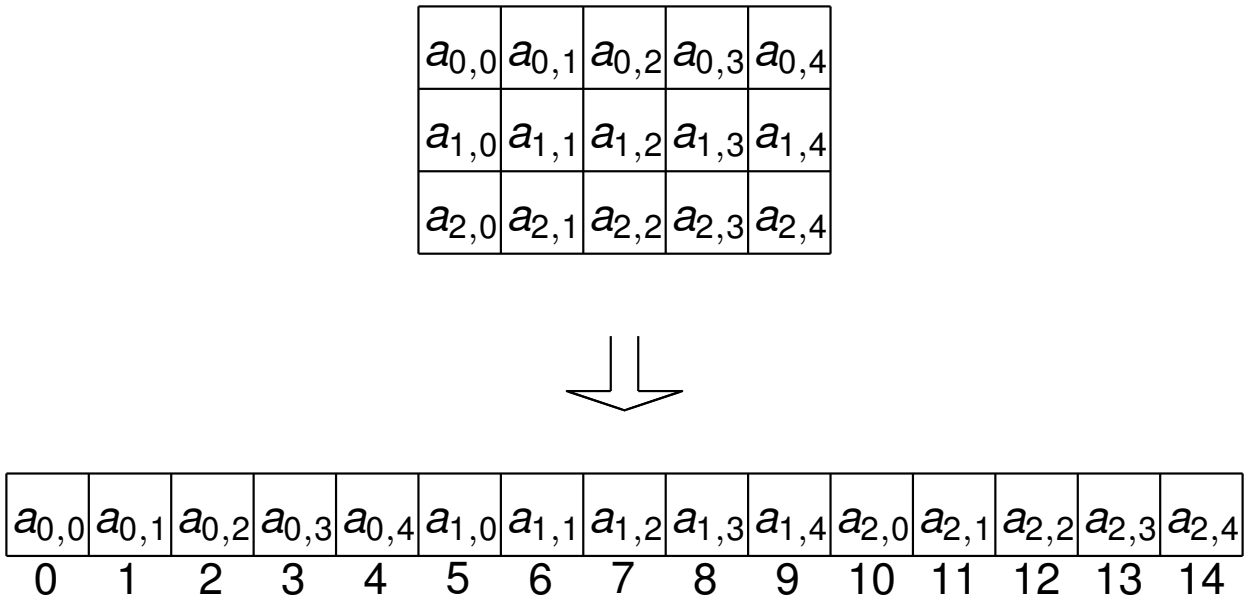
```
$ python3 matmatmulcuda.py
matrix A:
[[ 0.  0.  1.  0.]
 [ 0.  0.  1.  1.]
 [ 0.  1.  1.  0.]]
matrix B:
[[ 1.  1.  0.  1.  1.]
 [ 1.  0.  1.  0.  0.]
 [ 0.  0.  1.  1.  0.]
 [ 0.  0.  1.  1.  0.]]
multiplied A*B:
[[ 0.  0.  1.  1.  0.]
 [ 0.  0.  2.  2.  0.]
 [ 1.  0.  2.  1.  0.]]
$
```

Programming GPUs with PyCUDA and Julia

1. Data Parallelism
   - the programming model
   - host (CPU) and device (GPU)
   - matrix matrix multiplication
2. PyCUDA
   - about PyCUDA
   - using PyCUDA for matrix matrix multiplication
   - the kernel explained
3. GPU Programming in Julia
   - about CUDA.jl
   - a first kernel
   - vendor agnostic GPU computing

linear address system

Consider a 3-by-5 matrix stored row-wise (as in C):



The `float *a` in the kernel represents a one dimensional array.

assigning inner products to threads

Consider a 3-by-4 matrix A and a 4-by-5 matrix B:

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$	$b_{0,4}$
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$	$b_{1,4}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$	$b_{2,4}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$	$b_{3,4}$

$c_{0,0}$	$c_{0,1}$	$c_{0,2}$	$c_{0,3}$	$c_{0,4}$	$c_{1,0}$	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$c_{1,4}$	$c_{2,0}$	$c_{2,1}$	$c_{2,2}$	$c_{2,3}$	$c_{2,4}$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

The `idx = p*threadIdx.x + threadIdx.y`
determines what entry in $C = A \cdot B$ will be computed.

a two dimensional grid of threads

A is n-by-m, B is m-by-p, and $C = A \cdot B$ is n-by-p.

```
func = mod.get_function("multiply")
func(n, m, p, a_gpu, b_gpu, c_gpu, \
     block=(numpy.int(n), numpy.int(p), 1), \
     grid=(1, 1), shared=0)
```

