Julia: A competitive high-level choice for performance portability in HPC?

Jim M. R. Teichgräber

5th July, 2022

Outline

Outline

Outline

1. Motivation & Background
    - Julia's compilation process

Sources

- Bezanson et al. "Julia: A fast dynamic language for technical computing" [Bez+12]

Outline

1. Motivation & Background
    - Julia's compilation process
    - Julia as a high-level language

Sources

- Bezanson et al. "Julia: A fast dynamic language for technical computing" [Bez+12]

Outline

1. Motivation & Background
   - Julia's compilation process
   - Julia as a high-level language
   - Programming accelerators in Julia

Sources

- Bezanson et al. "Julia: A fast dynamic language for technical computing" [Bez+12]
- Besard et al. "Effective Extensible Programming: Unleashing Julia on GPUs" [BFD18]

Outline

1. Motivation & Background
   - Julia's compilation process
   - Julia as a high-level language
   - Programming accelerators in Julia
2. Benchmarks

Sources

- Bezanson et al. "Julia: A fast dynamic language for technical computing" [Bez+12]
- Besard et al. "Effective Extensible Programming: Unleashing Julia on GPUs" [BFD18]
- Lin et al. "Comparing Julia to Performance Portable Programming Models for HPC" [LM21]

Outline

1. Motivation & Background
   - Julia's compilation process
   - Julia as a high-level language
   - Programming accelerators in Julia
2. Benchmarks
3. Performance results

Sources

- Bezanson et al. "Julia: A fast dynamic language for technical computing" [Bez+12]
- Besard et al. "Effective Extensible Programming: Unleashing Julia on GPUs" [BFD18]
- Lin et al. "Comparing Julia to Performance Portable Programming Models for HPC" [LM21]
- Hunold et al. "Benchmarking Julia's Communication Performance: Is Julia HPC ready or Full HPC?" [HS20]

Outline

1. Motivation & Background
   - Julia's compilation process
   - Julia as a high-level language
   - Programming accelerators in Julia
2. Benchmarks
3. Performance results
4. Discussion

Sources

- Bezanson et al. "Julia: A fast dynamic language for technical computing" [Bez+12]
- Besard et al. "Effective Extensible Programming: Unleashing Julia on GPUs" [BFD18]
- Lin et al. "Comparing Julia to Performance Portable Programming Models for HPC" [LM21]
- Hunold et al. "Benchmarking Julia's Communication Performance: Is Julia HPC ready or Full HPC?" [HS20]

Outline

1. Motivation & Background
   - Julia's compilation process
   - Julia as a high-level language
   - Programming accelerators in Julia
2. Benchmarks
3. Performance results
4. Discussion
5. Conclusion

Sources

- Bezanson et al. "Julia: A fast dynamic language for technical computing" [Bez+12]
- Besard et al. "Effective Extensible Programming: Unleashing Julia on GPUs" [BFD18]
- Lin et al. "Comparing Julia to Performance Portable Programming Models for HPC" [LM21]
- Hunold et al. "Benchmarking Julia's Communication Performance: Is Julia HPC ready or Full HPC?" [HS20]

Motivation & Background

Julia

- Dynamic language

Julia

- Dynamic language
  $\rightarrow$ needs some kind of runtime environment to enable arbitrary types at runtime

Julia

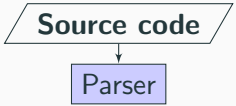
- Dynamic language
  $\rightarrow$ needs some kind of runtime environment to enable arbitrary types at runtime
- Goal: Speed and convenience

Julia

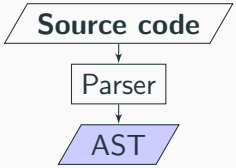
- Dynamic language
  $\rightarrow$ needs some kind of runtime environment to enable arbitrary types at runtime
- Goal: Speed and convenience
  $\rightarrow$ Just-in-time (JIT) compilation

