intro_cuda

March 17, 2022

Programmer en CUDA avec Julia

Marc Fuentes : SED de l'INRIA de l'UPPA

1 Installation

- sur un ordi perso, le gestionnaire de paquets de Julia `Pkg` va télécharger des artefacts

```
using Pkg
Pkg.add("CUDA")
```

- sur plafrim (pour ce TP) on peut utiliser sur GPU Pascal ou Volta (`salloc -C "sirocco&p100"`)

```
> module load language/julia/1.7.2
> julia
```

- certaines variables peuvent influer sur la detection de l'installation cuda
- `JULIA_CUDA_VERSION`
- `JULIA_CUDA_BUILDBINARY=false`

[1]:
```
# verifier la config
using CUDA
CUDA.versioninfo()
```

```
CUDA toolkit 11.4.1, artifact installation
CUDA driver 11.6.0
NVIDIA driver 510.54.0

Libraries:
- CUBLAS: 11.5.4
- CURAND: 10.2.5
- CUFFT: 10.5.1
- CUSOLVER: 11.2.0
- CUSPARSE: 11.6.0
- CUPTI: 14.0.0
- NVML: 11.0.0+510.54
- CUDNN: 8.20.2 (for CUDA 11.4.0)
- CUTENSOR: 1.3.0 (for CUDA 11.2.0)

Toolchain:
```

```
- Julia: 1.7.0-beta3
- LLVM: 12.0.0
- PTX ISA support: 3.2, 4.0, 4.1, 4.2, 4.3, 5.0, 6.0, 6.1, 6.3, 6.4, 6.5, 7.0
- Device capability support: sm_35, sm_37, sm_50, sm_52, sm_53, sm_60, sm_61,
sm_62, sm_70, sm_72, sm_75, sm_80

1 device:
  0: Quadro T2000 with Max-Q Design (sm_75, 3.815 GiB / 4.000 GiB available)
```

2 Compilation et arrière-boutique GPU

- l'interprète Julia integre un compilateur «à la volée» basé sur llvm
- le paquet CUDA.jl est basé sur des paquets de plus bas-niveau pour compiler le code vers le GPU

3 GPU : généralités sur l'architecture

- le GPU est un accélérateur possédant sa mémoire (DRAM) et un grand nombre de «fils d'exécution» (threads)
- quelques principes à retenir
- le parallèlisme GPU a pour cible beaucoup de tâches élémentaires identiques (grain fin)
- limiter les transferts (ou les recouvrir par des calculs)
- assurer la contiguïté des données en mémoires (coalescence)
- donner suffisament de grain a moudre au GPU (calcul vectoriel) (occupation)
- éviter les divergences de branches # Paradigme de programmation sur GPU :
- remplacer un indice de boucle par un indice de «thread»

```
for i=...
    a[i] = ...
end
```

devient ainsi

```
i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
a[i] = ...
```

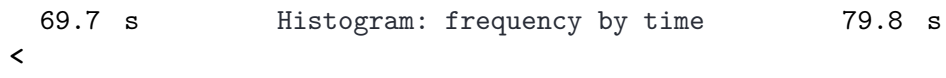
- illustration de la numérotation 1D

4 Parallélisme implicite

- il suffit d'avoir recours a des abstractions parallèles agissant sur le conteneur `CuArray`

[2]:
```
#version GPU
using BenchmarkTools
N = 2^10*32
A = CuArray([1:N;])
B = CuArray([0:N-1;])
@benchmark z = reduce(+, A.^3+B.^2-2 * A .* B)
```

[2]:
```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):
69.679  s …  37.112 ms    GC
(min … max): 0.00% … 19.60%
 Time  (median):     74.266  s
 GC (median):    0.00%
 Time  (mean ± ):
85.548  s ± 632.854  s    GC
(mean ± ):  2.43% ±  0.33%
```




```
<

 Memory estimate: 8.77 KiB, allocs estimate:
203.
```

[3]:
```
# version CPU
A = [1:N;]
B = [0:N-1;]
@benchmark z = reduce(+, A.^2+B.^2-2 * A .* B)
```

[3]:
```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):
130.679  s …   2.240 ms    GC
(min … max):  0.00% … 93.68%
 Time  (median):     144.859  s
 GC (median):     0.00%
 Time  (mean ± ):
187.424  s ± 220.662  s    GC
(mean ± ):  21.86% ± 16.08%
```

```
  131  s        Histogram:
log(frequency) by
time       1.36 ms <

 Memory estimate: 1.50 MiB, allocs estimate:
23.
```

Attention avec ce paradigme il faut eviter d'acceder individuellement aux indices!

