Unified GPU Programming in Julia: Vendor Abstraction and High-Level Control with KernelAbstractions

Julian Samaroo
Rabab Alomairy
with material by Valentin Churavy

Outline

- Results to show the power of Julia.
- How to program a GPU
- Matrix Multiplication Example (CUDA C++)
- Matrix Multiplication Example (CUDA.jl, AMDGPU.jl, oneAPI.jl and Metal.jl)
- Matrix Multiplication Example (KernelAbstractions.jl)
- 2D Heat Diffusion Simulation Using Stencil Computation in Julia
- 2D Gray-Scott Reaction-Diffusion Model Using Stencil Computation in Julia

2D Heat Diffusion Simulation in Julia

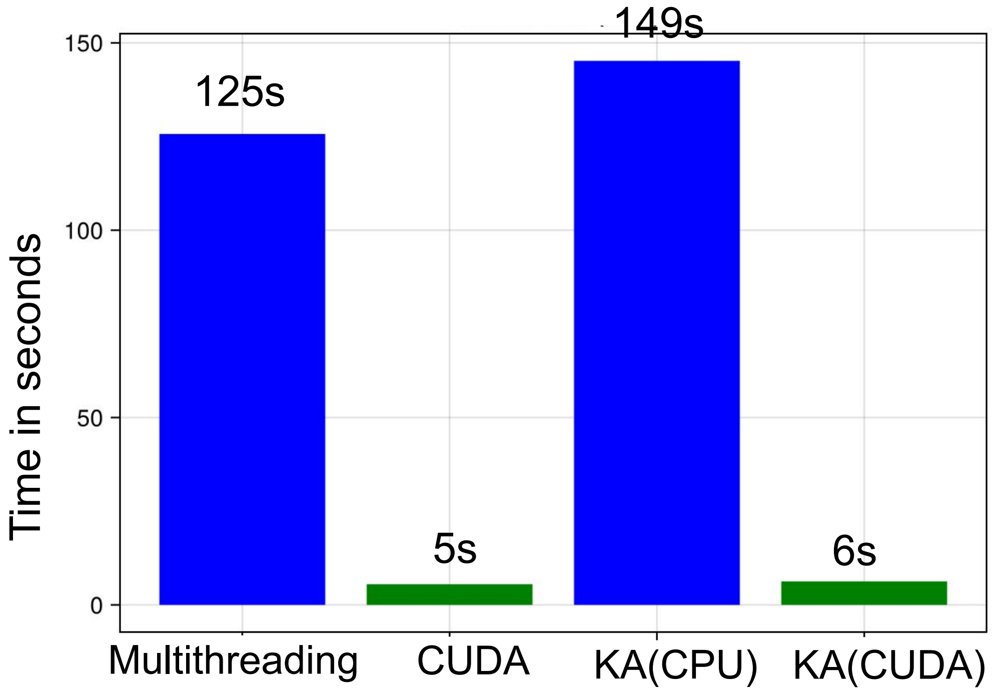
- The grid size is **10240x10240** and number of time steps **2000**.
- Comparing performance of four Julia parallelism paradigms:

  - Multithreading
  - CUDA
  - KernelAbstractions with CPUs
  - KernelAbstractions with CUDA