Julia — Compilation

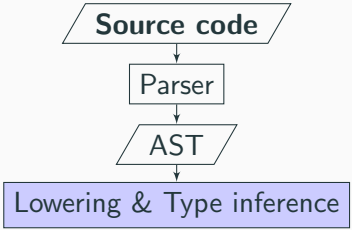
Julia — Compilation



Julia — Compilation

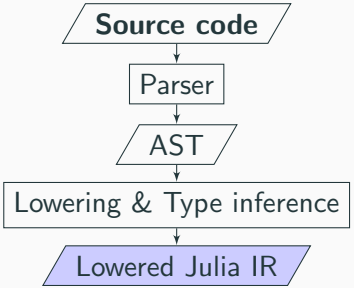


Julia — Compilation



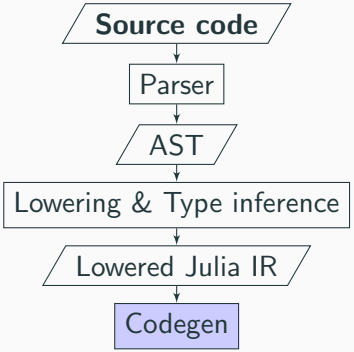
- Type inference
  - Allows static memory allocation
  - Avoid dynamic multiple dispatch

Julia — Compilation



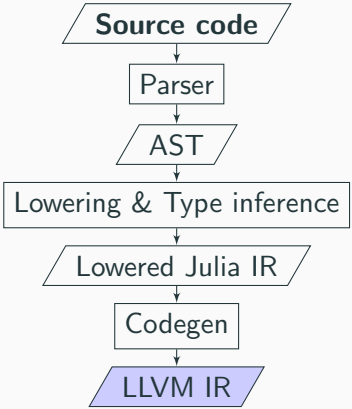
- Type inference
  - Allows static memory allocation
  - Avoid dynamic multiple dispatch

Julia — Compilation



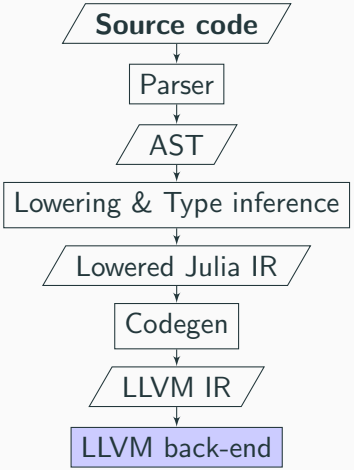
- Type inference
  - Allows static memory allocation
  - Avoid dynamic multiple dispatch

Julia — Compilation



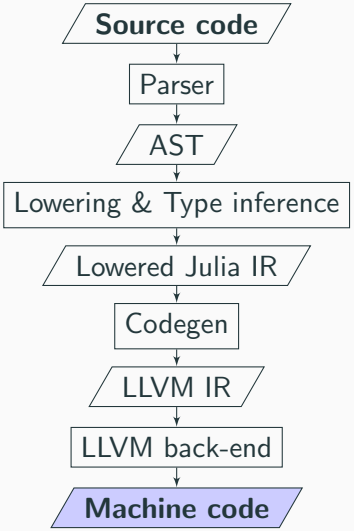
- Type inference
  - Allows static memory allocation
  - Avoid dynamic multiple dispatch
- LLVM
  - Optimization
  - Platform independence

Julia — Compilation



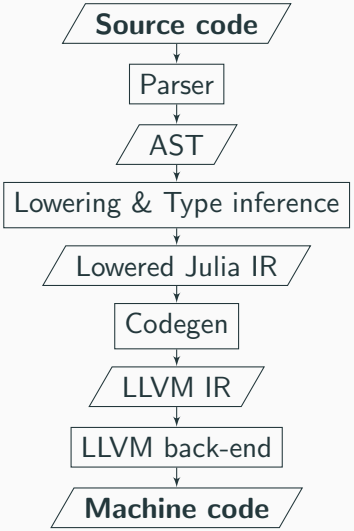
- Type inference
  - Allows static memory allocation
  - Avoid dynamic multiple dispatch
- LLVM
  - Optimization
  - Platform independence

Julia — Compilation



- Type inference
  - Allows static memory allocation
  - Avoid dynamic multiple dispatch
- LLVM
  - Optimization
  - Platform independence

