INTRODUCTION TO OPENACC

JSC OPENACC COURSE 2024

29 October 2024 | Andreas Herten | Forschungszentrum Jülich

Outline

OpenACC Mission Statement

[…] OpenACC [is] for writing parallel programs in C, C++, and Fortran that run identified regions in parallel on multicore CPUs or attached accelerators.
[…] a model for parallel programming that is portable across operating systems and various types of multicore CPUs and accelerators.
– *OpenACC API Documentation*  , `openacc.org`

OpenACC History

2011 OpenACC 1.0 specification is released at SC11
NVIDIA, Cray, PGI, CAPS

2013 OpenACC 2.0: More functionality, portability

2015 OpenACC 2.5: Enhancements, clarifications

2017 OpenACC 2.6: Deep copy, …

2019 OpenACC 3.0: Newer C++, more lambdas, …

2021 OpenACC 3.2: Error-handling API, `async` extensions, …

2022 OpenACC 3.3: More gang parallelism, Fortran API, …

- Run as a non-profit organization, OpenACC.org
- Members from industry and academia

→ `https://www.openacc.org/` (*see also: Best practice guide*)

OpenACC-enabled Applications

- ANSYS Fluent
- Gaussian
- VASP
- COSMO
- GTC
- SOMA
- …

Open{MP↔ACC}

Everything's connected

- OpenACC modeled after OpenMP …
- … but specific for accelerators
- OpenMP 4.0/4.5: Offloading; compiler support improving (Clang, XL, GCC, …)
- OpenACC more descriptive, OpenMP more prescriptive
- OpenMP 5.0: Descriptive directive `loop`
- Same basic principle: Fork/join model

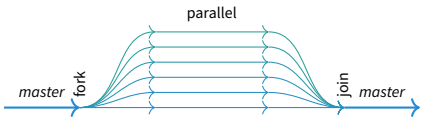
  Master thread launches parallel child threads; merge after execution

Open{MP↔ACC}

Everything's connected

- OpenACC modeled after OpenMP …
- … but specific for accelerators
- OpenMP 4.0/4.5: Offloading; compiler support improving (Clang, XL, GCC, …)
- OpenACC more descriptive, OpenMP more prescriptive
- OpenMP 5.0: Descriptive directive `loop`
- Same basic principle: Fork/join model

  Master thread launches parallel child threads; merge after execution



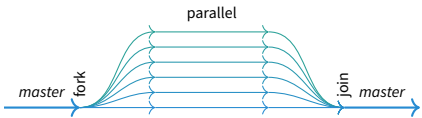
OpenMP

Open{MP↔ACC}

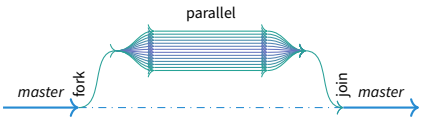
Everything's connected

- OpenACC modeled after OpenMP …
- … but specific for accelerators
- OpenMP 4.0/4.5: Offloading; compiler support improving (Clang, XL, GCC, …)
- OpenACC more descriptive, OpenMP more prescriptive
- OpenMP 5.0: Descriptive directive `loop`
- Same basic principle: Fork/join model

  Master thread launches parallel child threads; merge after execution



OpenMP



OpenACC

JÜLICH Forschungszentrum

OpenACC

Modus Operandi

OpenACC Acceleration Workflow

Three-step program

1. Annotate code with directives, indicating parallelism
2. OpenACC-capable compiler generates accelerator-specific code
3. $success

1 Directives

pragmatic

- Compiler directives state intend to compiler

C/C++

```
#pragma acc kernels
for (int i = 0; i < 23; i++)
// ...
```

Fortran

```
!$acc kernels
do i = 1, 24
! ...
!$acc end kernels
```

- Ignored by compiler which does not understand OpenACC
- OpenACC: Compiler directives, library routines, environment variables
- Portable across host systems and accelerator architectures

2 Compiler

Simple and abstracted

- Trust compiler to generate intended parallelism; always check status output!
- No need to know details of accelerator; leave it to expert compiler engineers *Tuning possible*
- One code can target different accelerators: GPUs, CPUs $\rightarrow$ **Portability**

2 Compiler

Simple and abstracted

- Trust compiler to generate intended parallelism; always check status output!
- No need to know details of accelerator; leave it to expert compiler engineers *Tuning possible*
- One code can target different accelerators: GPUs, CPUs → **Portability**

Compiler	Targets	Languages	OSS	Free	Comment
NVIDIA HPC SDK	NVIDIA GPU, CPU	C, C++, Fortran	No	Yes	Best performance
GCC	NVIDIA GPU, AMD GPU	C, C++, Fortran	Yes	Yes	
HPE Cray	NVIDIA GPU	Fortran	No	No	*???*
Clang/LLVM	CPU, NVIDIA GPU	C, *C++*. *Fortran*	Yes	Yes	Via Clang OpenMP backend

2 Compiler

Simple and abstracted

- Trust compiler to generate intended parallelism; always check status output!
- No need to know details of accelerator; leave it to expert compiler engineers *Tuning possible*
- One code can target different accelerators: GPUs, CPUs → **Portability**

Compiler	Targets	Languages	OSS	Free	Comment
NVHPC	NVIDIA GPU, CPU	C, C++, Fortran	No	Yes	Best performance
GCC	NVIDIA GPU, AMD GPU	C, C++, Fortran	Yes	Yes	
HPE Cray	NVIDIA GPU	Fortran	No	No	*???*
Clang/LLVM	CPU, NVIDIA GPU	C, *C++*. *Fortran*	Yes	Yes	Via Clang OpenMP backend

2 Compiler

Flags and options

OpenACC compiler support: activate with compile flag

NVHPC `nvc -acc`

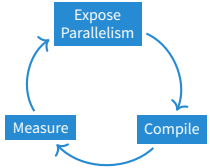
`-acc=gpu|multicore`	Target GPU or CPU
`-acc=gpu -gpu=cc80`	Generate Ampere-compatible code
`-gpu=cc80,lineinfo`	Add source code correlation into binary
`-gpu=managed`	Use unified memory
`-Minfo=accel`	Print acceleration info

GCC `gcc -fopenacc`

`-fopenacc-dim=geom`	Use *geom* configuration for threads
`-foffload="-lm -O3"`	Provide flags to offload compiler
`-fopt-info-omp`	Print acceleration info

3 $success

Iteration is key



- Serial to parallel: fast
- Serial to fast parallel: more time needed
- Start simple $\rightarrow$ refine
- Expose more and more parallelism

$\Rightarrow$ **Productivity**

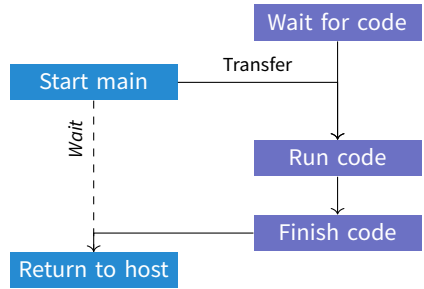
- Because of *generality*: Sometimes not last bit of hardware performance accessible
- But: Use OpenACC together with other accelerator-targeting techniques (CUDA, libraries, …)

OpenACC Accelerator Model

For computation and memory spaces

- Main program executes on host
- Device code is transferred to accelerator
- Execution on accelerator is started
- Host waits until return (except: async)

- Two separate memory spaces; data transfers back and forth
  - Transfers hidden from programmer
  - Memories not coherent!
  - Compiler helps; GPU runtime helps



JÜLICH Forschungszentrum

A Glimpse of OpenACC

```
#pragma acc data copy(x[0:N],y[0:N])
#pragma acc parallel loop
{
    for (int i=0; i<N; i++) {
        x[i] = 1.0;
        y[i] = 2.0;
    }
    for (int i=0; i<N; i++) {
        y[i] = i*x[i]+y[i];
    }
}
```

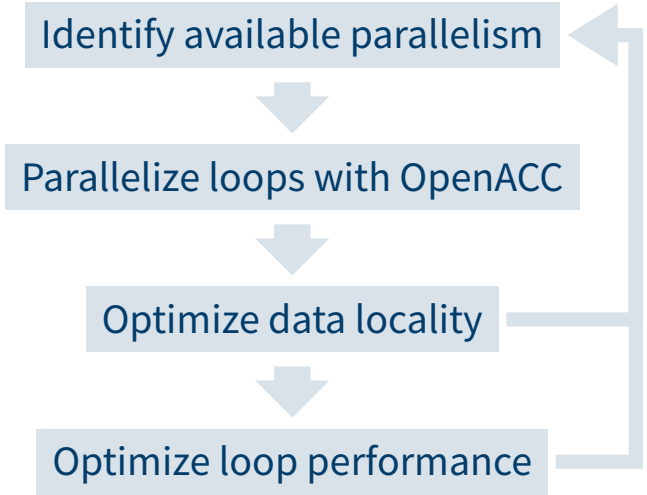
```
!$acc data copy(x(1:N),y(1:N))
!$acc parallel loop

    do i = 1, N
        x(i) = 1.0
        y(i) = 2.0
    end do
    do i = 1, N
        y(i) = i*x(i)+y(i);
    end do

!$acc end parallel loop
!$acc end data
```