- One block of threads is launched of size `n`-by-`p`.
- Each thread computes one element of the product, defined by

```
idx = p*threadIdx.x + threadIdx.y
c[idx] = 0.0;
for(int k=0; k<m; k++)
   c[idx] += a[m*threadIdx.x+k]
             *b[threadIdx.y+k*p];
```

Programming GPUs with PyCUDA and Julia

1. Data Parallelism
   - the programming model
   - host (CPU) and device (GPU)
   - matrix matrix multiplication
2. PyCUDA
   - about PyCUDA
   - using PyCUDA for matrix matrix multiplication
   - the kernel explained
3. GPU Programming in Julia
   - about CUDA.jl
   - a first kernel
   - vendor agnostic GPU computing

about CUDA.jl

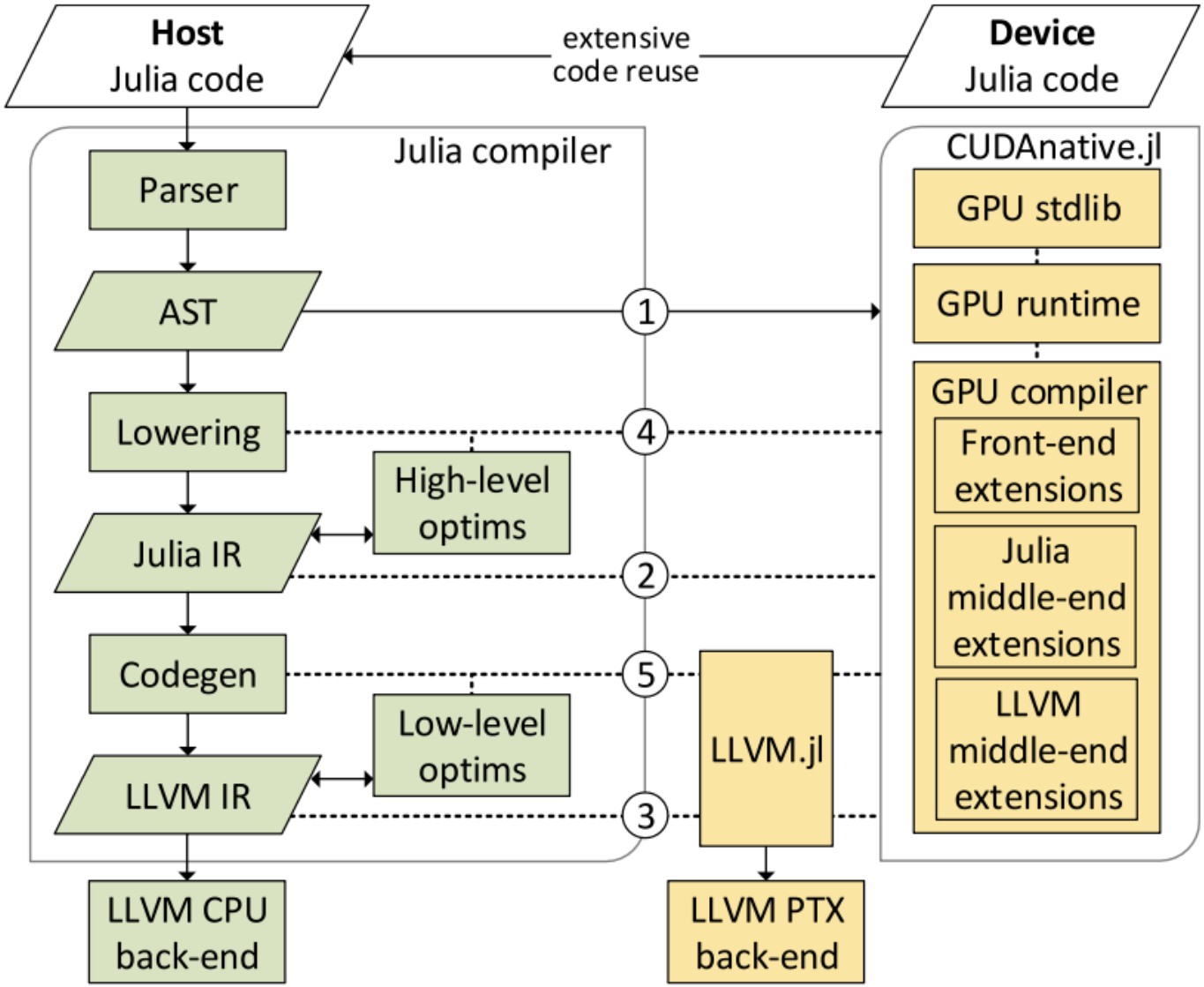
T. Besard, C. Foket, and B. De Sutter:
Effective Extensible Programming: Unleashing Julia on GPUs.
IEEE Transactions on Parallel and Distributed Systems,
Vol. 30, No. 4, pages 827–841, 2019.