Similar APIs

Unified API

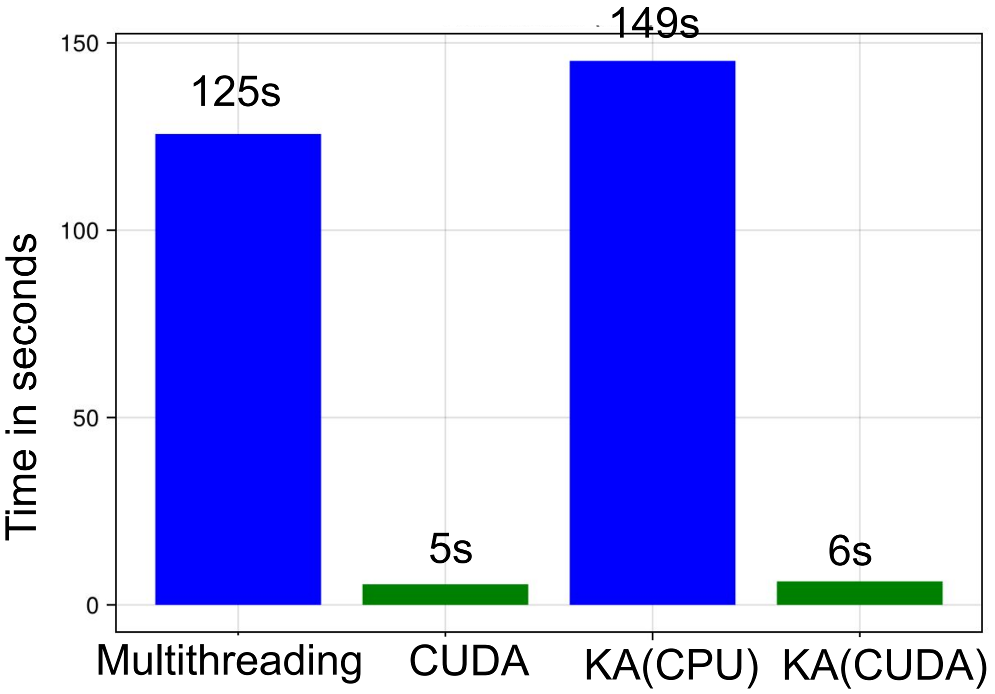
- Blue bars correspond to CPU backend
- Green bars correspond to CUDA backend



Single A100 GPU in Perlmutter

2D Heat Diffusion Simulation in Julia

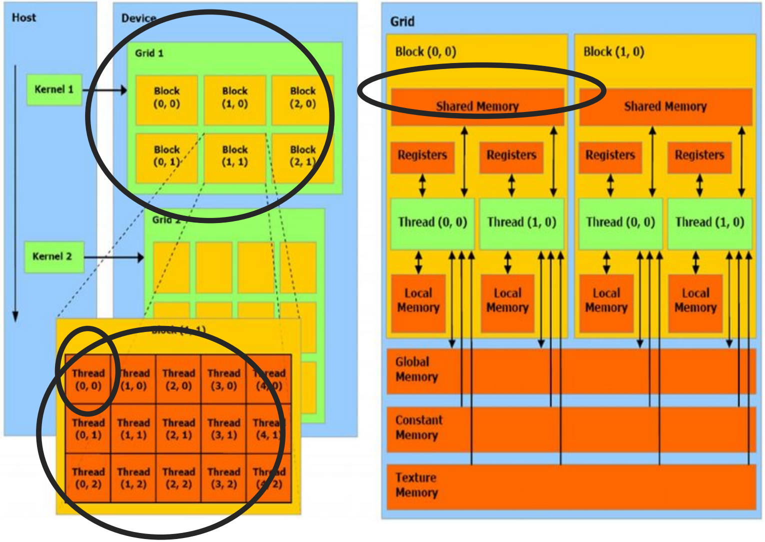
- Performance gains
  - GPU faster than CPU
- Flexibility vs. Performance
  - CUDA is great
  - KernelAbstractions.jl is flexible with slight performance trade-off
- Ease of Use
  - Seamless integration
  - Keeping readability and maintainability