OpenACC by Example

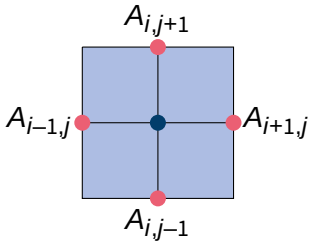
Parallelization Workflow

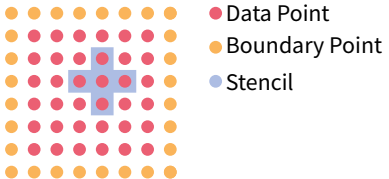


Jacobi Solver

Algorithmic description

- Example for acceleration: **Jacobi solver**
- Iterative solver, converges to correct value
- Each iteration step: compute average of neighboring points
- Example: 2D Poisson equation: $\nabla^2 A(x, y) = B(x, y)$





$$A_{k+1}(i,j) = -\frac{1}{4}\left(B(i,j) - (A_k(i-1,j) + A_k(i,j+1), + A_k(i+1,j) + A_k(i,j-1))\right)$$

Jacobi Solver

Source code

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    for (int ix = ix_start; ix < ix_end; ix++) {
        for (int iy = iy_start; iy < iy_end; iy++) {
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -
                (  A[iy*nx+ix+1] + A[iy*nx+ix-1]
                 + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));
            error = fmaxr(error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
    }}
    for (int iy = iy_start; iy < iy_end; iy++) {
        for( int ix = ix_start; ix < ix_end; ix++ ) {
            A[iy*nx+ix] = Anew[iy*nx+ix];
    }}
    for (int ix = ix_start; ix < ix_end; ix++) {
            A[0*nx+ix]      = A[(ny-2)*nx+ix];
            A[(ny-1)*nx+ix] = A[1*nx+ix];
    }
    // same for iy
    iter++;
}
```

Jacobi Solver

Source code

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    for (int ix = ix_start; ix < ix_end; ix++) {
        for (int iy = iy_start; iy < iy_end; iy++) {
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -
                (  A[iy*nx+ix+1] + A[iy*nx+ix-1]
                 + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));
            error = fmaxr(error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
    }}
    for (int iy = iy_start; iy < iy_end; iy++) {
        for( int ix = ix_start; ix < ix_end; ix++ ) {
            A[iy*nx+ix] = Anew[iy*nx+ix];
    }}
    for (int ix = ix_start; ix < ix_end; ix++) {
            A[0*nx+ix]      = A[(ny-2)*nx+ix];
            A[(ny-1)*nx+ix] = A[1*nx+ix];
    }
    // same for iy
    iter++;
}
```

Iterate until converged

Jacobi Solver

Source code

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    for (int ix = ix_start; ix < ix_end; ix++) {
        for (int iy = iy_start; iy < iy_end; iy++) {
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -
                (  A[iy*nx+ix+1] + A[iy*nx+ix-1]
                 + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));
            error = fmaxr(error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
    }}
    for (int iy = iy_start; iy < iy_end; iy++) {
        for( int ix = ix_start; ix < ix_end; ix++ ) {
            A[iy*nx+ix] = Anew[iy*nx+ix];
    }}
    for (int ix = ix_start; ix < ix_end; ix++) {
            A[0*nx+ix]      = A[(ny-2)*nx+ix];
            A[(ny-1)*nx+ix] = A[1*nx+ix];
    }
    // same for iy
    iter++;
}
```

Iterate until converged

Iterate across matrix elements

Jacobi Solver

Source code

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    for (int ix = ix_start; ix < ix_end; ix++) {
        for (int iy = iy_start; iy < iy_end; iy++) {
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -
                (  A[iy*nx+ix+1] + A[iy*nx+ix-1]
                 + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));
            error = fmaxr(error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
    }}
    for (int iy = iy_start; iy < iy_end; iy++) {
        for( int ix = ix_start; ix < ix_end; ix++ ) {
            A[iy*nx+ix] = Anew[iy*nx+ix];
    }}
    for (int ix = ix_start; ix < ix_end; ix++) {
            A[0*nx+ix]      = A[(ny-2)*nx+ix];
            A[(ny-1)*nx+ix] = A[1*nx+ix];
    }
    // same for iy
    iter++;
}
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Jacobi Solver

Source code

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    for (int ix = ix_start; ix < ix_end; ix++) {
        for (int iy = iy_start; iy < iy_end; iy++) {
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -
                (  A[iy*nx+ix+1] + A[iy*nx+ix-1]
                 + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));
            error = fmaxr(error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
    }}
    for (int iy = iy_start; iy < iy_end; iy++) {
        for( int ix = ix_start; ix < ix_end; ix++ ) {
            A[iy*nx+ix] = Anew[iy*nx+ix];
    }}
    for (int ix = ix_start; ix < ix_end; ix++) {
            A[0*nx+ix]      = A[(ny-2)*nx+ix];
            A[(ny-1)*nx+ix] = A[1*nx+ix];
    }
    // same for iy
    iter++;
}
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Accumulate error

Jacobi Solver

Source code

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    for (int ix = ix_start; ix < ix_end; ix++) {
        for (int iy = iy_start; iy < iy_end; iy++) {
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -
                (  A[iy*nx+ix+1] + A[iy*nx+ix-1]
                 + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));
            error = fmaxr(error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
    }}
    for (int iy = iy_start; iy < iy_end; iy++) {
        for( int ix = ix_start; ix < ix_end; ix++ ) {
            A[iy*nx+ix] = Anew[iy*nx+ix];
    }}
    for (int ix = ix_start; ix < ix_end; ix++) {
            A[0*nx+ix]      = A[(ny-2)*nx+ix];
            A[(ny-1)*nx+ix] = A[1*nx+ix];
    }
    // same for iy
    iter++;
}
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Accumulate error

Swap input/output

Jacobi Solver

Source code

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    for (int ix = ix_start; ix < ix_end; ix++) {
        for (int iy = iy_start; iy < iy_end; iy++) {
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -
                (  A[iy*nx+ix+1] + A[iy*nx+ix-1]
                 + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));
            error = fmaxr(error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
    }}
    for (int iy = iy_start; iy < iy_end; iy++) {
        for( int ix = ix_start; ix < ix_end; ix++ ) {
            A[iy*nx+ix] = Anew[iy*nx+ix];
    }}
    for (int ix = ix_start; ix < ix_end; ix++) {
            A[0*nx+ix]      = A[(ny-2)*nx+ix];
            A[(ny-1)*nx+ix] = A[1*nx+ix];
    }
    // same for iy
    iter++;
}
```

Iterate until converged

Iterate across matrix elements

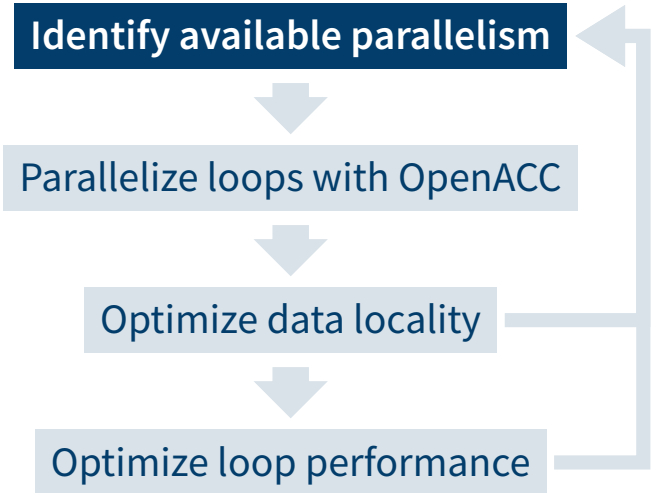
Calculate new value from neighbors

Accumulate error

Swap input/output

Set boundary conditions

Parallelization Workflow



Profiling

*Pro*file

> *[…] premature optimization is the root of all evil.*
>
> *– Donald Knuth [3]*

- Investigate hot spots of your program!

→ Profile!

- Many tools, many levels: perf, PAPI, Score-P, Intel Advisor, NVIDIA profilers, …
- Here: Examples from GCC

Profiling

*Pro*file

> *[…] premature optimization is the root of all evil.*
> ***Yet we should not pass up our [optimization] opportunities […]***
> *– Donald Knuth [3]*

- Investigate hot spots of your program!

→ Profile!

- Many tools, many levels: perf, PAPI, Score-P, Intel Advisor, NVIDIA profilers, …
- Here: Examples from GCC

Identify Parallelism

TASK 1

Generate Profile

- Use `gprof` to analyze unaccelerated version of Jacobi solver
- Investigate!

Task 1: Analyze Application

- Re-load NVHPC compiler with `module load NVHPC`
- Change to `Task1/` directory
- Compile: `make task1`
  Usually, compile just with `make` (but this exercise is special)
- Submit *profiling run* to the batch system: `make task1_profile`
  Study `srun` call and `gprof` call; try to understand

Identify Parallelism

TASK 1

Generate Profile

- Use `gprof` to analyze unaccelerated version of Jacobi solver
- Investigate!

