Preparing code for GPU porting

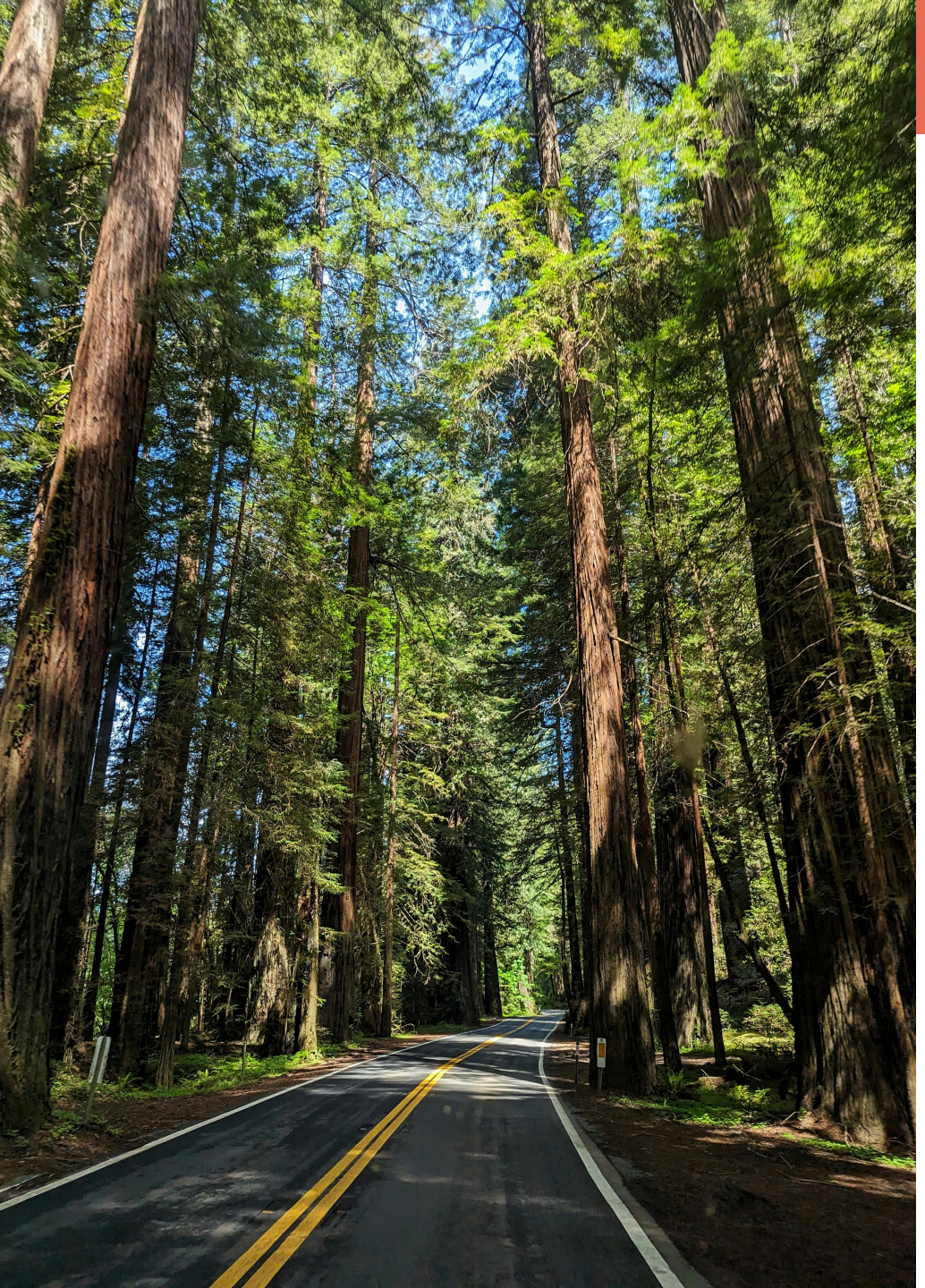


Table of Contents

Why port from CPU to GPU

Advantages of GPU Programming

- Massive parallelism
- Higher memory bandwidth
- Energy efficiency
- Supercomputers are GPU-dominated
- Specialized libraries and ecosystem

Why port from CPU to GPU

Advantages of GPU Programming

- Massive parallelism
- Higher memory bandwidth
- Energy efficiency
- Supercomputers are GPU-dominated
- Specialized libraries and ecosystem

It matters for legacy CPU code

- Most scientific codes are old but extremely valuable
- GPU programming models support incremental (non-invasive) porting
- Port only hotspots, not the entire code
- Easier path to modernization