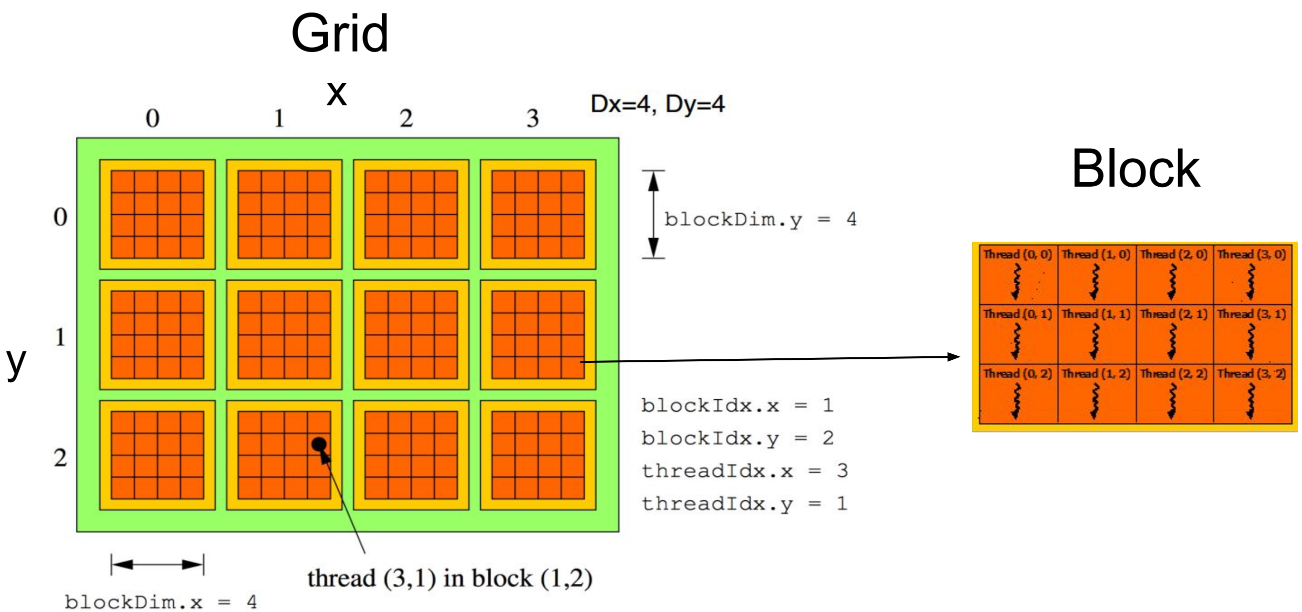
Single A100 GPU in Perlmutter

How to program a GPU

- Single-Instruction-Multiple-Threads (SIMT).
- SIMT is designed to allow GPUs to execute the same instruction across multiple threads
- Programming perspective is a thread
- Threads are grouped into blocks/warps
- Blocks can then be grouped into grids
- Threads within block can access shared memory and synchronize as needed
- GPUs are designed to maximize throughput.
- Transferring data from the CPU to the GPU is a necessary step and can be a bottleneck



SIMT Programming model



threadIdx:

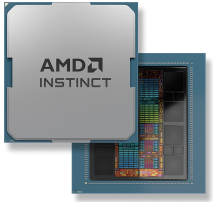
$$x = \texttt{blockIdx.x} \times \texttt{blockDim.x} + \texttt{threadIdx.x} = 1 \times 4 + 3 = 7$$

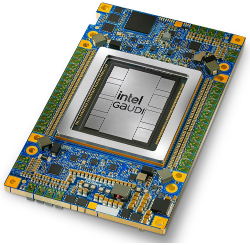
$$y = \texttt{blockIdx.y} \times \texttt{blockDim.y} + \texttt{threadIdx.y} = 2 \times 4 + 1 = 9$$

- Global thread ID: the position of the thread within block (threadIdx) , the position of the block within the grid (blockIdx), and block dimension (blockDim)

Existing GPUs and their Software

Hardware	NVIDIA	AMD	Intel	Apple
Software	CUDA	ROCm	OneAPI	Metal





NVIDIA Ampere A100

- An NVIDIA A100 GPU contains 108 Streaming Multiprocessors (SMs)
- It has 40 MB L2 cache, which helps reduce latency
- It supports up to 80 GB of HBM2 memory with a maximum memory bandwidth of 2039 GB/s , essential for handling large data and reducing data transfer.
- It contains in total 6912 FP32/INT32 CUDA cores and 3456 FP64 CUDA cores

NVIDIA H100: offers up to 30 TFLOPs of FP64 performance, which is over 3x the FP64 performance of the A100

GB200 Grace Blackwell Superchip: Provides 90 TFLOPS of FP64 performance

ASCI White supercomputer
Lawrence Livermore National Laboratory
Top #1 in 2007, 7.9 TFLOPS