Task 1: Analyze Application

- Re-load NVHPC compiler with `module load NVHPC`
- Change to `Task1/` directory
- Compile: `make task1`
  Usually, compile just with `make` (but this exercise is special)
- Submit *profiling run* to the batch system: `make task1_profile`
  Study `srun` call and `gprof` call; try to understand

??? Where is hotspot? Which parts should be accelerated?

Profile of Application

```
$ gcc -g -pg -DUSE_DOUBLE -c -o poisson2d_reference.o poisson2d_reference.c
$ gcc -g -pg -DUSE_DOUBLE -lm poisson2d_reference.o poisson2d.c -o poisson2d
$ gprof -p -l ./poisson2d gmon.out
  Flat profile:
  Each sample counts as 0.01 seconds.
    %   cumulative   self              self     total
   time   seconds   seconds    calls  Ts/call  Ts/call  name
   46.29      1.28     1.28                             main (poisson2d.c:107 @ 40135c)
   30.01      2.11     0.83                             main (poisson2d.c:108 @ 4013cd)
   12.66      2.46     0.35                             main (poisson2d.c:109 @ 401458)
    6.15      2.63     0.17                             main (poisson2d.c:107 @ 401421)
```

- Very simple here: All in `main`
- Lines 107, 108, 109: within the inner grid loop
- Good position to start! Let's study this further in *independency analysis*

Code Independency Analysis

Independence is key

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    for (int ix = ix_start; ix < ix_end; ix++) {
        for (int iy = iy_start; iy < iy_end; iy++) {
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -
                (  A[iy*nx+ix+1] + A[iy*nx+ix-1]
                 + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));
            error = fmaxr(error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
    }}
    for (int iy = iy_start; iy < iy_end; iy++) {
        for( int ix = ix_start; ix < ix_end; ix++ ) {
            A[iy*nx+ix] = Anew[iy*nx+ix];
    }}
    for (int ix = ix_start; ix < ix_end; ix++) {
            A[0*nx+ix]      = A[(ny-2)*nx+ix];
            A[(ny-1)*nx+ix] = A[1*nx+ix];
    }
    // same for iy
    iter++;
}
```

Code Independency Analysis

Independence is key

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    for (int ix = ix_start; ix < ix_end; ix++) {
        for (int iy = iy_start; iy < iy_end; iy++) {
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -
                (  A[iy*nx+ix+1] + A[iy*nx+ix-1]
                 + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));
            error = fmaxr(error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
    }}
    for (int iy = iy_start; iy < iy_end; iy++) {
        for( int ix = ix_start; ix < ix_end; ix++ ) {
            A[iy*nx+ix] = Anew[iy*nx+ix];
    }}
    for (int ix = ix_start; ix < ix_end; ix++) {
            A[0*nx+ix]      = A[(ny-2)*nx+ix];
            A[(ny-1)*nx+ix] = A[1*nx+ix];
    }
    // same for iy
    iter++;
}
```

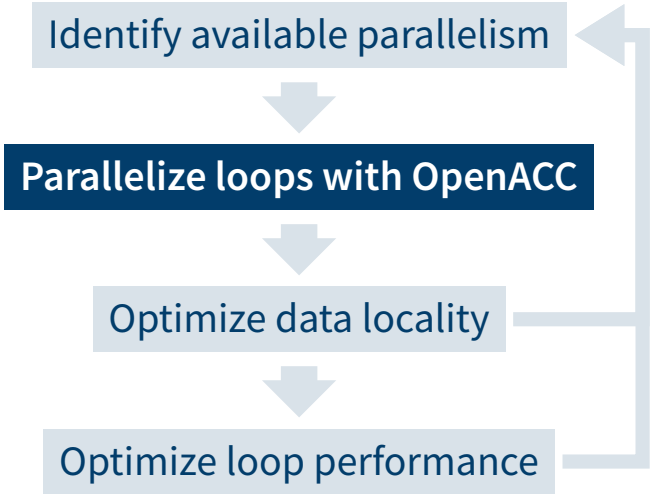
Data dependency between iterations

Independent loop iterations

Independent loop iterations

Independent loop iterations

Parallelization Workflow



Parallel Loops: Parallel

An important directive

- Programmer identifies block containing parallelism
  $\rightarrow$ compiler generates offload code
- Program launch creates *gangs* of parallel threads on parallel device
- Implicit barrier at end of parallel region
- Each gang executes same code sequentially

OpenACC: `parallel` C

```
#pragma acc parallel [clause, [, clause] ...] newline
{structured block}
```

Parallel Loops: Parallel

An important directive

- Programmer identifies block containing parallelism
  $\rightarrow$ compiler generates offload code
- Program launch creates *gangs* of parallel threads on parallel device
- Implicit barrier at end of parallel region
- Each gang executes same code sequentially

OpenACC: `parallel` F

```
!$acc parallel [clause, [, clause] ...]
!$acc end parallel
```

Parallel Loops: Parallel

An important directive

- Programmer identifies block containing parallelism
  $\rightarrow$ compiler generates offload code
- Program launch creates *gangs* of parallel threads on parallel device
- Implicit barrier at end of parallel region
- Each gang executes same code sequentially

OpenACC: `parallel` F

```
!$acc parallel [clause, [, clause] ...]
!$acc end parallel
```

Parallel Loops: Parallel

Clauses

Diverse clauses to augment the parallel region

`private(var)`	A copy of variables `var` is made for each gang
`firstprivate(var)`	Same as `private`, except `var` will initialized with value from host
`if(cond)`	Parallel region will execute on accelerator only if `cond` is true
`reduction(op:var)`	Reduction is performed on variable `var` with operation `op`; supported: `+ * max min` …
`async[(int)]`	No implicit barrier at end of parallel region

Parallel Loops: Loops

Also an important directive

- Programmer identifies loop eligible for parallelization
- Directive must be directly before loop
- Optional: Describe type of parallelism

OpenACC: `loop` C

```
#pragma acc loop [clause, [, clause] ...] newline
{structured block}
```

Parallel Loops: Loops

Also an important directive

- Programmer identifies loop eligible for parallelization
- Directive must be directly before loop
- Optional: Describe type of parallelism

OpenACC: `loop`

F

```
!$acc loop [clause, [, clause] ...]
!$acc end loop
```

Parallel Loops: Loops

Clauses

`independent`	Iterations of loop are data-independent (implied if in `parallel` region (and no `seq` or `auto`))
`collapse(int)`	Collapse `int` tightly-nested loops
`seq`	This loop is to be executed sequentially (not parallel)
`tile(int[,int])`	Split loops into loops over tiles of the full size
`auto`	Compiler decides what to do

Parallel Loops: Parallel Loops

Maybe the most important directive

- Combined directive: shortcut
  Because its used so often
- Any clause that is allowed on `parallel` or `loop` allowed
- Restriction: May not appear in body of another parallel region

OpenACC: `parallel loop` C

```
#pragma acc parallel loop [clause, [, clause] ...] newline
{structured block}
```

Parallel Loops: Parallel Loops

Maybe the most important directive

- Combined directive: shortcut
  Because its used so often
- Any clause that is allowed on `parallel` or `loop` allowed
- Restriction: May not appear in body of another parallel region

OpenACC: `parallel loop` F

```
!$acc parallel loop [clause, [, clause] ...]
!$acc end parallel loop
```

Parallel Loops: Parallel Loops

Maybe the most important directive

- Combined directive: shortcut
  Because its used so often
- Any clause that is allowed on `parallel` or `loop` allowed
- Restriction: May not appear in body of another parallel region

OpenACC: `parallel loop`

```
#pragma acc parallel loop [clause, [, clause] ...]
```

Parallel Loops Example

```c
double sum = 0.0;
#pragma acc parallel loop
for (int i=0; i<N; i++) {
    x[i] = 1.0;
    y[i] = 2.0;
}

#pragma acc parallel loop reduction(+:sum)
for (int i=0; i<N; i++) {
    y[i] = i*x[i]+y[i];
    sum+=y[i];
}
```

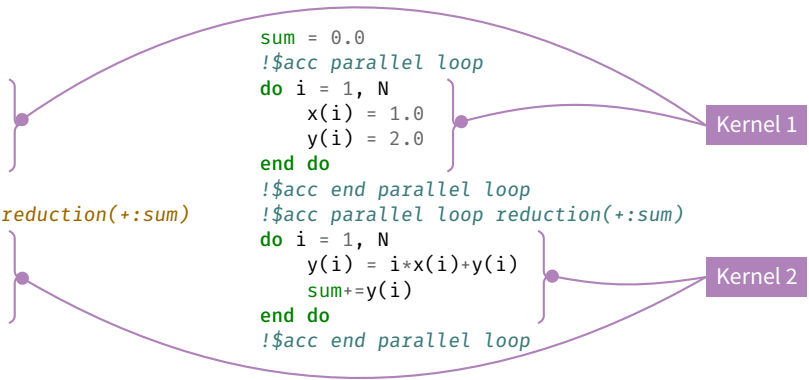
```fortran
sum = 0.0
!$acc parallel loop
do i = 1, N
    x(i) = 1.0
    y(i) = 2.0
end do
!$acc end parallel loop
!$acc parallel loop reduction(+:sum)
do i = 1, N
    y(i) = i*x(i)+y(i)
    sum+=y(i)
end do
!$acc end parallel loop
```

Parallel Loops Example

```c
double sum = 0.0;
#pragma acc parallel loop
for (int i=0; i<N; i++) {
    x[i] = 1.0;
    y[i] = 2.0;
}

#pragma acc parallel loop reduction(+:sum)
for (int i=0; i<N; i++) {
    y[i] = i*x[i]+y[i];
    sum+=y[i];
}
```

```
sum = 0.0
!$acc parallel loop
do i = 1, N
    x(i) = 1.0
    y(i) = 2.0
end do
!$acc end parallel loop
!$acc parallel loop reduction(+:sum)
do i = 1, N
    y(i) = i*x(i)+y(i)
    sum+=y(i)
end do
!$acc end parallel loop
```



JÜLICH Forschungszentrum

Parallel Jacobi

Add parallelism

TASK 2

- Add OpenACC parallelism to main double loop in Jacobi solver source code

→ Congratulations, you are a GPU developer!