Julia — Compilation

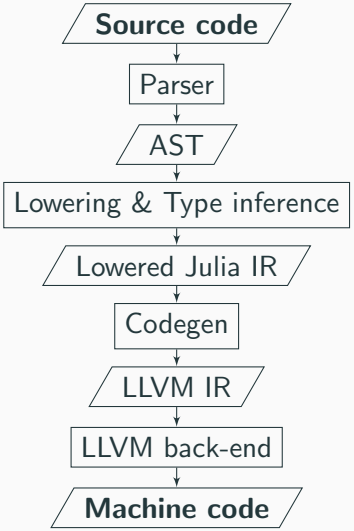


From [BFD18]

- Type inference
  - Allows static memory allocation
  - Avoid dynamic multiple dispatch
- LLVM
  - Optimization
  - Platform independence

- Dynamic code where it's convenient

Julia — Compilation



From [BFD18]

- Type inference
  - Allows static memory allocation
  - Avoid dynamic multiple dispatch
- LLVM
  - Optimization
  - Platform independence

- Dynamic code where it's convenient
- Speed where it's necessary

Julia — high-level features and idioms

1. Memory management

Julia — high-level features and idioms

1. Memory management
2. Array bounds checking

Julia — high-level features and idioms

1. Memory management
2. Array bounds checking
   `@inbounds` to circumvent

Julia — high-level features and idioms

1. Memory management
2. Array bounds checking
   `@inbounds` to circumvent
3. Operators and Functions on complex types

Julia — high-level features and idioms

1. Memory management
2. Array bounds checking
   `@inbounds` to circumvent
3. Operators and Functions on complex types
   `A\b` $\Leftrightarrow$ $Ax = b$

Julia — high-level features and idioms

1. Memory management
2. Array bounds checking
   `@inbounds` to circumvent
3. Operators and Functions on complex types
   `A\b` $\Leftrightarrow$ $Ax = b$
   `[3,3,12]+[2,4,10]`

Julia — high-level features and idioms

1. Memory management
2. Array bounds checking
   `@inbounds` to circumvent
3. Operators and Functions on complex types
   `A\b` $\Leftrightarrow$ $Ax = b$
   `[3,3,12]+[2,4,10]`
4. Multi-threading and vectorization

Julia — high-level features and idioms

1. Memory management
2. Array bounds checking
   `@inbounds` to circumvent
3. Operators and Functions on complex types
   `A\b` $\Leftrightarrow$ $Ax = b$
   `[3,3,12]+[2,4,10]`
4. Multi-threading and vectorization
   `@threads`

Julia — high-level features and idioms

1. Memory management
2. Array bounds checking
   `@inbounds` to circumvent
3. Operators and Functions on complex types
   `A\b` $\Leftrightarrow$ $Ax = b$
   `[3,3,12]+[2,4,10]`
4. Multi-threading and vectorization
   `@threads`, `@simd`

- Accomplished through packages

Julia — Accelerator programming

- Accomplished through packages
  - GPU-specific packages eg. `CUDA.jl`, `AMDGPU.jl`, ...

Julia — Accelerator programming

- Accomplished through packages
  - GPU-specific packages eg. `CUDA.jl`, `AMDGPU.jl`, ...
  - Hook into compilation process through extension interfaces (e.g. disallow exceptions)

Julia — Accelerator programming

- Accomplished through packages
  - GPU-specific packages eg. `CUDA.jl`, `AMDGPU.jl`, ...
  - Hook into compilation process through extension interfaces (e.g. disallow exceptions)
  - Different LLVM backends to produce GPU code

Julia — Accelerator programming

- Accomplished through packages
  - GPU-specific packages eg. `CUDA.jl`, `AMDGPU.jl`, ...
  - Hook into compilation process through extension interfaces (e.g. disallow exceptions)
  - Different LLVM backends to produce GPU code
- API uniformity

Benchmarks

Compute bound — miniBUDE

- Molecular dynamics based application

Compute bound — miniBUDE

- Molecular dynamics based application
- Predict structure of an arrangement of 2 molecules (“docking”)

Compute bound — miniBUDE

- Molecular dynamics based application
- Predict structure of an arrangement of 2 molecules ("docking")
- Trigonometric function evaluations, square roots, absolute values

Compute bound — miniBUDE

- Molecular dynamics based application
- Predict structure of an arrangement of 2 molecules ("docking")
- Trigonometric function evaluations, square roots, absolute values