NVIDIA A100 TENSOR CORE GPU SPECIFICATIONS (SXM4 AND PCIE FORM FACTORS)

	A100 40GB PCIe	A100 80GB PCIe	A100 40GB SXM	A100 80GB SXM
FP64	9.7 TFLOPS			
FP64 Tensor Core	19.5 TFLOPS			
FP32	19.5 TFLOPS			
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*			
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
INT8 Tensor Core	624 TOPS | 1248 TOPS*			
GPU Memory	40GB HBM2	80GB HBM2e	40GB HBM2	80GB HBM2e
GPU Memory Bandwidth	1,555GB/s	1,935GB/s	1,555GB/s	2,039GB/s
Max Thermal Design Power (TDP)	250W	300W	400W	400W
Multi-Instance GPU	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB
Form Factor	PCIe		SXM	
Interconnect	NVIDIA® NVLink® Bridge for 2 GPUs: 600GB/s ** PCIe Gen4: 64GB/s		NVLink: 600GB/s PCIe Gen4: 64GB/s	
Server Options	Partner and NVIDIA-Certified Systems™ with 1-8 GPUs		NVIDIA HGX™ A100-Partner and NVIDIA-Certified Systems with 4,8, or 16 GPUs NVIDIA DGX™ A100 with 8 GPUs	

* With sparsity

** SXM4 GPUs via HGX A100 server boards; PCIe GPUs via NVLink Bridge for up to two GPUs

Differences between a CPU and a GPU

- Handles all tasks independently
- Able to rapidly switch between tasks
- Can handle complex tasks well

- Relies on CPU for instructions
- Performs same tasks multiple times very fast
- Handles simple tasks extremely well

Vector Addition Example

CPU code

```
vector_size = 1024
a = rand(1:4, vector_size)
b = rand(1:4, vector_size)
c = zeros(Int, vector_size)

function vadd(c, a, b)
    for i in 1:vector_size
        c[i] = a[i] + b[i]
    end
    return
end
```

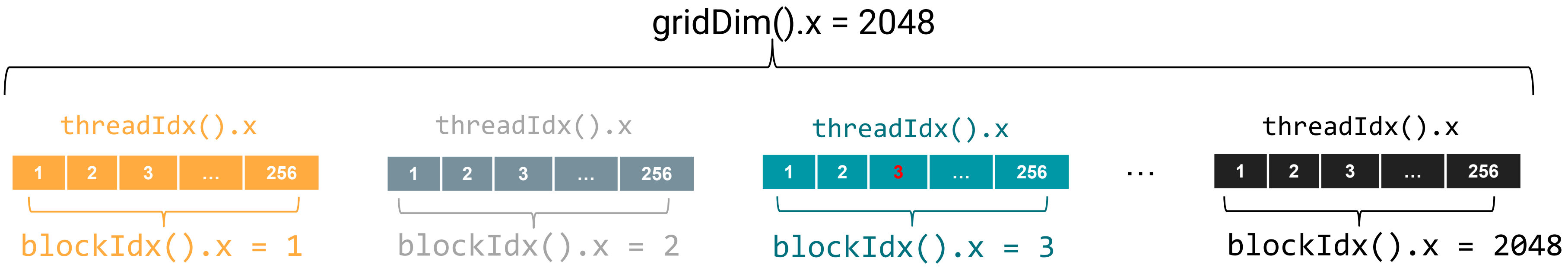
GPU (CUDA) kernel

```
da = CuArray(a)
db = CuArray(b)
dc = CUDA.zeros(Int, size(a))


function vadd(c, a, b)
    i = threadIdx().x + (blockIdx().x - 1) *
blockDim().x
    c[i] = a[i] + b[i]
    return
end
@cuda threads=length(a) vadd(dc, da, db)
```