Task 2: A First Parallel Loop

- Change to `Task2/` directory
- Compile: `make`
- Submit parallel run to the batch system: `make run`

  Adapt the `srun` call and run with other number of iterations, matrix sizes

Fortran

- All tasks available in Fortran: `exercises/Fortran/Task2/`
- Fortran *much* faster than C
- Slides follow C results
- Fortran: No command line options parsed

Parallel Jacobi

Source Code

```
110  #pragma acc parallel loop reduction(max:error)
111  for (int ix = ix_start; ix < ix_end; ix++)
112  {
113    for (int iy = iy_start; iy < iy_end; iy++)
114      {
115        Anew[iy*nx+ix] = -0.25 * ( rhs[iy*nx+ix]   -
116                                   ( A[iy*nx+ix+1]   + A[iy*nx+ix-1]
117                                   + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix] ));
118          error = fmaxr( error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
119      }
120  }
```

Parallel Jacobi

Compilation result

```
$ make
nvc -DUSE_DOUBLE -Minfo=accel -O1 -acc=gpu -gpu=managed poisson2d.c poisson2d_reference.o
 -o poisson2d
 poisson2d_reference.o -o poisson2d
poisson2d.c:
main:
    106, Generating Tesla code
        110, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(max:error)
        112, #pragma acc loop seq
    106, Generating implicit copyin(A[:]) [if not already present]
         Generating implicit copy(error) [if not already present]
         Generating implicit copyin(rhs[:]) [if not already present]
    112, Complex loop carried dependence of Anew-> prevents parallelization
         Loop carried dependence of Anew-> prevents parallelization
```

Parallel Jacobi

Run result

```
$ make run
srun --gres=gpu:1 --time 0:10:00 --pty ./poisson2d
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution.
    0, 0.249999
  100, 0.249760
  200, 0.…
Calculate current execution.
    0, 0.249999
  100, 0.249760
  200, 0.…
2048x2048: Ref: 105.6753 s, This:  14.0692 s, speedup:     7.51
```

Nsight Systems

NVIDIA's Application Profiler

- Profiler for GPU applications
- CLI and GUI (timeline view)
- Sister tool: Nsight Compute (kernel profiler)
- More: tomorrow in dedicated session

Profile of Jacobi

With nsys

```
$ make profile
srun --gres=gpu:1 --time 0:10:00 --pty nsys nvprof ./poisson2d 10

CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls    Average      Minimum     Maximum            Name
 -------  ---------------  ---------  ------------  ----------  ----------  --------------------
    90.9      160,407,572         30   5,346,919.1       1,780  25,648,117  cuStreamSynchronize

CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances    Average      Minimum     Maximum          Name
 -------  ---------------  ---------  ------------  ----------  ----------  -----------------
   100.0      158,686,617         10  15,868,661.7  14,525,819  25,652,783  main_106_gpu
     0.0           25,120         10       2,512.0       2,304       3,680  main_106_gpu__red
```

Profile of Jacobi

With nsys

```
$ make profile
srun --gres=gpu:1 --time 0:10:00 --pty nsys nvprof (

CUDA API Statistics:

 Time(%)  Total Time
 -------  ----------
    90.9      160,407,

CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances    Average      Minimum     Maximum          Name
 -------  ---------------  ---------  ------------  ----------  ----------  ----------------
   100.0      158,686,617         10  15,868,661.7  14,525,819  25,652,783  main_106_gpu
     0.0           25,120         10       2,512.0       2,304       3,680  main_106_gpu__red
```

Only one function is parallelized!
Let's do the rest!

More Parallelism: Kernels

More freedom for compiler

- Kernels directive: second way to expose parallelism
- Region may contain parallelism
- Compiler determines parallelization opportunities

→ More freedom for compiler

- Rest: Same as for `parallel`

OpenACC: `kernels`

```
#pragma acc kernels [clause, [, clause] ...]
```

Kernels Example

```
double sum = 0.0;
#pragma acc kernels
{
for (int i=0; i<N; i++) {
    x[i] = 1.0;
    y[i] = 2.0;
}
for (int i=0; i<N; i++) {
    y[i] = i*x[i]+y[i];
    sum+=y[i];
}
}
```

`kernels` vs. `parallel`

- Both approaches equally valid; can perform equally well

`kernels` vs. `parallel`

- Both approaches equally valid; can perform equally well
- `kernels`
  - Compiler performs parallel analysis
  - Can cover large area of code with single directive
  - Gives compiler additional leeway
- `parallel`
  - Requires parallel analysis by programmer
  - Will also parallelize what compiler may miss
  - More explicit
  - Similar to OpenMP

`kernels` vs. `parallel`

- Both approaches equally valid; can perform equally well
- `kernels`
  - Compiler performs parallel analysis
  - Can cover large area of code with single directive
  - Gives compiler additional leeway
- `parallel`
  - Requires parallel analysis by programmer
  - Will also parallelize what compiler may miss
  - More explicit
  - Similar to OpenMP
- Both regions may not contain other `kernels`/`parallel` regions
- No branching into or out
- Program must not depend on order of evaluation of clauses
- At most: One `if` clause

Parallel Jacobi II

TASK 3

Add more parallelism

- Add OpenACC parallelism to other loops of `while` (L:123 – L:141)
- Use either `kernels` or `parallel`
- Do they perform equally well?

Task 3: More Parallel Loops

- Change to `Task3/` directory
- Compile: `make`
  Study the compiler output!
- Submit parallel run to the batch system: `make run`

? What's your speed-up?

Parallel Jacobi

Source Code

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    #pragma acc parallel loop reduction(max:error)
    for (int ix = ix_start; ix < ix_end; ix++) {
        for (int iy = iy_start; iy < iy_end; iy++) {
            Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] -
                (   A[iy*nx+ix+1] + A[iy*nx+ix-1]
               + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix]));
            error = fmaxr(error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
    }}
    #pragma acc parallel loop
    for (int iy = iy_start; iy < iy_end; iy++) {
        for( int ix = ix_start; ix < ix_end; ix++ ) {
            A[iy*nx+ix] = Anew[iy*nx+ix];
    }}
    #pragma acc parallel loop
    for (int ix = ix_start; ix < ix_end; ix++) {
            A[0*nx+ix]      = A[(ny-2)*nx+ix];
            A[(ny-1)*nx+ix] = A[1*nx+ix];
    }
    // same for iy
    iter++;
}
```

Parallel Jacobi II

Compilation result

```
$ make
nvc -c -DUSE_DOUBLE -Minfo=accel -O1 -acc=gpu -gpu=managed poisson2d_reference.c -o poisson2d_reference.o
nvc -DUSE_DOUBLE -Minfo=accel -O1 -acc=gpu -gpu=managed poisson2d.c poisson2d_reference.o -o poisson2d
poisson2d.c:
main:
    106, Generating Tesla code
        110, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(max:error)
        112, #pragma acc loop seq
    106, ...
    118, Generating Tesla code
        123, #pragma acc loop gang /* blockIdx.x */
        125, #pragma acc loop vector(128) /* threadIdx.x */
    118, Generating implicit copyin(Anew[:]) [if not already present]
         Generating implicit copyout(A[:]) [if not already present]
    125, Loop is paral…
```

Parallel Jacobi II

Run result

```
$ make run
srun --gres=gpu:1 --time 0:10:00 --pty ./poisson2d
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution.
    0, 0.249999
  100, 0.249760
  200, 0.…
Calculate current execution.
    0, 0.249999
  100, 0.249760
  200, 0.…
2048x2048: Ref: 105.4636 s, This:   0.3448 s, speedup:   305.86
```

Parallel Jacobi II

Run result