[4]:
```
A = CuArray([1:1000;])
s = 0
#CUDA.allowscalar(false) tweak that!
for i =1:1000
   s += A[i]
end
s
```

```
 Warning: Performing scalar indexing on task Task (runnable)
@0x00007f76492d8e70.
 Invocation of getindex resulted in scalar indexing of a GPU array.
 This is typically caused by calling an iterating implementation of a method.
 Such implementations *do not* execute on the GPU, but very slowly on the CPU,
 and therefore are only permitted from the REPL for prototyping purposes.
 If you did intend to index this array, annotate the caller with @allowscalar.
 @ GPUArrays /home/fux/.julia/packages/GPUArrays/UBzTm/src/host/indexing.jl:56
```

[4]:
```
500500
```

5 Parallèlisme explicite

- on code et on appelle explicitement un «noyau» sur le GPU
- noyau : routine s'executant sur le GPU que chacun des threads va executer «individuellement»
- l'appel du noyau se fait au moyen de la macro `@cuda` en passant en paramètre le nombre de blocs et le nombre de threads/bloc

```
@cuda threads=nThreads blocks=nbBlocks ma_routine!(a,b)
```

- `nThreads` et `nbBlocks` peuvent etre des couples ou des triplets (grille 2D ou 3D)
- la fonction noyau doit se terminer OBLIGATOIREMENT par un `return`

[5]:
```
# exemple noyau d'homothetie a ← a
function scale_gpu!(a, )
  i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
  if (i <= size(a, 1))
    a[i] *=
  end
  return
end
```

[5]:
```
scale_gpu! (generic function with 1 method)
```

[6]:
```
# appel du noyau
using Test
x= CUDA.ones(4096)
@cuda threads=512 blocks=cld(4096, 512) scale_gpu!(x, 4.0f0)
@test sum((x .- 4.0f0).^2) < 1e-12
```

[6]:
```
Test Passed
  Expression: sum((x .- 4.0f0) .^ 2) < 1.0e-12
   Evaluated: 0.0f0 < 1.0e-12
```

6 heuristique pour l'occupation

[36]:
```
CUDA.attribute(device(), CUDA.DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK)
```

[36]:
```
1024
```

[37]:
```
noyau = @cuda launch=false scale_gpu!(x, 4.0f0)
```

[37]:
```
CUDA.HostKernel{typeof(scale_gpu!), Tuple{CuDeviceVector{Float32, 1},
Float32}}(scale_gpu!, CuContext(0x000055f52e30b8e0, instance 606ac60cd4240541),
CuModule(Ptr{Nothing} @0x000055f52f2778c0, CuContext(0x000055f52e30b8e0,
instance 606ac60cd4240541)), CuFunction(Ptr{Nothing} @0x000055f5332a5bf0,
CuModule(Ptr{Nothing} @0x000055f52f2778c0, CuContext(0x000055f52e30b8e0,
instance 606ac60cd4240541))))
```

[38]:
```
config = CUDA.launch_configuration(noyau.fun)
```

[38]:
```
(blocks = 16, threads = 1024)
```

[42]:
```
@show nThreads = min(length(x), config.threads)
@show nBlocks = cld(length(x), nThreads)
x = CUDA.ones(4096)
noyau(x, 4.0f0; threads=nThreads, blocks=nBlocks)
@test sum((x .- 4.0f0).^2) < 1e-12
```

```
nThreads = min(length(x), config.threads) = 1024
nBlocks = cld(length(x), nThreads) = 4
```

[42]:
```
Test Passed
  Expression: sum((x .- 4.0f0) .^ 2) < 1.0e-12
   Evaluated: 0.0f0 < 1.0e-12
```

[46]:
```
CUDA.occupancy(noyau.fun, nThreads)
```

[46]: 1.0

7 Données trop grosses

- que faire si N > nBlocks * nThreads ?
- on peut utiliser une boucle avec un pas utilisant la taille de la grille

[65]:
```
N2 = 32 * 1024 * 1024
x = CUDA.ones(N2)
@cuda threads=1024 blocks=cld(N, 1024) scale_gpu!(x, 4.0f0)
@test sum((x .- 4.0f0).^2) < 1e-12
```

```
Test Failed at In[65]:4
  Expression: sum((x .- 4.0f0) .^ 2) < 1.0e-12
   Evaluated: 3.0195302f8 < 1.0e-12
```

```
There was an error during testing

Stacktrace:
 [1] record(ts::Test.FallbackTestSet, t::Union{Test.Error, Test.Fail})
   @ Test ~/iturriak/julia/usr/share/julia/stdlib/v1.7/Test/src/Test.jl:903
 [2] do_test(result::Test.ExecutionResult, orig_expr::Any)
   @ Test ~/iturriak/julia/usr/share/julia/stdlib/v1.7/Test/src/Test.jl:637
 [3] top-level scope
   @ ~/iturriak/julia/usr/share/julia/stdlib/v1.7/Test/src/Test.jl:445
 [4] eval
   @ ./boot.jl:373 [inlined]
 [5] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
 ↪filename::String)
   @ Base ./loading.jl:1196
```