Enhanced GPU Vector Addition

- threadIdx().x + (blockIdx().x - 1) * blockDim().x = 3 + (3-1) * 256 = 515

Gets the global index of the thread in a multidimensional grid



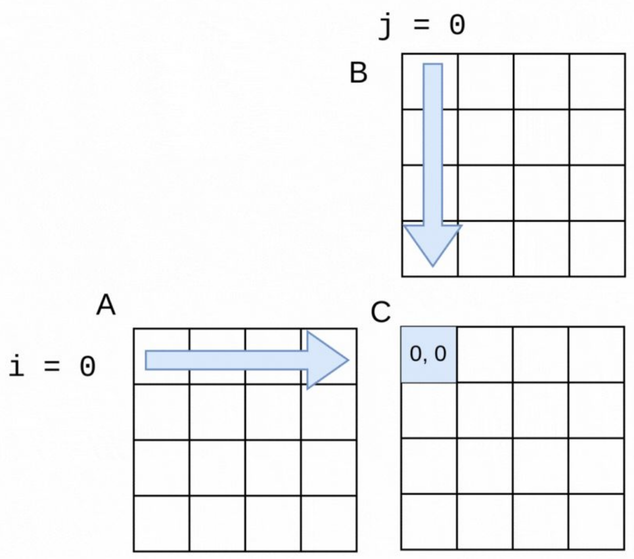

```julia
da = CuArray(a)
db = CuArray(b)
dc = CUDA.zeros(Int, size(a))

function vadd(c, a, b)
    i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
    c[i] = a[i] + b[i]
    return
end
@cuda threads=1024 blocks=cld(length(da),1024) vadd(dc, da, db)
```

Matrix Multiplication Example

- Matrix multiplication for computing each element of matrix C from matrices A and B can be written:

$$C_{i,j} = \sum_{k=0}^{l} A_{i,k} * B_{k,j}$$

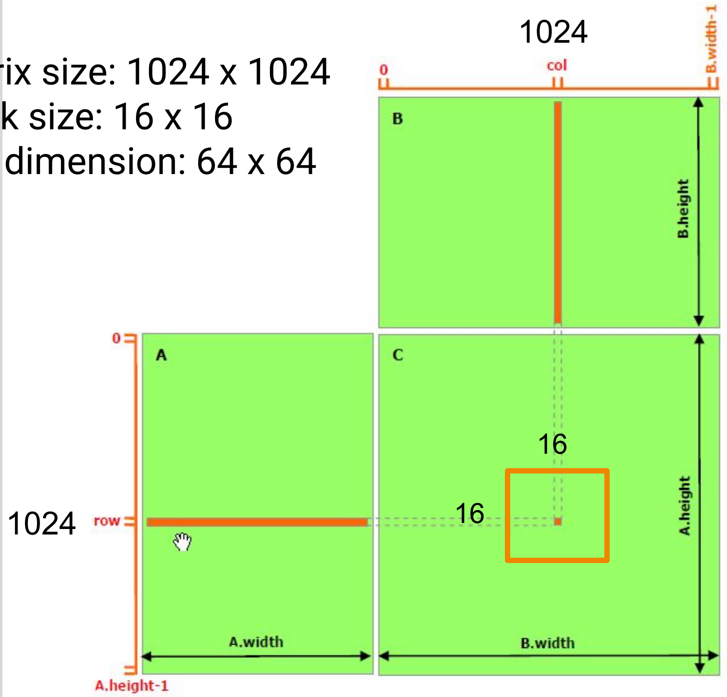
```
for i in 0:m-1
    for j in 0:n-1
        C[i, j] = 0
        for k in 0:l-1
            C[i, j] += A[i, k] * B[k, j]
        end
    end
end
```



Matrix Multiplication Example (CUDA C++)

```
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);

    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);

    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    cudaMemcpy(C.elements, d_C.elements, size,
               cudaMemcpyDeviceToHost);

    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}

__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

Matrix size: 1024 x 1024
Block size: 16 x 16
Grid dimension: 64 x 64



CUDA C++ Programming Guide
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=matrix%20multiply

CUDA.jl

```
using CUDA

function MatrixMultiplication!(A,B,C)

    row = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    col = (blockIdx().y - 1) * blockDim().y + threadIdx().y

    sum = zero(eltype(C))

    if row <= size(A, 1) && col < size(B, 2)
        for i = 1:size(A, 2)
            sum += A[row, i] * B[i, col]
        end
        C[row, coll] = sum
    end

    return
end
```

GPU array

CPU array

```
a = rand(1024, 1024
b = rand(1024, 1024
A = CuArray(a)
B = CuArray(b)
C = CUDA.zeros(1024, 1024

threads = (16, 16)
blocks  = (64, 64)