```
$ make run
srun --gres=gpu:1 --time 0:10:00 --pty ./poisson2d
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial execution.
    0, 0.249999
  100, 0.249760
  200, 0.…
Calculate current execution.
    0, 0.249999
  100, 0.249760
  200, 0.…
2048x2048: Ref: 105.4636 s, This:   0.3448 s, speedup:   305.86
```

Done?!

OpenACC by Example

Data Transfers

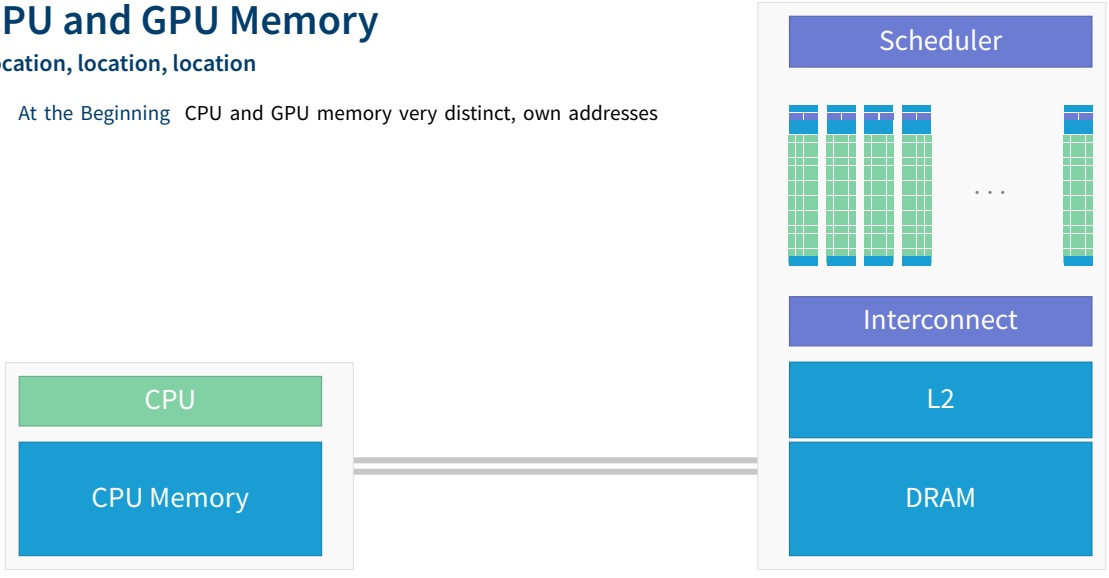
Automatic Data Transfers

- Up to now: We did not care about **data transfers**
- Compiler and runtime care
- Magic keyword: `-gpu=managed`
- Only feature of (recent) NVIDIA GPUs!

CPU and GPU Memory

Location, location, location

At the Beginning CPU and GPU memory very distinct, own addresses

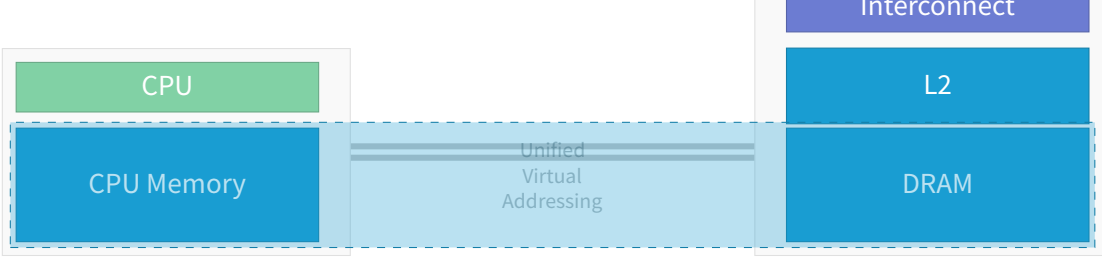


CPU and GPU Memory

Location, location, location

At the Beginning CPU and GPU memory very distinct, own addresses

CUDA 4.0 Unified Virtual Addressing: pointer from same address pool, but data copy manual



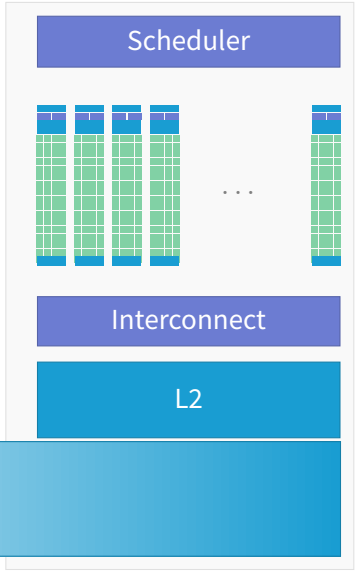
CPU and GPU Memory

Location, location, location

At the Beginning CPU and GPU memory very distinct, own addresses

CUDA 4.0 Unified Virtual Addressing: pointer from same address pool, but data copy manual

CUDA 6.0 Unified Memory*: Data copy by driver, but whole data at once



CPU and GPU Memory

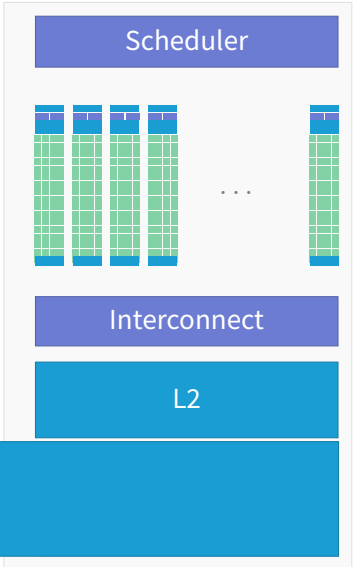
Location, location, location

At the Beginning CPU and GPU memory very distinct, own addresses

CUDA 4.0 Unified Virtual Addressing: pointer from same address pool, but data copy manual

CUDA 6.0 Unified Memory*: Data copy by driver, but whole data at once

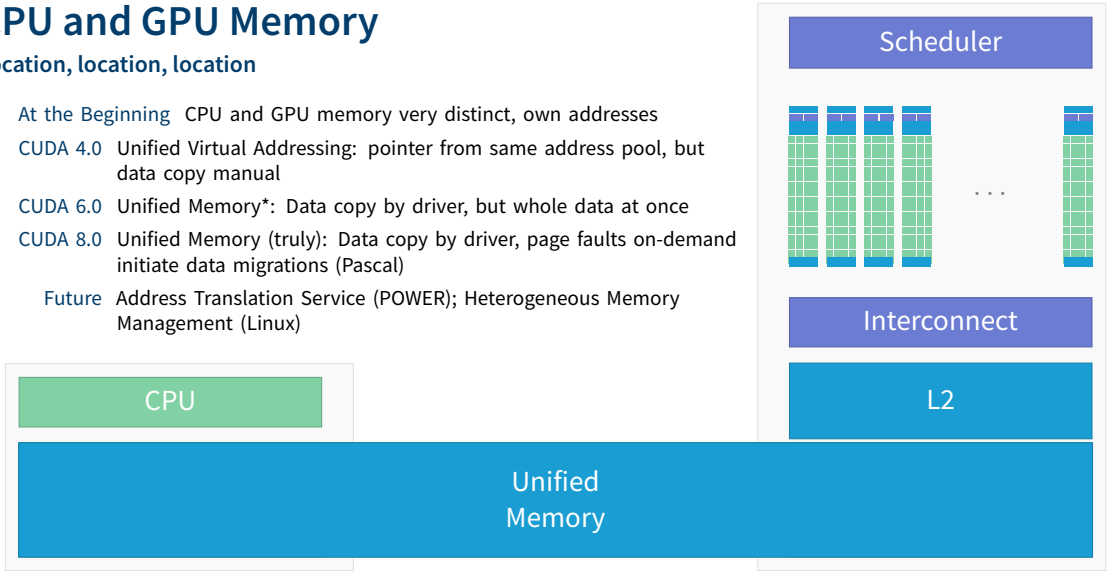
CUDA 8.0 Unified Memory (truly): Data copy by driver, page faults on-demand initiate data migrations (Pascal)



CPU and GPU Memory

Location, location, location

At the Beginning	CPU and GPU memory very distinct, own addresses
CUDA 4.0	Unified Virtual Addressing: pointer from same address pool, but data copy manual
CUDA 6.0	Unified Memory*: Data copy by driver, but whole data at once
CUDA 8.0	Unified Memory (truly): Data copy by driver, page faults on-demand initiate data migrations (Pascal)
Future	Address Translation Service (POWER); Heterogeneous Memory Management (Linux)



Portability

- Managed memory: Very productive feature
- Manual transfers: Fine-grained control, possibly faster, **portability**

→ Code should also be fast without `-gpu=managed`!

- Let's remove it from compile flags!

Portability

Only C

- Managed memory: Very productive feature
- Manual transfers: Fine-grained control, possibly faster, **portability**

→ Code should also be fast without `-gpu=managed`!

- Let's remove it from compile flags!