Steps for porting code from CPU to GPU

- **Identify bottleneck sections**
  - usually 10–20% of code consumes 80–90% of total runtime
  - computationally intensive sections such as loops or matrix operations
- **Find equivalent GPU libraries**
  - if a bottleneck uses a CPU library like FFT, look for the GPU equivalents
  - *i.e.*, `cuFFT` or `hipFFT` can replace CPU-based FFT libraries
- **Refactor loop structures**
  - split loops into smaller parts for independent calculations
- **Memory access optimization**
  - GPUs work best when memory access is aligned and coalesced
  - minimize global memory, maximize shared memory and registers

Code example for computing SOAP descriptor

```
k2 = 0
do i = 1, n_sites
  do j = 1, n_neigh(i)
    k2 = ...
    counter = ...
    do n = 1, n_max
      do np = n, n_max

        do l = 0, l_max
          do m = 0, l
            ...
          end do
        end do

      end do
    end do

    ...

    if( j == 1 )then
    else
      ...
    end if

  end do
end do
```

- outer loop (i) goes over all atoms
- inner loop (j) goes over each atom's neighbors
- loop over n for one radial index from 1 to n_{max}
  - nested np loop from n to n_{max}
- loop over l for angular momentum from 0 to l_{max}
  - nested loop m from 0 to l
- inside these two nested do loops, simple arithmetic calculations are performed
- after looping over all n, np, l, and m, code normalizes computational results with two if branches

Code example for computing SOAP descriptor

```
k2 = 0
do i = 1, n_sites
  do j = 1, n_neigh(i)
    k2 = ...
    counter = ...
    do n = 1, n_max
      do np = n, n_max

        do l = 0, l_max
          do m = 0, l
            ...
          end do
        end do

      end do
    end do

    ...

    if( j == 1 )then
    else
      ...
    end if

  end do
end do
```

Q1: How many kernels do we need for this code?

- split into 2–4 kernels is usually best
- kernel A: inner-combination kernel (heavy work)
- kernel B: normalization & projection
- kernel C: cartesian transform & combination
- (optional) kernel D: pair handling of k2 and k3

Code example for computing SOAP descriptor

```
k2 = 0
do i = 1, n_sites
  do j = 1, n_neigh(i)
    k2 = k2 + 1
    counter = 0
    counter2 = 0
    do n = 1, n_max
      do np = n, n_max

        do l = 0, l_max
          counter=counter+1
          do m = 0, l
            counter2=counter2+1
            ... <in-place>
          end do
        end do

      end do
    end do

    <in-place updated arrays>

    if( j == 1 )then
    else
      ...
    end if

  end do
end do
```

Q1: How many kernels do we need for this code?

- split into 2–4 kernels is usually best
- kernel A: inner-combination kernel (heavy work)
- kernel B: normalization & projection
- kernel C: cartesian transform & combination
- (optional) kernel D: pair handling of k2 and k3

Q2: Any variables that may lead to false dependencies between iterations?

- k2, counter, counter2
- in-place updated arrays

Code example for computing SOAP descriptor

```
k2 = 0
do i = 1, n_sites
  do j = 1, n_neigh(i)
    k2 = ...
    counter = ...
    do n = 1, n_max
      do np = n, n_max

        do l = 0, l_max
          do m = 0, l
            ...
          end do
        end do

      end do
    end do

    ...

    if( j == 1 )then
    else
      ...
    end if

  end do
end do
```

Q3: Is it efficient to split computations over index i?

- yes! splitting over i (atoms) can be efficient
- memory access considerations for Fortran arrays
- strategies to improve memory access
- parallelizing over i is a trade-off
  - good for independence and simplicity
  - suboptimal for natural continuous memory access
- a combined scheme might be optimal
  - parallelize over i (atoms) at block level
  - parallelize over j (neighbors) within a block for memory coalescing

Example: Porting code for GPU acceleration

- Loop A: inner-combination loop over all atom pairs (i,j)

```
do k2 = 1, k2_max
  i=list_of_i(k2)

    do n = 1, n_max
      do np = n, n_max

        do l = 0, l_max
          do m = 0, l
            ...
          end do
        end do

      end do
    end do

  end do
end do
```

Example: Porting code for GPU acceleration

- Loop A: inner-combination loop over all atom pairs (i,j)

```
do k2 = 1, k2_max
  i=list_of_i(k2)

    do n = 1, n_max
      do np = n, n_max

        do l = 0, l_max
          do m = 0, l
            ...
          end do
        end do

      end do
    end do

  end do
end do
```