@cuda threads=threads blocks=blocks
MatrixMultiplication!(A, B, C)
c = Array(C)
```

Allocate and transfer to GPU

GPU resident

Lunch GPU kernel

Allocate and transfer to CPU

AMDGPU.jl

```
using AMDGPU

function MatrixMultiplication!(A, B,C)

    row = (workgroupIdx()).x - 1) * workgroupDim(). x + workitemIdx().x
    col = (workgroupIdx()).y - 1) * workgroupDim(). y + workitemIdx()).y

    sum = zero (eltype(C))

    if row <= size(A, 1) 8& col <= size(B, 2)
        for 1 = 1:size(A, 2)
            sum += A[row, i] * B[i, col]
        end
        C[row, col] = sum
    end

    return
end
```

```
a = rand(1024, 1024)
b = rand(1024, 1024)
A = ROCArray(a)
B = ROCArray(b)
C = AMDGPU.zeros(1024, 1024)

threads = (16, 16)
blocks  = (64, 64)

@roc groupsize=threads gridsize=blocks
MatrixMultiplication!(A, B, C)
c = Array(C)
```

oneAPI.jl and Metal.jl

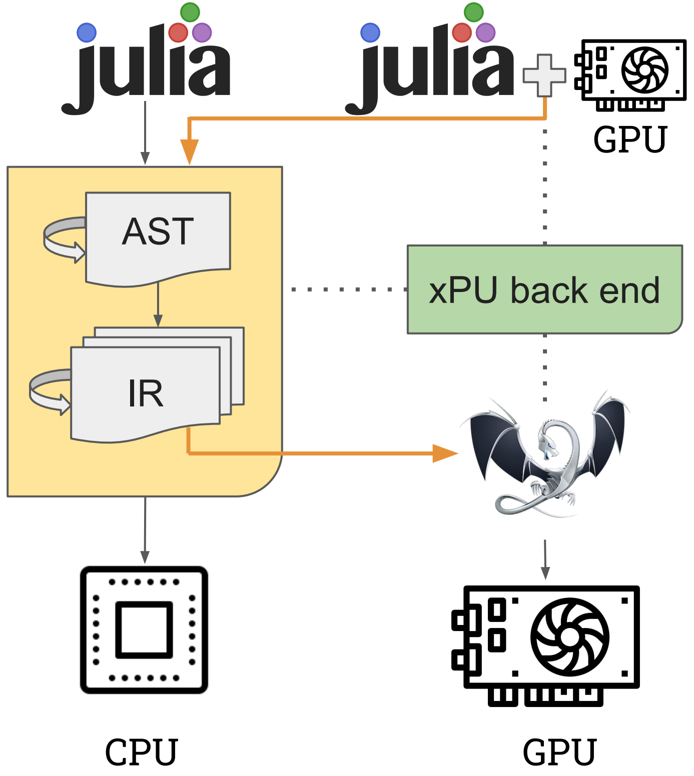
```julia
using oneAPI

function MatrixMultiplication!(A,B,C)

    row = get_global_id(0)
    col = get_global_id(1)

    sum = zero(eltype(C))

    if row <= size(A, 1) && col <= size(B, 2)
        for i = 1:size(A, 2)
            sum += A[row, i] * B[i, col]
        end
        C[row, col] = sum
    end

    return

end
```

```julia
using Metal

function MatrixMultiplication!(A, B,C)

    row, col = thread_position_in_grid_2d()

    sum = zero(eltype(C))

    if row <= size(A, 1) && col <= size(B, 2)
        for i = 1:size(A, 2)
            sum += A[row, i] * B[i, col]
        end
        C[row, col] = sum
    end

    return


end
```

julia gets its Power from Extensible Compiler Design

Language design

Efficient execution



Julia: Dynamism and Performance Reconciled by Design
Bezanson J. et al
(*[doi:10.1145/3276490](https://doi.org/10.1145/3276490)*)

Effective Extensible Programming: Unleashing Julia on GPUs
Besard T. et al
(*[doi:10.1109/TPDS.2018.2872064](https://doi.org/10.1109/TPDS.2018.2872064)*)

GPUs from different vendors are similar

1. All of them are **implicit vector** architectures
2. All of them are focused around **kernels**
   => Programs that are launched on the host,
   and execute asynchronously on the device

`KernelAbstractions.jl` provides a shallow portability layer for GPUs from AMD, Intel, Apple and NVIDIA.