```
$ make
nvc -c -DUSE_DOUBLE -Minfo=accel -O1 -acc=gpu poisson2d_reference.c -o
  poisson2d_reference.o
nvc -DUSE_DOUBLE -Minfo=accel -O1 -acc=gpu poisson2d.c poisson2d_reference.o -o poisson2d
poisson2d.c:
NVC++-S-0155-Compiler failed to translate accelerator region (see -Minfo messages): Could
  not find allocated-variable index for symbol - rhs (poisson2d.c: 106)
...
NVC++-F-0704-Compilation aborted due to previous errors. (poisson2d.c)
NVC++/x86-64 Linux 21.9-0: compilation aborted
```

Copy Statements

- Compiler implicitly created `copy` clauses to copy data to device

```
106, Generating implicit copyin(A[:],rhs[:]) [if not already present]
     Generating implicit copy(error) [if not already present]
```

- It couldn't determine length of copied data …but before: no problem – Unified Memory!

Copy Statements

- Compiler implicitly created `copy` clauses to copy data to device

```
106, Generating implicit copyin(A[:],rhs[:]) [if not already present]
     Generating implicit copy(error) [if not already present]
```

- It couldn't determine length of copied data …but before: no problem – Unified Memory!
- Now: Problem! We need to give that information!
  (Fortran: can often be determined by compiler)

Copy Statements

- Compiler implicitly created `copy` clauses to copy data to device

```
106, Generating implicit copyin(A[:],rhs[:]) [if not already present]
     Generating implicit copy(error) [if not already present]
```

- It couldn't determine length of copied data …but before: no problem – Unified Memory!
- Now: Problem! We need to give that information!
  (Fortran: can often be determined by compiler)

OpenACC: `copy` C

```
#pragma acc parallel copy(A[start:length])
```

Also: `copyin(B[s:l])` `copyout(C[s:l])` `present(D[s:l])` `create(E[s:l])`

Copy Statements

- Compiler implicitly created `copy` clauses to copy data to device

```
106, Generating implicit copyin(A[:],rhs[:]) [if not already present]
     Generating implicit copy(error) [if not already present]
```

- It couldn't determine length of copied data …but before: no problem – Unified Memory!
- Now: Problem! We need to give that information!
  (Fortran: can often be determined by compiler)

OpenACC: `copy` F

`#pragma acc parallel copy(A(low:high))`

Also: `copyin(B(l:h)` `copyout(C(l:h)` `present(D(l:h)` `create(E(l:h)`

Data Copies

Get that data!

- Add `copy` clause to parallel regions

Task 4: Data Copies

- Change to `Task4/` directory
- Work on `TODO`s
- Compile: `make`
- Submit parallel run to the batch system: `make run`

? What's your speed-up?

Data Copies

Compiler Output

```
$ make
nvc -DUSE_DOUBLE -Minfo=accel -O1 -acc=gpu poisson2d.c poisson2d_reference.o -o poisson2d
poisson2d.c:
main:
    106, Generating copy(A[:ny*nx],rhs[:ny*nx]) [if not already present]
         Generating implicit copy(error) [if not already present]
         Generating Tesla code
        110, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(max:error)
        112, #pragma acc loop seq
    106, Generating copy(Anew[:ny*nx]) [if not already present]
    112, Complex loop carried dependence of Anew-> prevents parallelization
         Loop carried dependence of Anew-> prevents parallelization
         Loop carried backward dependence of Anew-> prevents vectorization
```

Data Copies

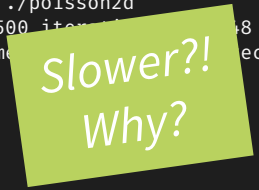
Run Result

```
$ make run
srun --partition=gpus --gres=gpu:1  ./poisson2d
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution.
    0, 0.249999
  100, 0.249760
  200, 0...
Calculate current execution.
    0, 0.249999
  100, 0.249760
  200, 0...
2048x2048: Ref:  89.8862 s, This:  22.8402 s, speedup:     3.94
```

Data Copies

Run Result

```
$ make run
srun --partition=gpus --gres=gpu:1  ./poisson2d
Jacobi relaxation calculation: max 500 iter[illegible]48 x 2048 mesh
Calculate reference solution and time[illegible]ecution.
    0, 0.249999
  100, 0.249760
  200, 0...
Calculate current execution.
    0, 0.249999
  100, 0.249760
  200, 0...
2048x2048: Ref:  89.8862 s, This:  22.8402 s, speedup:     3.94
```

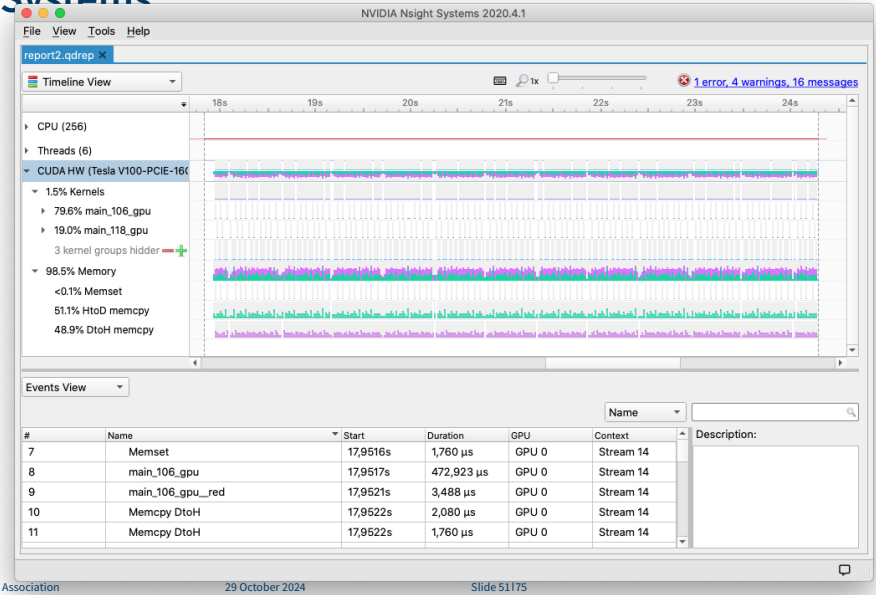


Nsight Systems

- Let’s check again with profiler!
- This time: GUI of Nsight Systems with timeline

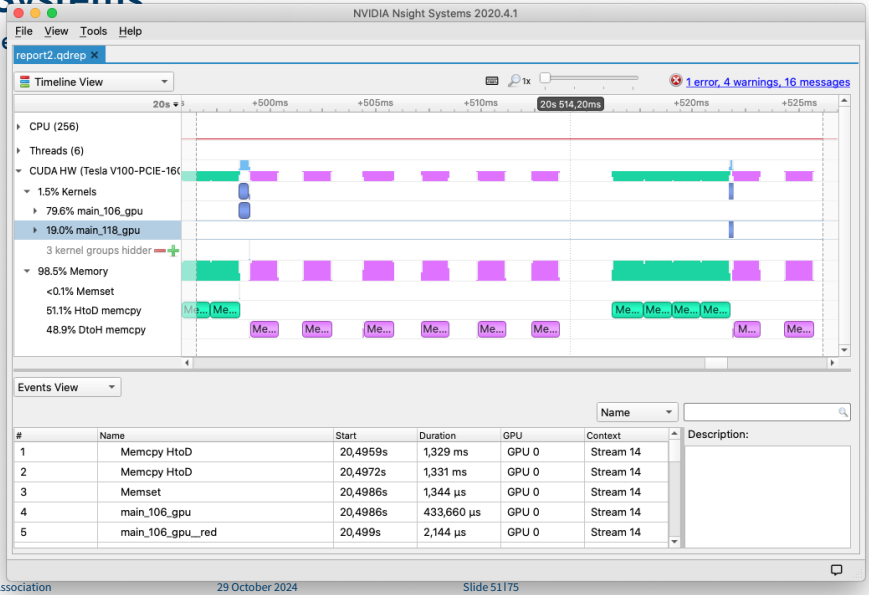
Nsight Systems

Overview

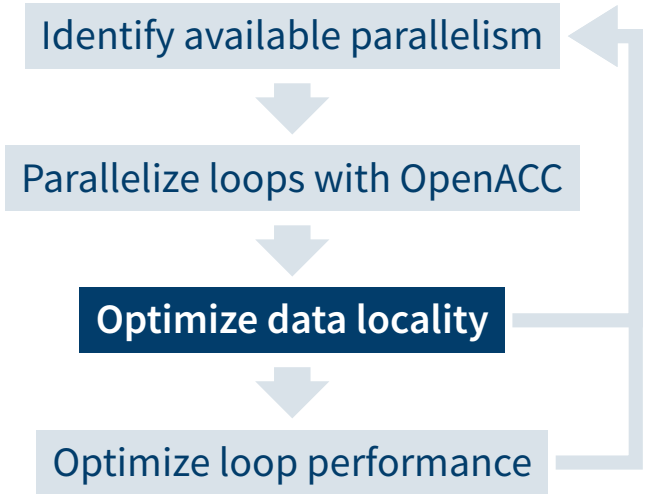


Nsight Systems

Zoom to kerne



Parallelization Workflow



Analyze Jacobi Data Flow

In code

```
while (error > tol && iter < iter_max) {
    error = 0.0;
```

A, Anew resident on **host**

```
#pragma acc parallel loop
```

```
for (int ix = ix_start; ix < ix_end;
↪  ix++) {
    for (int iy = iy_start; iy < iy_end;
    ↪  iy++) {
    // ...
}}
```

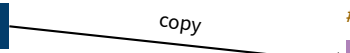
```
    iter++
}
```

Analyze Jacobi Data Flow

In code

```
while (error > tol && iter < iter_max) {
    error = 0.0;
```

A, Anew resident on **host**




