JuliaHub

oneAPI programming in Julia with oneAPI.jl

Tim Besard

Why Julia?

High-level programming language designed for performance

Why Julia?

High-level programming language designed for performance

```
julia> data = (1, rand())
(1, 0.5326182923218289)

julia> sum(data)
1.532618292321829
```

Why Julia?

High-level programming language
designed for performance

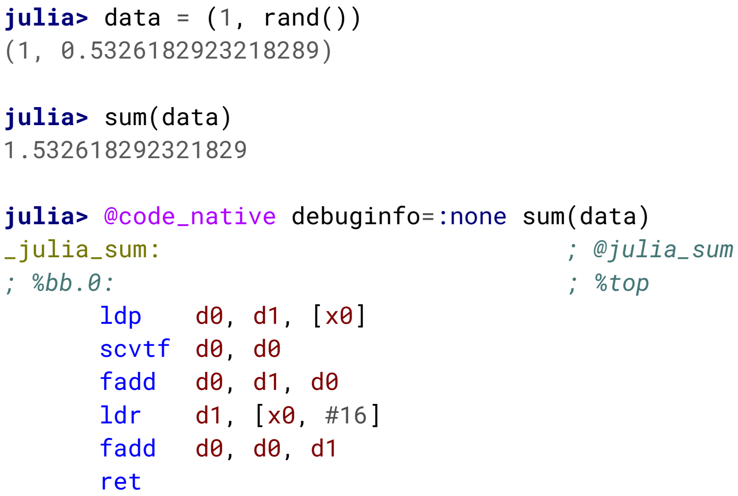
```
julia> data = (1, rand())
(1, 0.5326182923218289)

julia> sum(data)
1.532618292321829

julia> @code_llvm sum(data)
define double @julia_sum({ i64, double }* nocapture nonnull readonly align 8 dereferenceable(16) %0) #0 {
top:
  %1 = getelementptr inbounds { i64, double }, { i64, double }* %0, i64 0, i32 0
  %2 = getelementptr inbounds { i64, double }, { i64, double }* %0, i64 0, i32 1
  %3 = load i64, i64* %1, align 8
  %4 = sitofp i64 %3 to double
  %5 = load double, double* %2, align 8
  %6 = fadd double %5, %4
  ret double %6
}
```

Why Julia?

High-level programming language
designed for performance



```
julia> data = (1, rand())
(1, 0.5326182923218289)

julia> sum(data)
1.532618292321829

julia> @code_native debuginfo=:none sum(data)
_julia_sum:                            ; @julia_sum
; %bb.0:                               ; %top
       ldp    d0, d1, [x0]
       scvtf  d0, d0
       fadd   d0, d1, d0
       ldr    d1, [x0, #16]
       fadd   d0, d0, d1
       ret
```

GPU support in Julia

GPU-enabled applications

- Flux.jl (deep learning)
- CLiMA (ocean modeling)
- DifferentialEquations.jl
- Yao.jl (quantum information)
- …

oneAPI.jl

Metal.jl

AMDGPU.jl

CUDA.jl

Shared infrastructure

- GPUCompiler.jl
- GPUArrays.jl
- KernelAbstractions.jl
- …

Easy to get started

1. Download and unpack Julia 1.8: https://julialang.org/downloads/
2. Launch Julia and enter the the package manager

```
pkg> add oneAPI
```

3. Import and verify the oneAPI.jl package

```
julia> using oneAPI
Downloading artifacts: ...

julia> oneAPI.versioninfo()
Binary dependencies:
- NEO_jll: 22.53.25242+0
- libigc_jll: 1.0.12812+0
- …

1 device:
- Intel(R) Arc(TM) A770 Graphics [0x56a0]
```

Automatic download of binary dependencies

Array abstraction

```
julia> vec = oneArray([1])
1-element oneVector{Int64, oneL0.DeviceBuffer}:
 1
```

`oneArray` serves multiple purposes:

1. container for device memory
2. abstraction for data-parallel programming

```
julia> vec .+ 1
1-element oneVector{Int64, oneL0.DeviceBuffer}:
 2
```

Array abstraction

Linear algebra:

```
julia> using LinearAlgebra

julia> vec = oneVector(rand(Float32, 2))
       dot(vec, vec)

julia> mat = oneMatrix(rand(Float32, 2, 2))
       mat * mat
```

Higher-order functions:

```
julia> map(vec) do val
           val + 1
       end

julia> reduce(+, vec)
```

Obviates kernel programming!