Highly compute bound

Memory bound — BabelStream

Memory bound — BabelStream

- Vector operations

Memory bound — BabelStream

- Vector operations
- Simple computations

Memory bound — BabelStream

- Vector operations
- Simple computations
- Number of array accesses much more significant

Memory bound — BabelStream

- Vector operations
- Simple computations
- Number of array accesses much more significant

procedure $\mathrm{MUL}(A[n], C[n], scalar, n)$
    for $i \leftarrow 0, n$ **do**
        $C[i] \leftarrow scalar * A[i]$

Taken from [LM21]

Memory bound — BabelStream

- Vector operations
- Simple computations
- Number of array accesses much more significant

procedure $\mathrm{MUL}(A[n], C[n], scalar, n)$
    for $i \leftarrow 0, n$ **do**
        $C[i] \leftarrow scalar * A[i]$

Memory-bandwidth bound

Taken from [LM21]

Inter-node communication — ReproMPI

- **Message Passing Interface** (**MPI**): Inter-node communication standard

Inter-node communication — ReproMPI

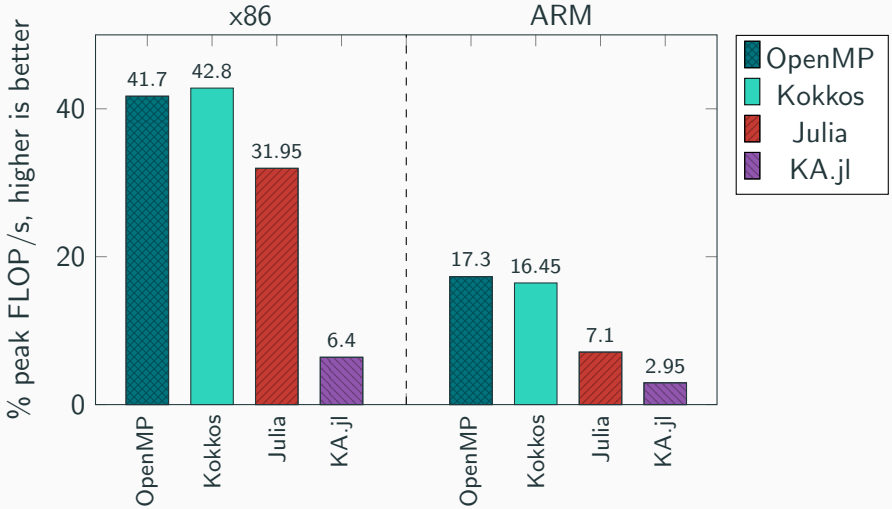
- **Message Passing Interface** (**MPI**): Inter-node communication standard
- ReproMPI

Inter-node communication — ReproMPI

- **Message Passing Interface** (**MPI**): Inter-node communication standard
- ReproMPI
- Varying message sizes