```
#pragma acc parallel loop
```

A, Anew resident on **device**

```
for (int ix = ix_start; ix < ix_end;
↪  ix++) {
    for (int iy = iy_start; iy < iy_end;
    ↪  iy++) {
    // ...
}}
```

```
    iter++
}
```

Analyze Jacobi Data Flow

In code

```
while (error > tol && iter < iter_max) {
    error = 0.0;
```

A, Anew resident on **host**


```
#pragma acc parallel loop
```

A, Anew resident on **device**

```
for (int ix = ix_start; ix < ix_end;
↪  ix++) {
    for (int iy = iy_start; iy < iy_end;
    ↪  iy++) {
    // ...
}}
```

A, Anew resident on **device**

```
    iter++
}
```

Analyze Jacobi Data Flow

In code

```
while (error > tol && iter < iter_max) {
    error = 0.0;
```

A, Anew resident on **host**

copy

```
#pragma acc parallel loop
```

A, Anew resident on **device**

```
for (int ix = ix_start; ix < ix_end;
↪  ix++) {
    for (int iy = iy_start; iy < iy_end;
    ↪  iy++) {
    // ...
}}
```

A, Anew resident on **device**

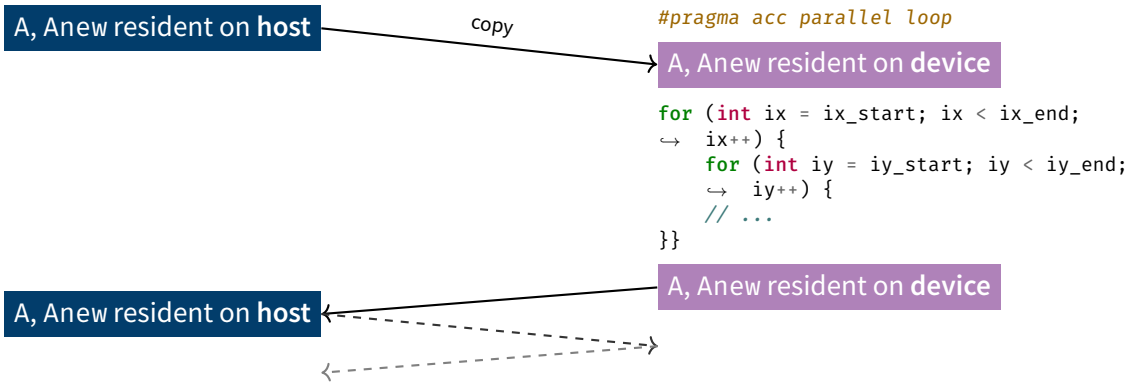
A, Anew resident on **host**

```
    iter++
}
```

Analyze Jacobi Data Flow

In code

```
while (error > tol && iter < iter_max) {
    error = 0.0;
```




```
    iter++
}
```


JÜLICH Forschungszentrum

Analyze Jacobi Data Flow

In code

```
while (error > tol && iter < iter_max) {
    error = 0.0;
```

A, Anew resident on **host**

copy

```
#pragma acc parallel loop
```

A, Anew resident on **device**

```
for (int ix = ix_start; ix < ix_end;
↪  ix++) {
    for (int iy = iy_start; iy < iy_end;
    ↪  iy++) {
    // ...
}}
```

Copies are done between **each** loop *and in **each** iteration!*

A, Anew resident on **device**

A, Anew resident on **host**

```
    iter++
}
```

Analyze Jacobi Data Flow

Summary

- By now, whole algorithm is using GPU
- At beginning of `while` loop, data copied to device; at end of loop, copied back to host
- Depending on type of parallel regions in `while` loop: Data copied in between regions as well

Analyze Jacobi Data Flow

Summary

- By now, whole algorithm is using GPU
- At beginning of `while` loop, data copied to device; at end of loop, copied back to host
- Depending on type of parallel regions in `while` loop: Data copied in between regions as well
- **Slow! Data copies are expensive!**

Data Regions

Structured Data Regions

- Defines region of code in which data remains on device
- Data is shared among all kernels in region
- Explicit data transfers

OpenACC: `data`

```
#pragma acc data [clause, [, clause] ...]
```

Data Regions

Clauses

Clauses to augment the data regions

`copy(var)`	Allocates memory of `var` on GPU, copies data to GPU at beginning of region, copies data to host at end of region Specifies size of `var`: `var[`*`lowerBound`*`:`*`size`*`]`
`copyin(var)`	Allocates memory of `var` on GPU, copies data to GPU at beginning of region
`copyout(var)`	Allocates memory of `var` on GPU, copies data to host at end of region
`create(var)`	Allocates memory of `var` on GPU
`present(var)`	Data of `var` is not copies automatically to GPU but considered present

Data Region Example

```c
#pragma acc data copyout(y[0:N]) create(x[0:N])
{
double sum = 0.0;
#pragma acc parallel loop
for (int i=0; i<N; i++) {
    x[i] = 1.0;
    y[i] = 2.0;
}

#pragma acc parallel loop
for (int i=0; i<N; i++) {
    y[i] = i*x[i]+y[i];
}
}
```

```fortran
!$acc data copyout(y(1:N)) create(x(1,N))

sum = 0.0;
!$acc parallel loop
do i = 1, N
    x(i) = 1.0
    y(i) = 2.0
end do
!$acc end parallel loop
!$acc parallel loop
do i = 1, N
    y(i) = i*x(i)+y(i)
end do
!$acc end parallel loop
!$acc end data
```

Data Regions II

Unstructured Data Regions

- Define data regions, but not for structured block
- Clauses executed at the very position the directive encountered
- Closest to `cudaMemcpy()`
- Still, explicit data transfers

OpenACC: `enter data`

```
#pragma acc enter data [clause, [, clause] ...]
#pragma acc exit data [clause, [, clause] ...]
```

Data Region

More parallelism, Data locality

TASK 5

- Add `data` regions such that all data resides on device during iterations

Task 5: Data Region

- Change to `Task5/` directory
- Work on `TODO`s
- Compile: `make`
- Submit parallel run to the batch system: `make run`

? What's your speed-up?

Parallel Jacobi II

Source Code

```
105 #pragma acc data copy(A[0:nx*ny]) copyin(rhs[0:nx*ny]) create(Anew[0:nx*ny])
106 while ( error > tol && iter < iter_max )
107 {
108     error = 0.0;
109
110     // Jacobi kernel
111     #pragma acc parallel loop reduction(max:error)
112     for (int ix = ix_start; ix < ix_end; ix++)
113     {
114         for (int iy = iy_start; iy < iy_end; iy++)
115         {
116             Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] - ( A[iy*nx+ix+1] + A[iy*nx+ix-1]
117                                                       + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix] ));
118             error = fmaxr( error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
119         }
120     }
121
122     // A <-> Anew
123     #pragma acc parallel loop
124     for (int iy = iy_start; iy < iy_end; iy++)
125     // …
126 }
```

Data Region

Compiler Output

```
$ make
nvc -DUSE_DOUBLE -Minfo=accel -O1 -acc=gpu poisson2d.c poisson2d_reference.o -o poisson2d
poisson2d.c:
main:
    105, Generating create(Anew[:ny*nx]) [if not already present]
         Generating copy(A[:ny*nx]) [if not already present]
         Generating copyin(rhs[:ny*nx]) [if not already present]
    107, Generating Tesla code
        111, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(max:error)
        113, #pragma acc loop seq
        ...
```

Data Region

C

Run Result

```
$ make run
srun --pty ./poisson2d
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution.
    0, 0.249999
  100, 0.249760
  200, 0...
Calculate current execution.
    0, 0.249999
  100, 0.249760
  200, 0...
2048x2048: Ref:  94.3213 s, This:   0.3506 s, speedup:   269.05
```

Data Region

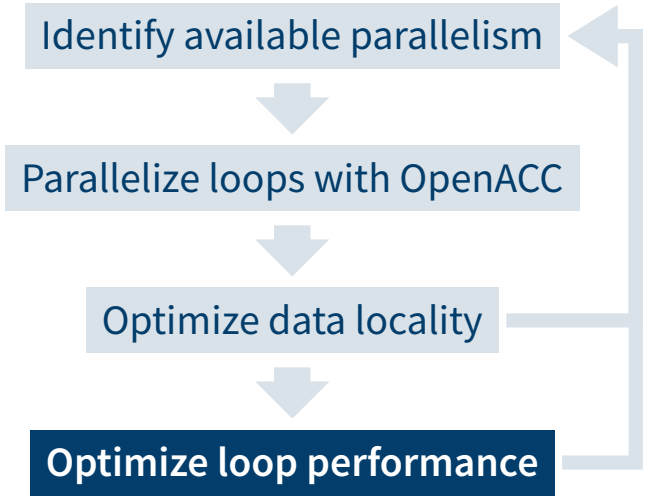
Run Result

Fortran

```
$ make run
srun --pty ./poisson2d
Jacobi relaxation Calculation: 2048 x 2048
Calculate reference solution and time CPU
     0  0.250000
   100  0.002396
   200  0...
GPU execution.
     0  0.250000
   100  0.002396
   200  0...
2048 x 2048: 1 GPU: 0.1570s, 1 CPU cores: 3.5955s, speedup: 22.90
```

JÜLICH Forschungszentrum

Parallelization Workflow



Loop Performance Optimization

Opportunities