[67]:
```
x = CUDA.ones(N2)
function scale_gpu2!(a, )
  i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
  for j=i:gridDim().x * blockDim().x: size(a, 1) # maintenant on gère des␣
 ↪données plus grosses que la grille
    a[j] *=
  end
  return
end
@cuda threads=1024 blocks=cld(N, 1024) scale_gpu2!(x, 4.0f0)
@test sum((x .- 4.0f0).^2) < 1e-12
```

[67]:
```
Test Passed
  Expression: sum((x .- 4.0f0) .^ 2) < 1.0e-12
   Evaluated: 0.0f0 < 1.0e-12
```

8 Résolution de l'équation de laplace en 2D par Jacobi

- On se propose de résoudre l'équation

$$\Delta\Phi = \frac{\partial^2\Phi}{\partial x^2} + \frac{\partial^2\Phi}{\partial y^2} = 0$$

sur le carré $[0,1]^2$

- Pour cela on discrétise le carré $[0,1]^2$avec un pas de taille $h = \frac{1}{n+1}$
- on utilise le schéma d'ordre suivant (Ferziger,1981) qui approxime le laplacien par un opérateur H à l'ordre 4
- En décomposant $H = D - F$, le schéma itératif de Jacobi donne

$$\Phi^{k+1} = D^{-1}F\Phi^k = J\Phi^k$$

avec

$$J\Phi = \frac{1}{20}\left[\Phi_{i-1,j-1} + \Phi_{i-1,j+1} + \Phi_{i+1,j+1} + \Phi_{i+1,j-1}\right] + \frac{1}{5}\left[\Phi_{i,j-1} + \Phi_{i,j+1} + \Phi_{i+1,j} + \Phi_{i-1,j}\right]$$

[7]:
```
function jacobi_gpu!(ap, a)
    i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
    j = threadIdx().y + (blockIdx().y - 1) * blockDim().y
    if ((i >= 2) && (i <= (size(a,1)-1)) && (j >= 2) && (j <= (size(a,2)-1)))
      ap[i,j] = 0.2f0 * (a[i-1,j]  + a[i+1,j]   + a[i,j-1]  + a[i,j+1])  +
                0.05f0 * (a[i-1,j-1]+ a[i+1,j-1] + a[i-1,j+1] + a[i+1,j+1])
    end
    return
end
```

[7]:
```
jacobi_gpu! (generic function with 1 method)
```

on initialise les bords

[8]:
```
function init_sol!(a)
    a .= 0.0f0
    m = size(a,1)
    y  = sin.( *[0:m-1;] ./ (m))
    a[:,1] = y
    a[:,end]= y  * exp(- )
end
```

[8]:
```
init_sol! (generic function with 1 method)
```

[9]:
```
N = 4096
a = CuArray{Float32}(undef, N, N)
ap = similar(a);
```

```
const BLOCK_X = 32
const BLOCK_Y = 16
```

[9]: 16

[27]:
```
function chrono_gpu!(a, ap, aff)
  init_sol!(a);
  init_sol!(ap);
  nThreads = 32
  for i = 1:100
     @cuda threads=(BLOCK_X,BLOCK_Y) blocks=(cld(N,BLOCK_X), cld(N, BLOCK_Y))␣
 ↪jacobi_gpu!(ap,a)
     error = reduce(max, abs.(ap-a))
     if (aff != 0 && (i % aff) == 0)
          println("i =", i, " error = ", error)
     end
     if (error<=1e-3)
       break
     end
     a = copy(ap)
  end

end
chrono_gpu!(a, ap, 20 )
```

```
i =20 error = 0.011931241
i =40 error = 0.0060647726
i =60 error = 0.0040402412
i =80 error = 0.003028661
i =100 error = 0.0024201274
```

[28]:
```
@benchmark chrono_gpu!($a, $ap, 0)
```

[28]:
```
BenchmarkTools.Trial: 9 samples with 1 evaluation.
 Range (min … max):
591.229 ms … 593.967 ms    GC
(min … max): 1.84% … 2.16%
 Time  (median):     592.528 ms
  GC (median):    1.99%
 Time  (mean ± ):
592.707 ms ±   1.053 ms    GC
(mean ± ):  2.00% ± 0.19%
```

```
  591 ms           Histogram: frequency by time          594 ms
<

 Memory estimate: 49.84 MiB, allocs estimate:
1617262.
```