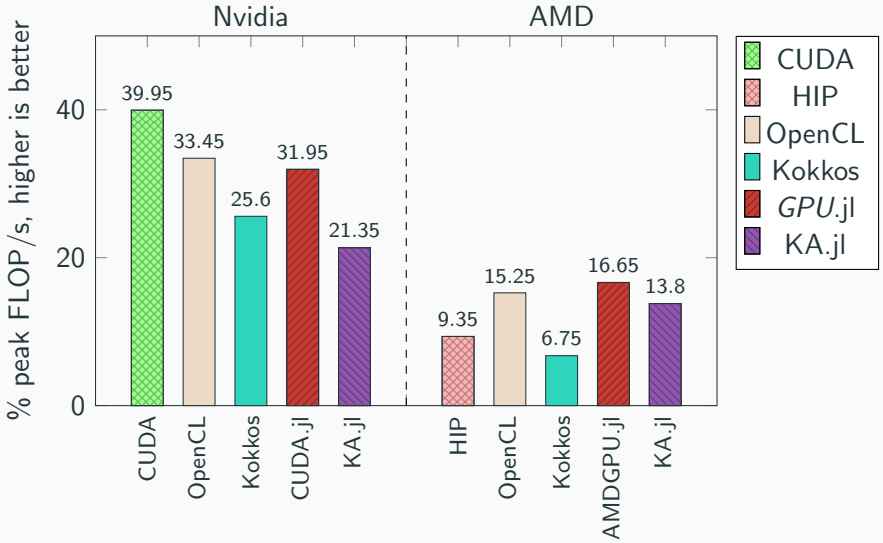
Performance Results

Compute bound — miniBUDE: CPU



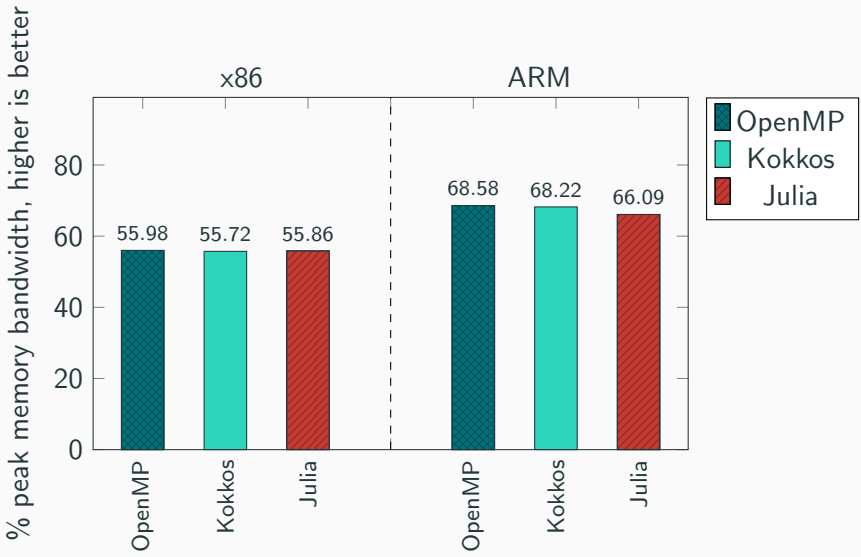
Adapted from [LM21]

Compute bound — miniBUDE: GPU



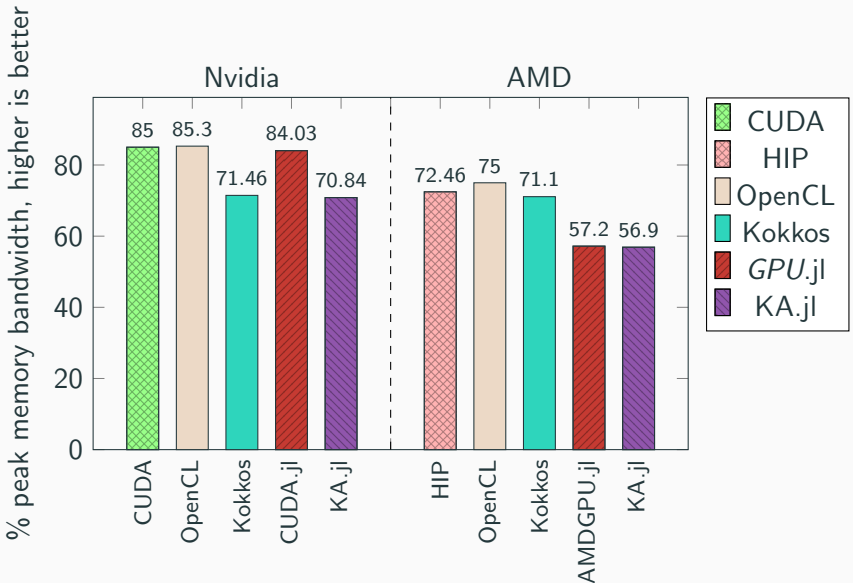
Adapted from [LM21]

Memory bound — BabelStream: CPU



Adapted from [LM21]

Memory bound — BabelStream: GPU



Adapted from [LM21]

Inter-node communication — MPI

- Open MPI C and MPI.jl (Julia) performed almost identically, across message sizes

Taken from [HS20]