Installation:

1. Verify that the computer has an NVIDIA GPU
   and that the CUDA Software Development Kit is installed.
2. Type `using CUDA` to install in Julia.
3. Type `]` to get the `pkg>` prompt and type `test CUDA`.
4. Be patient.

`https://juliagpu.org` is the site of JuliaGPU, the organization to unify the many packages for programming GPUs in Julia.

the compilation process for Julia GPU code



From Besard et al., 2019.

Programming GPUs with PyCUDA and Julia

1. Data Parallelism
   - the programming model
   - host (CPU) and device (GPU)
   - matrix matrix multiplication
2. PyCUDA
   - about PyCUDA
   - using PyCUDA for matrix matrix multiplication
   - the kernel explained
3. GPU Programming in Julia
   - about CUDA.jl
   - a first kernel
   - vendor agnostic GPU computing

a first kernel

https://cuda.juliagpu.org/stable/tutorials/introduction

```
using CUDA
using Test

function gpu_add1!(y, x)
    for i = 1:length(y)
        @inbounds y[i] += x[i]
    end
    return nothing
end

N = 2^20                # adding one million Float32 numbers
x_d = CUDA.fill(1.0f0, N) # stored on GPU filled with 1.0
y_d = CUDA.fill(2.0f0, N) # stored on GPU filled with 2.0

fill!(y_d, 2)
@cuda gpu_add1!(y_d, x_d)
result = (@test all(Array(y_d) .== 3.0f0))
println(result)
```

running at the command prompt

The code on the previous slide is saved in `gpuadd.jl`.

Type `julia gpuadd.jl` at the command prompt.

Some warnings may appear ...

but `Test Passed` should be the result.

Programming GPUs with PyCUDA and Julia

1. Data Parallelism
   - the programming model
   - host (CPU) and device (GPU)
   - matrix matrix multiplication
2. PyCUDA
   - about PyCUDA
   - using PyCUDA for matrix matrix multiplication
   - the kernel explained
3. GPU Programming in Julia
   - about CUDA.jl
   - a first kernel
   - vendor agnostic GPU computing

vendor agnostic GPU computing

In addition to CUDA.jl, the following packages are available:

- AMDGPU.jl for AMD GPUs,
- oneAPI.jl for the Intel oneAPI,
- Metal.jl to program GPUs in Apple hardware.

U. Utkarsh, V. Churavy, Y. Ma, T. Besard, P. Srisuma, T. Gymnich, A. R. Gerlach, A. Edelman, G. Barbastathis, R. D. Braatz, and C. Rackauckas: **Automated Translation and Accelerated Solving of Differential Equations on Multiple GPU Platforms**. *Computer Methods in Applied Mechanics and Engineering*, Vol 419, 2024, article 116591.

adding vectors on an M1 MacBook Air GPU with Metal

```
using Metal
using Test

function gpu_add1!(y, x)
    for i = 1:length(y)
        @inbounds y[i] += x[i]
    end
    return nothing
end

N = 32
x_d = Metal.fill(1.0f0, N)  # filled with Float32 1.0 on GPU
y_d = Metal.fill(2.0f0, N)  # filled with Float32 2.0

# run with N threads

@metal threads=N gpu_add1!(y_d, x_d)
result = (@test all(Array(y_d) .== 3.0f0))

println(result)
```

multiplying matrices with Metal

The CUDA version of the example is copied from the Julia for High-Performance Scientific Computing web site, adjusted

1. using `Metal` instead of using `CUDA`,
2. work with `Float32` instead of `Float64`,
3. use `MtlArray` instead of `CuArray`.

```
using Metal
using BenchmarkTools

dim = 2^10
A_h = rand(Float32, dim, dim);
A_d = MtlArray(A_h);

@btime $A_h * $A_h;
@btime $A_d * $A_d;
```

Prints 6.229 ms and 23.625 μs for CPU and GPU respectively.

Exercises

Read the literature on PyCUDA and at `https://juliagpu.org`.

1. The matrix matrix multiplication example with PyCUDA uses $(3, 4, 5)$ as values for (n, m, p). Considering massive parallelism, what are the largest dimensions you could consider for one block of threads on the P100 and/or the A100?
   Illustrate your values for the dimensions experimentally.
2. In the PyCUDA matrix matrix multiplication, change the `float32` types into `float64` and redo the previous exercise.
   Time the code. Do you notice a difference?
3. On your own computer, check the vendor of the GPU and run the equivalent `gpuadd.jl` after installing the proper Julia package.
   Report on the performance, relative to the CPU in your computer.