Statistics:

```
julia> using Statistics

julia> mean(mat)

julia> std(mat)
```

Kernel programming

```julia
function vadd(a, b, c)
   function kernel(d_a, d_b, d_c)
       i = get_global_id()
       d_c[i] = d_a[i] + d_b[i]
       return
   end

   d_a = oneArray(a)
   d_b = oneArray(b)
   d_c = oneArray(c)

   len = prod(size(a))
   @oneapi items=len kernel(d_a, d_b, d_c)
   c .= Array(d_c)
end
```

Similar to CUDA.jl, AMDGPU.jl, ...

Differences with DPC++/SYCL

- OpenCL intrinsics
- Global semantics

Kernel programming

```julia
function vadd(a, b, c)
   function kernel(d_a, d_b, d_c)
       i = get_global_id()
       d_c[i] = d_a[i] + d_b[i]
       return
   end

   d_a = oneArray(a)
   d_b = oneArray(b)
   d_c = oneArray(c)

   len = prod(size(a))
   @oneapi items=len kernel(d_a, d_b, d_c)
   c .= Array(d_c)
end
```

```
julia> @device_code_llvm vadd([1], [2], [0])

define spir_kernel void @kernel(
    { { [1 x i64], i8 addrspace(1)* } }* byval,
    { { [1 x i64], i8 addrspace(1)* } }* byval,
    { { [1 x i64], i8 addrspace(1)* } }* byval
   ) local_unnamed_addr {
entry:
 %3 = call i64 @_Z13get_global_idj(i32 0)
 ...
 store i64 %14, i64 addrspace(1)* %18, align 8
 ret void
}
```

Kernel programming

```julia
function vadd(a, b, c)
   function kernel(d_a, d_b, d_c)
       i = get_global_id()
       d_c[i] = d_a[i] + d_b[i]
       return
   end

   d_a = oneArray(a)
   d_b = oneArray(b)
   d_c = oneArray(c)

   len = prod(size(a))
   @oneapi items=len kernel(d_a, d_b, d_c)
   c .= Array(d_c)
end
```

```
julia> @device_code_spirv vadd([1], [2], [0])

; SPIR-V
; Version: 1.0
; Bound: 45
; Schema: 0
              OpCapability Addresses
              OpCapability Linkage
              OpCapability Kernel
              ...
              OpStore %43 %39 Aligned 8
              OpReturn
              OpFunctionEnd
```

Level Zero wrappers

```
julia> using .oneL0

julia> drivers()
ZeDriver iterator for 1 drivers:
1. ZeDriver(00000000-0000-0000-174c-dd890103629a): version 1.3.25242
julia> drv = first(drivers());

julia> devices(drv)
ZeDevice iterator for 1 devices:
1. Intel(R) Arc(TM) A770 Graphics [0x56a0]
julia> dev = first(devices(drv));

julia> queue = ZeCommandQueue(ctx, dev);
julia> execute!(queue) do list
           append_barrier!(list)
       end
```

Issues developing oneAPI.jl

✗ SPIR-V fragility
  - SPIRV-LLVM-Translator: incomplete LLVM support, invalid SPIR-V
  - Easy to trigger IGC aborts

✗ Math libraries (oneMKL, oneDNN, etc) are problematic
  - lack of C APIs
  - difficult to redistribute (cf. libcublas.so)
  - assumes SYCL environment (events, queues, etc)

Work in progress

- Performance: optimization for Intel hardware
- Integration with performance tools (VTune)
- Platform support: Linux & Intel GPUs only

Try it out!

https://github.com/JuliaGPU/oneAPI.jl