[12]:
```
function jacobi_cpu!(ap, a)
    m,n = size(a)
    for i=2:m-1
        for j=2:n-1
            ap[i,j] = 0.2f0 * (a[i-1,j]  + a[i+1,j]   + a[i,j-1]  + a[i,j+1])  +
                      0.05f0 * (a[i-1,j-1]+ a[i+1,j-1] + a[i-1,j+1] +␣
 ↪a[i+1,j+1])
        end
    end
    return
end
```

[12]:
```
jacobi_cpu! (generic function with 1 method)
```

[32]:
```
function chrono_cpu!(b,c, aff)
  init_sol!(b)
  init_sol!(c);
  for i = 1:100
     jacobi_cpu!(c,b)
     error = maximum(abs.(c-b))
     if (aff != 0) && (i % aff) == 0
          println("i =", i, " error = ", error)
     end
     if (error<=1e-3)
       break
     end
     b = copy(c)
  end
end
```

[32]:
```
chrono_cpu! (generic function with 1 method)
```

[33]:
```
b = Array{Float32}(undef, N,N)
c = similar(b)
chrono_cpu!(b, c, 20)
```

```
i =20 error = 0.011931226
i =40 error = 0.0060647726
i =60 error = 0.0040402412
i =80 error = 0.003028661
i =100 error = 0.0024201572
```

[15]:
```
#@benchmark chrono_cpu!($b, $c, 0)
```

9 Mémoire partagée

- On peut essayer d'augmenter la localité des données en utilisant de la mémoire partagée
- `@cuStaticSharedMem` permet d'allouer statiquement de la mémoire partagée
- on synchronise les fils d'exécution grâce à `sync_threads`

[20]:
```
function jacobi_gpu_shared!(a, ap)
    tile = @cuStaticSharedMem(Float32, (BLOCK_X+2, BLOCK_Y+2))
    i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
    j = threadIdx().y + (blockIdx().y - 1) * blockDim().y
    is = threadIdx().x
    js = threadIdx().y
    nx = size(a, 1)
    ny = size(a, 2)

    if ( i > 1 && j > 1)
      tile[is, js] = a[i-1, j-1]
    end
    if ( i > 1 && j < ny && js > BLOCK_Y-2)
      tile[is,  js+2] = a[i-1, j+1]
    end
    if ( j > 1 && i < nx && is > BLOCK_X-2)
      tile[is+2,  js] = a[i+1, j-1]
    end
    if ( i < nx && j < ny && is > BLOCK_X-2 && js > BLOCK_Y - 2)
      tile[is+2,js+2] = a[i+1, j+1]
    end

    sync_threads()

    if (i > 1 && i < nx && j > 1 && j < ny)
      ap[i,j] = 0.2f0  * (tile[is, js+1]  + tile[is+2, js+1]  +
                          tile[is+1, js]  + tile[is+1, js+2]) +
                0.05f0 * (tile[is, js] + tile[is, js+2] +
                          tile[is+2, js] + tile[is+2, js+2])
    end
    return
end
```

[20]:
```
jacobi_gpu_shared! (generic function with 1 method)
```

[29]:
```
function chrono_shared!(a, ap, aff)
  init_sol!(a)
  init_sol!(ap)
  for i = 1:100
     @cuda threads=(BLOCK_X,BLOCK_Y) blocks=(cld(N,BLOCK_X), cld(N, BLOCK_Y))␣
 ↪jacobi_gpu_shared!(a,ap)
     error = reduce(max,abs.(a-ap))
     if (aff != 0) && (i % aff) == 0
          println("i =", i, " error = ",error)
     end
     if (error<=1e-3)
       break
     end
     a = copy(ap)
  end
end
```

[29]:
```
chrono_shared! (generic function with 1 method)
```

[30]:
```
chrono_shared!(a, ap, 20)
```

```
i =20 error = 0.011931241
i =40 error = 0.0060647726
i =60 error = 0.0040402412
i =80 error = 0.003028661
i =100 error = 0.0024201274
```

[31]:
```
@benchmark chrono_shared!($a, $ap, 0)
```

```
BenchmarkTools.Trial: 8 samples with 1 evaluation.
 Range (min … max):
692.533 ms … 694.553 ms    GC
(min … max): 2.00% … 1.97%
 Time  (median):     693.907 ms
 GC (median):    1.88%
 Time  (mean ± ):
693.656 ms ± 704.834  s    GC
(mean ± ):  1.86% ± 0.18%
```

```
  693 ms           Histogram: frequency by time          695 ms
<

 Memory estimate: 58.87 MiB, allocs estimate:
1913226.
```

Malheureusement sur cette exemple on améliore pas le temps d'exécution

10 Pour aller plus loin

10.1 points non abordés

- aborder les réductions (opérations atomiques)
- utiliser les flux

10.2 références

- https://github.com/maleadt/juliacon21-gpu_workshop (code + video)
- CUDA Fortrran for Scientists and Engineers