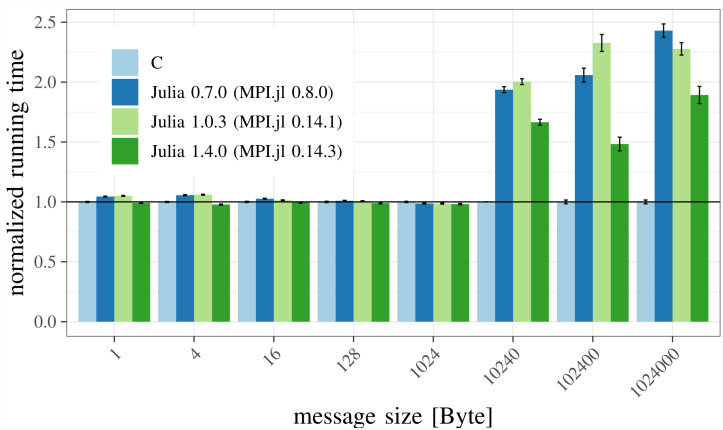
Inter-node communication — MPI

- Open MPI C and MPI.jl (Julia) performed almost identically, across message sizes
- Exception: `MPI_Allreduce`

Taken from [HS20]

Inter-node communication — MPI

- Open MPI C and MPI.jl (Julia) performed almost identically, across message sizes
- Exception: `MPI_Allreduce`



Taken from [HS20]

Discussion

Performance

- Julia generally not lacking far behind in intra-node workloads

Performance

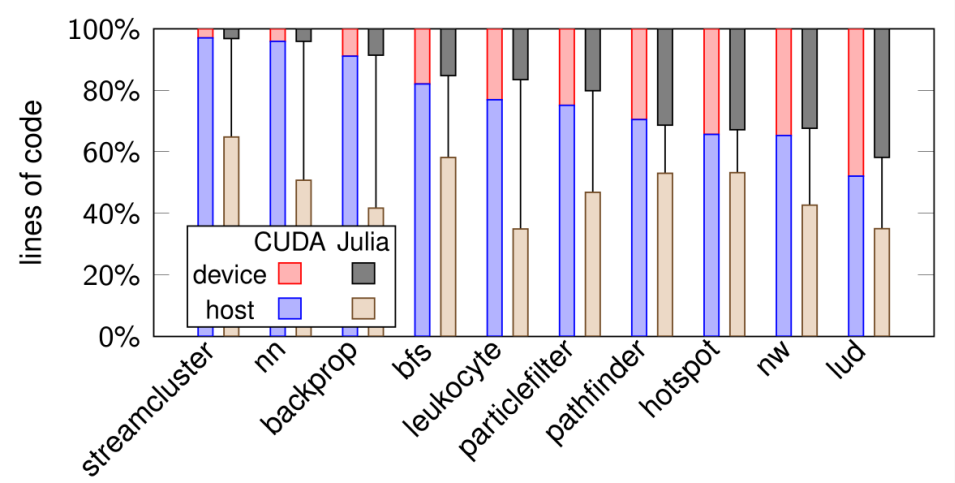
- Julia generally not lacking far behind in intra-node workloads
- Communication performance for inter-node workloads basically identical, except for the `MPI_Allreduce` anomaly

Lines of code

- `CUDANative.jl`, example kernels:

Lines of code

- `CUDANative.jl`, example kernels:



Taken from [BFD18]

Portability

- LLVM and JIT compilation make portability easier to implement, hopefully future-proof

Portability

- LLVM and JIT compilation make portability easier to implement, hopefully future-proof
- Single-source portability on GPUs constrained to the individual GPU packages

Portability

- LLVM and JIT compilation make portability easier to implement, hopefully future-proof
- Single-source portability on GPUs constrained to the individual GPU packages
- Manual porting between GPU packages relatively easy, often simple API call substitutions

Conclusion

- JIT, LLVM, and compiler extensions mean speed, portability, code reuse

Conclusion

- JIT, LLVM, and compiler extensions mean speed, portability, code reuse
- Convenience through dynamic code and high-level idioms

Conclusion

- JIT, LLVM, and compiler extensions mean speed, portability, code reuse
- Convenience through dynamic code and high-level idioms
- Julia is competitive in HPC, but porting still requires some manual effort

Conclusion

- JIT, LLVM, and compiler extensions mean speed, portability, code reuse
- Convenience through dynamic code and high-level idioms
- Julia is competitive in HPC, but porting still requires some manual effort
- Performance itself is quite good across architectures and nodes

References i

[Bez+12] Jeff Bezanson et al. “Julia: A fast dynamic language for technical computing”. In: arXiv preprint (2012). DOI: 10.48550/ARXIV.1209.5145.