```julia
using CUDA
function gemm!(A,B,C)
    row = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    col = (blockIdx().y - 1) * blockDim().y + threadIdx().y

    sum = zero(eltype(C))
```

```julia
using Metal
function gemm!(A,B,C)
    row, col = thread_position_in_grid_2d()

    sum = zero(eltype(C))

    if row <= size(A, 1) && col <= size(B, 2)
```

```julia
using AMDGPU
function gemm!(A,B,C)
    row = (workgroupIdx().x - 1) * workgroupDim().x + workitemIdx().x
    col = (workgroupIdx().y - 1) * workgroupDim().y + workitemIdx().y

    sum = zero(eltype(C))

    if row <= size(A, 1) && col <= size(B, 2)
        for i = 1:size(A, 2)
            @inbounds sum += A[row, i] * B[i, col]
```

```julia
using oneAPI
function gemm!(A,B,C)
    row = get_global_id(0)
    col = get_global_id(1)

    sum = zero(eltype(C))

    if row <= size(A, 1) && col <= size(B, 2)
        for i = 1:size(A, 2)
            @inbounds sum += A[row, i] * B[i, col]
        end
        @inbounds C[row, col] = sum
    end

    return
end
```

```julia
using KernelAbstractions

@kernel function MatrixMultiplication_kernel!(A, B, C)

    row, col = @index(Global, NTuple)

    sum = zero(eltype(C))

    if row <= size(A, 1) && col <= size(B, 2)
     for i = 1:size(A, 2)
         sum += A[row, i] * B[i, col]
     end
         C[row, col] = sum
     end

end
```

KernelAbstractions.jl

```
using KernelAbstractions

@kernel function MatrixMultiplication_kernel!(A, B, C)

    row, col = @index(Global, NTuple)

    sum = zero(eltype(C))

    if row <= size(A, 1) && col <= size(B, 2)
     for i = 1:size(A, 2)
         sum += A[row, i] * B[i, col]
     end
         C[row, col] = sum
     end

end
```

```
# Allocate and initialize GPU arrays
A = rand!(allocate(Backend, Type, 1024, 1024))
B = rand!(allocate(Backend, Type, 1024, 1024))
C = KernelAbstractions.zeros(Backend, Type, 1024, 1024)

# Compile the kernel for this workgroup (block) size
workgroupsize = (16, 16)
kernel! = MatrixMultiplication_kernel!(Backend, workgroupsize)
# Launch the kernel, and synchronize the current stream
kernel!(A, B, C, ndrange=(size(C)))
KernelAbstractions.synchronize(Backend)
```

Kernel language — `@kernel`

KernelAbstractions defines a language valid within `@kernel` definitions

- `@Const`: Declares an argument to not be aliased or written to
- `@index`: Which kernel element are we operating upon?
- `@localmem`: Allocate local (shared) memory
- `@synchronize`: Synchronize warps/wavefronts
- `@private`: Allocate private memory for a lane
- `@print`: Unified printing of device-side data
- …

Kernel language: Indexing

`Indexing Method:`

- `@index(Global, Kind)`: Global index for accessing input arguments
- `@index(Group, Kind)`: Which work-group does the lane belong to?
- `@index(Local, Kind)`: Which item inside a work-group is this lane?

`Indexing Kind:`

- `@index(Locale, Cartesian)`: Index as a `CartesianIndex`
- `@index(Locale, NTuple)`: Index as a tuple
- `@index(Locale, Linear)`: Index as a scalar

Thank You