- Loops B & C: projection & normalization

```
do k2 = 1, k2_max
  i=list_of_i(k2)
  do is = 1,nsoap
    ...
  end do
  ..
end do
```

```
do k2 = 1, k2_max
  i=list_of_i(k2)
  do is = 1, nsoap
    ...
  end do
  ..
end do
```

Example: Porting code for GPU acceleration

- Loop A: inner-combination loop over all atom pairs (i,j)

```
do k2 = 1, k2_max
  i=list_of_i(k2)

    do n = 1, n_max
      do np = n, n_max

        do l = 0, l_max
          do m = 0, l
            ...
          end do
        end do

      end do
    end do

  end do
end do
```

- Loops B & C: projection & normalization

```
do k2 = 1, k2_max
  i=list_of_i(k2)
  do is = 1,nsoap
    ...
  end do
  ..
end do
```

```
do k2 = 1, k2_max
  i=list_of_i(k2)
  do is = 1, nsoap
    ...
  end do
  ..
end do
```

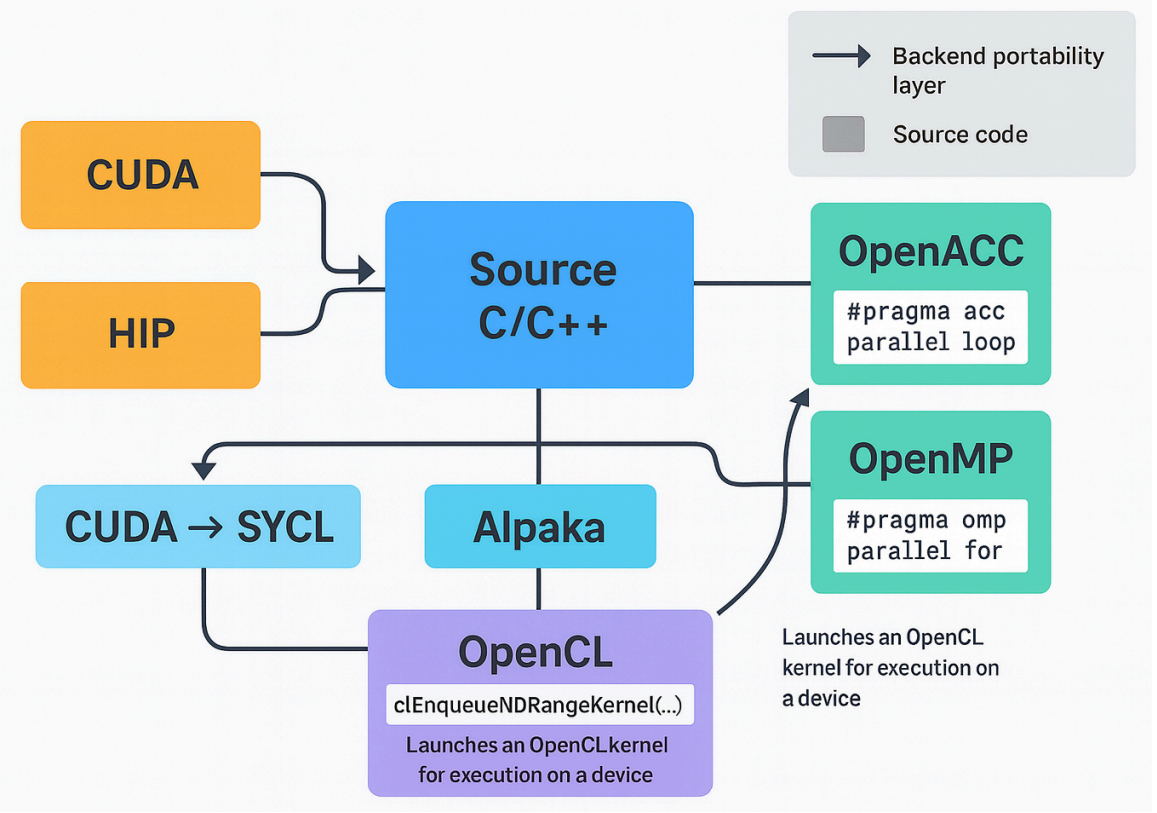
- Loops D & E: transform & combination

```
do k2 = 1, k2_max
  k3=list_k2k3(k2)
  do is = 1, nsoap
    if (k3/=k2)...
  end do
end do
```

```
do k2 = 1, k2_max
  i=list_of_i(k2)
  do is=1, nsoap
    do k2=k3+1, ...
      ...
    end do
  end do
end do
```