[BFD18] Tim Besard, Christophe Foket, and Bjorn De Sutter. “Effective Extensible Programming: Unleashing Julia on GPUs”. In: IEEE Transactions on Parallel and Distributed Systems (2018). ISSN: 1045-9219. DOI: 10.1109/TPDS.2018.2872064. arXiv: 1712.03112 [cs.PL].

References ii

[LM21] Wei-Chen Lin and Simon McIntosh-Smith. "Comparing Julia to Performance Portable Parallel Programming Models for HPC". In: 2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). 2021, pp. 94–105. DOI: 10.1109/PMBS54543.2021.00016.

[HS20] Sascha Hunold and Sebastian Steiner. "Benchmarking Julia's Communication Performance: Is Julia HPC ready or Full HPC?" In: 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). 2020, pp. 20–25. DOI: 10.1109/PMBS51919.2020.00008.

References iii

[PLM21] Andrei Poenaru, Wei-Chen Lin, and Simon McIntosh-Smith. "A Performance Analysis of Modern Parallel Programming Models Using a Compute-Bound Application". In: 36th International Conference, ISC High Performance 2021. Frankfurt, Germany, 2021.

[McI+15] Simon McIntosh-Smith et al. "High performance in silico virtual drug screening on many-core processors". In: The International Journal of High Performance Computing Applications 29.2 (2015). PMID: 25972727, pp. 119–134. DOI: 10.1177/1094342014528252.

[McC+95] John D McCalpin et al. "Memory bandwidth and machine balance in current high performance computers". In: IEEE computer society technical committee on computer architecture (TCCA) newsletter 2.19-25 (1995).

[Dea+18] T Deakin et al. “Evaluating attainable memory bandwidth of parallel programming models via BabelStream”. In: International Journal of Computational Science and Engineering 17.3 (Special issue 2018), pp. 247–262. DOI: 10.1504/IJCSE.2018.095847.

GPU Kernel API Cross-Refrence i

	CUDA.jl	AMDGPU.jl
Global size	`blockDim().x * gridDim().x`	`gridDim().x`
Group count	`gridDim().x`	`gridDimWG().x`
Group size	`blockDim()`	`workgroupDim().x`

Adapted from [LM21]

GPU Kernel API Cross-Refrence ii

	CUDA.jl	AMDGPU.jl	oneAPI.jl	KernelAbstractions.jl
Global size	`blockDim().x*gridDim().x`	`gridDim().x`	`get_global_size(0)`	(Not exposed)
Group count	`gridDim().x`	`gridDimWG().x`	`get_num_groups(0)`	(Not exposed)
Group size	`blockDim().x`	`workgroupDim().x`	`get_local_size(0)`	`@groupsize()[1]`
Index (global)	`(blockIdx().x - 1) * blockDim().x + threadIdx().x`	`(workgroupIdx().x - 1) * workgroupDim().x + workitemIdx().x`	`get_global_id(0)`	`@index(Global)`
Index (group)	`blockIdx().x`	`workgroupIdx().x`	`get_group_id(0)`	`@index(Group)`
Index (local)	`threadIdx().x`	`workitemIdx().x`	`get_local_id(0)`	`@index(Local)`
Synchronise	`sync_threads()`	`sync_workgroup()`	`barrier()`	`@synchronize`
Memory (register) size=N, type=T	`data = MArray{Tuple{N},T}`	`data = MArray{Tuple{N},T}`	`data = MArray{Tuple{N},T}`	`data = MArray{Tuple{N},T}`
Memory (private) size=N, type=T	`data = @cuStaticSharedMem(T, N)`	`data = ROCDeviceArray((N,), alloc_local(:data,T, N))`	`data = @LocalMemory(T, (N,))`	`data = @localmem T N`