Porting Code Between GPU Frameworks

- Hardware diversity in HPC (different GPU vendors)
- Broaden software applications and ensure long-term sustainability
- Reduce the risk of vendor lock-in
- Access to different performance characteristics
- Leverage portability frameworks to simplify maintenance



- Funding agencies and HPC centers require portability

Porting (I)-1: CUDA to HIP with hipify-perl

The **hipify-perl** tool is a perl-based script

- it is designed to facilitate porting of CUDA code to HIP
- it automatically translates CUDA-specific syntax, API calls, and kernel launch constructs into their corresponding HIP equivalents, enabling resulting code to run on AMD GPUs

CUDA → HIP translation is straightforward

- `cudaMalloc` → `hipMalloc`
- `cudaDeviceSynchronize` → `hipDeviceSynchronize`
- `cudaEventCreate()` → `hipEventCreate()`

Porting (I)-1: CUDA to HIP with hipify-perl

- generating `hipify-perl` script

```
module load rocm/6.0.3
hipify-clang --perl
```

- running `hipify-perl` script and covert cuda to hip code

```
hipify-perl program.cu > program.cu.hip
```

- compiling generated HIP code using `hipcc`

```
hipcc --offload-arch=gfx90a -o program.hip.exe \
        program.cu.hip
```

- submitting job to lumi

```
sbatch script.slurm
```

Porting (I)-1: CUDA to HIP with hipify-perl

- generating `hipify-perl` script

```
module load rocm/6.0.3
hipify-clang --perl
```

- running `hipify-perl` script and covert cuda to hip code

```
hipify-perl program.cu > program.cu.hip
```

- compiling generated HIP code using `hipcc`

```
hipcc --offload-arch=gfx90a -o program.hip.exe \
        program.cu.hip
```

Note: `hipify-perl` is suitable for small codebases, while `hipify-clang` is better for large applications

Porting (I)-2: CUDA to HIP with hipify-clang

The **hipify-clang** tool is based on clang for translating CUDA sources into HIP code

- `hipify-clang` requires `LLVM+CLANG` and `CUDA`
- we have pulled a CUDA singularity container, loaded a ROCm module, launched CUDA singularity, and set necessary environment variables
- running `hipify-clang` inside CUDA singularity container on LUMI

```
cd /content/examples/exercise_hipify/Hipify_clang/

./run_hipify-clang   # convert cuda to HIP code

./compile.sh         # compile HIP code using hipcc

sbatch script.slurm  # submit script to run a job
```

Porting (II): OpenACC to OpenMP with Clacc

The **Clacc** is an open-source compiler that brings support for directive-based OpenACC programming model to LLVM/Clang compiler

- with Clacc, developers can write C/C++ code using OpenACC directives (`pragmas`) to specify parallelism for GPUs
- Clacc then compiles code to target accelerators (GPUs/FPGAs(?))

Trade-off between CUDA/HIP and OpenACC + Clacc

- abstraction *vs.* control: OpenACC + Clacc give a higher-level abstraction with less control over threads and memory that raw CUDA provides
- performance *vs.* convenience: fine-tuned CUDA may maximize code performance; OpenACC + Clacc is less optimal, but is still acceptable for many workloads

Porting (II): OpenACC to OpenMP with Clacc

Basic steps to translate OpenACC to OpenMP with Clacc on LUMI

- ~~building and installing Clacc~~
- ~~setting up environment variables for Clacc~~
- conversion of OpenACC into OpenMP code via Clacc

```
clang -fopenacc-structured-ref-count-omp=no-ompx-hold \
        -fopenacc-print=omp openACC_code.c > openMP_code.c
```

- compiling OpenMP code and submitting job to LUMI

```
module load CrayEnv
module load PrgEnv-cray
module load craype-accel-amd-gfx90a
module load rocm/6.0.3
cc -fopenmp -o executable openMP_code.c
```

Take-Home Messages

- Why port code to GPUs
  - benefits of GPU programming
  - leveraging legacy CPU code
- Key steps for code porting
  - identify bottleneck sections, find equivalent GPU libraries
  - refactor loop structures, memory access optimization
  - demonstrated a code example for calculating SOAP descriptor
- **Porting code between GPU frameworks**
  - CUDA to HIP using `hipify-perl` and `hipify-clang`
  - OpenACC to OpenMP with Clacc

GPU Framework vs Project

Which GPU model fits your project?

- Vendor lock-in *vs.* portability
- Programming effort & popular framework
- Performance requirements
- Cost-benefit analysis

Questions & Comments?