Kernel launch & Mem. allocation Groups=GN, Blocks=BN, size=N, type=T	`data = CuArray{T}(undef, N)`
	`@cuda`
	`  blocks=N÷GN`
	`  threads=GN`
	`  kernel(data)`
		`data = ROCArray{T}(undef, N)`
		`@roc`
		`  groupsize=GN`
		`  gridsize=GN*BN`
		`  kernel(data)`
			`data = oneArray{T}(undef, N)`
			`@oneapi`
			`  items=GN`
			`  groups=BN`
			`  kernel(data)`
				`device= # one of :`
				`  # CPU()`
				`  # CUDADevice()`
				`  # ROCDevice()`
				`data= # one of :`
				`  # Array{T}`
				`  # CuArray{T}`
				`  # ROCArray{T}`
				`kernel(device, GN)(`
				`  data, ndrange=N)`

Taken from [LM21]

All BabelStream Kernels

procedure $\mathrm{COPY}(A[n], C[n], n)$
  for $i \leftarrow 0, n$ **do**
    $C[i] \leftarrow A[i]$

procedure $\mathrm{ADD}(A[n], B[n], C[n], n)$
  for $i \leftarrow 0, n$ **do**
    $C[i] \leftarrow A[i] + B[i]$

procedure $\mathrm{TRIAD}(A[n], B[n], C[n], scalar, n)$
  for $i \leftarrow 0, n$ **do**
    $C[i] \leftarrow A[i] + (scalar * B[i])$

procedure $\mathrm{MUL}(A[n], C[n], scalar, n)$
  for $i \leftarrow 0, n$ **do**
    $C[i] \leftarrow scalar * A[i]$

procedure $\mathrm{DOT}(A[n], B[n], scalar, n)$
  for $i \leftarrow 0, n$ **do**
    $R \leftarrow R + (A[i] * B[i])$
  return R

Taken from [LM21]

ReproMPI

- `MPI_Bcast`: Push data into network from one source (rooted)
- `MPI_Allreduce`: Reduce data from all sources into one result (non-rooted collective)
- `MPI_Alltoall`: All processes exchange data with all other processes (non-rooted collective)

Platform details

Vendor	Name	Architecture	Abbreviation	Device Type	Theoretical Peak Mem. Bandwidth (GB/s)	Theoretical Peak FP32 FLOP/s (GFLOP/s)
Intel	Xeon Gold 6230	Cascade Lake	Xeon	HPC CPU (20C*2, 2S)	281.6	4096
AMD	EPYC 7742	Zen2 (Rome)	EPYC	HPC CPU (64C*2, 2S)	409.6	9216
Marvell	ThunderX2	Vulcan	TX2	HPC CPU (32C*2, 2S)	288	2560
Fujitsu	A64FX	(Custom ARMv8)	A64FX	HPC CPU (48C*2, 2S)	1024	5530
NVIDIA	Tesla A100 (SXM 80GB)	Ampere	A100	HPC GPU	2039	19490
NVIDIA	Tesla V100 (PCIe 16GB)	Volta	V100	HPC GPU	900	14130
NVIDIA	RTX2080Ti	Turing	2080Ti	Consumer GPU	616	13800
AMD	Instinct MI100	CDNA	MI100	HPC GPU	1228	23100
AMD	Instinct MI50	GCN (Vega 20GL)	MI50	HPC GPU	1024	13300
AMD	Radeon VII	GCN (Vega 20)	RadeonVII	Consumer GPU	1024	13800
Intel	UHD P630 (Xeon E2176G)	Gen9.5	UHD	Server iGPU	42.6	460
Intel	IrisPro 580 (i7 6670HQ)	Gen9	IrisPro	Consumer iGPU	34	1094
Apple	M1	Firestorm+Icestorm	M1	Consumer CPU(4+4, 1S)	68.25	673

Taken from [LM21]

`@threads` vs. OpenMP

```
xs = Vector{Float32}(undef, N)
ys = fill(42f0, N)
Threads.@threads for i = 1:N
  @inbounds xs[i] = ys[i]
end
```

```
std::vector<float> xs(N, 0);
std::vector<float> ys(N, 42.f);
#pragma omp parallel for
for (int i = 0; i < N; i++)
  xs[i] = ys[i];
```

Taken from [LM21]