```
$ make -B
nvc -DUSE_DOUBLE -Minfo=accel -O1 -acc=gpu poisson2d.c poisson2d_reference.o -o poisson2d
main:
    113, Complex loop carried dependence of Anew-> prevents parallelization
         Loop carried dependence of Anew-> prevents parallelization
         Loop carried backward dependence of Anew-> prevents vectorization
    119, Generating Tesla code
        122, #pragma acc loop gang /* blockIdx.x */
        124, #pragma acc loop vector(128) /* threadIdx.x */
```

To be discussed in other sessions!

OpenACC by Example

Routines

Accelerated Routines

- Enable functions/sub-routines for acceleration
- Make routine callable from device (CUDA: `__device__`)
- Needed for refactoring, modular designs, …
- Position
  - C: At declaration and implementation; immediately before *See next slide*
  - Fortran: Within specification part sub-routine

OpenACC: `routine`

```
#pragma acc routine (name) [clause, [, clause] ...]
```

Routine Details

Clauses to Directive

`gang worker vector seq` Type of parallelism used inside of routine

Routine Details

Clauses to Directive

`gang worker vector seq` Type of parallelism used inside of routine

`(`*`name`*`)` Second version of directive

- Make named routine accelerated
- Applies to function within current scope with name *name*
- To be inserted before definition of named function

Routine Details

Clauses to Directive

`gang worker vector seq` Type of parallelism used inside of routine

`(name)` Second version of directive

- Make named routine accelerated
- Applies to function within current scope with name *name*
- To be inserted before definition of named function

`bind(func)` Bind routine to `func` device function

```
#pragma acc routine bind(func_dev)
void func(float *) {}
void func(float * A) {A[0] = 2;}
#pragma acc routine
void func_dev(float * A) {A[0] = 23;}
int main() {
    float * A = (float*) malloc(1*sizeof(float));
    func(A)  // A[0] == 2
    #pragma acc parallel
    func(A)  // A[0] == 23
}
```

Routine

TASK 6

Getting some routine!

- Extract the inner part of the double for-loop into a dedicated routine called `inner_loop()`
- C: `error` needs to be passed by reference!

Task 6: Routine

- Change to `Task6/` directory
- Work on TODOs
- Compile: `make`
- Submit parallel run to the batch system: `make run`
- *Fortran: Why did it get slower?!*

Jacobi Routine

Source Code

```
42  #pragma acc routine
43  void inner_loop(int ix, int nx, int iy_start, int iy_end, real * A, real * Anew, real * rhs,
    ↪   real * error) {
44      #pragma acc loop
45      for (int iy = iy_start; iy < iy_end; iy++)
46      {
47          Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] - ( A[iy*nx+ix+1] + A[iy*nx+ix-1]
48                                                    + A[(iy-1)*nx+ix] + A[(iy+1)*nx+ix] ));
49          *error = fmaxr( *error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
50      }
51  }
```

Jacobi Routine

Compiler Output

```
$ make
nvc -DUSE_DOUBLE -Minfo=accel -O1 -acc=gpu poisson2d.c poisson2d_reference.o -o poisson2d
poisson2d.c:
inner_loop:
     43, Generating acc routine seq
         Generating Tesla code
main:
    121, Generating create(Anew[:ny*nx]) [if not already present]
...
```

Other Directives

Further Keywords

Directives

`serial`	Serial GPU Region
`wait`	Wait for any async operation
`atomic`	Atomically access data (no interference of concurrent accesses)
`cache`	Fetch data to GPU caches
`declare`	Make data live on GPU for implicit region directly after variable declaration
`update`	Update device data
`shutdown`	Shutdown connection to GPU

Further Keywords

Directives

`serial`	Serial GPU Region
`wait`	Wait for any async operation
`atomic`	Atomically access data (no interference of concurrent accesses)
`cache`	Fetch data to GPU caches
`declare`	Make data live on GPU for implicit region directly after variable declaration
`update`	Update device data
`shutdown`	Shutdown connection to GPU

Clauses

`gang worker vector`	Type of parallelism
`collapse`	Combine tightly-nested loops
`tile`	Split loop into two loops
`(first)private`	Create thread-private data (and init)
`attach`	Reference counting for data pointers
`async`	Schedule operation asynchronously

Further Keywords

Directives

`serial`	Serial GPU Region
`wait`	Wait for any async operation
`atomic`	Atomically access data (no interference of concurrent accesses)
`cache`	Fetch data to GPU caches
`declare`	Make data live on GPU for implicit region directly after variable declaration
`update`	Update device data
`shutdown`	Shutdown connection to GPU

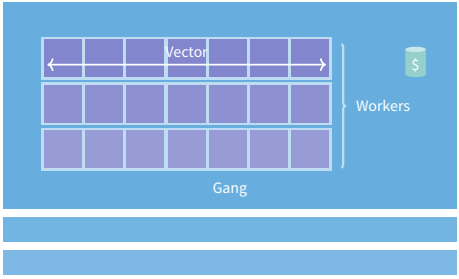
Clauses

`gang worker vector`	Type of parallelism
`collapse`	Combine tightly-nested loops
`tile`	Split loop into two loops
`(first)private`	Create thread-private data (and init)
`attach`	Reference counting for data pointers
`async`	Schedule operation asynchronously

Launch Configuration

Specify number of threads and blocks

- 3 **clauses** for changing distribution of group of threads (clauses of parallel region (`parallel`, `kernels`))
- Presence of keyword: Distribute using this level
- Optional size: Control size of parallel entity



OpenACC: `gang worker vector`

```
#pragma acc parallel loop gang worker vector
```
Size: `num_gangs(n)`, `num_workers(n)`, `vector_length(n)`

Conclusions

Conclusions

- OpenACC directives and clauses
  `#pragma acc parallel loop copyin(A[0:N]) reduction(max:err) vector`
- Start easy, optimize from there; express as much parallelism as possible
- Optimize data for locality, prevent unnecessary movements
- OpenACC is interoperable to other GPU programming models

Conclusions

- OpenACC directives and clauses
  `#pragma acc parallel loop copyin(A[0:N]) reduction(max:err) vector`
- Start easy, optimize from there; express as much parallelism as possible
- Optimize data for locality, prevent unnecessary movements
- OpenACC is interoperable to other GPU programming models

Thank you for your attention!

a.herten@fz-juelich.de

Appendix

- List of Tasks
- Glossary
- References

List of Tasks

Glossary I

AMD Manufacturer of CPUs and GPUs. 11, 12, 13

Ampere GPU architecture from NVIDIA (announced 2019). 14

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 15, 68, 69, 70, 71, 72

GCC The GNU Compiler Collection, the collection of open source compilers, among others for C and Fortran. 14, 29, 30

LLVM An open Source compiler infrastructure, providing, among others, Clang for C. 11, 12, 13

NVHPC NVIDIA HPC SDK; Collection of GPU-capable compilers and libraries. Formerly known as PGI.. 14

Glossary II

NVIDIA US technology company creating GPUs. 4, 11, 12, 13, 53, 67, 124, 125, 126

OpenACC Directive-based programming, primarily for many-core machines. 2, 4, 5, 6, 7, 8, 9, 10, 14, 15, 16, 17, 18, 19, 28, 36, 37, 38, 39, 41, 42, 44, 45, 46, 49, 56, 61, 66, 77, 78, 86, 95, 98, 104, 106, 107, 118, 120, 121

OpenMP Directive-based programming, primarily for multi-threaded machines. 2, 5, 6, 7, 11, 12, 13, 58, 59, 60

PAPI The Performance API, a C/C++ API for querying performance counters. 29, 30

Pascal GPU architecture from NVIDIA (announced 2016). 68, 69, 70, 71, 72

perf Part of the Linux kernel which facilitates access to performance counters; comes with command line utilities. 29, 30

Glossary III

PGI Compiler creators. Formerly *The Portland Group, Inc.*; since 2013 part of NVIDIA. 124

POWER CPU architecture from IBM, earlier: PowerPC. See also POWER8. 68, 69, 70, 71, 72, 126

POWER8 Version 8 of IBM's POWER processor, available also within the OpenPOWER Foundation. 126

CPU Central Processing Unit. 11, 12, 13, 68, 69, 70, 71, 72, 124, 126

GPU Graphics Processing Unit. 2, 11, 12, 13, 16, 49, 53, 67, 68, 69, 70, 71, 72, 93, 94, 96, 120, 121, 124, 125

References I

[3] Donald E. Knuth. “Structured Programming with Go to Statements.” In: *ACM Comput. Surv.* 6.4 (Dec. 1974), pp. 261–301. ISSN: 0360-0300. DOI: 10.1145/356635.356640. URL: http://doi.acm.org/10.1145/356635.356640 (pages 29, 30).

References: Images, Graphics

[1] Bill Jelen. *SpaceX Falcon Heavy Launch*. Freely available at Unsplash. URL: https://unsplash.com/photos/lDEMa5dPcNo.

[2] Setyo Ari Wibowo. *Ask*. URL: https://thenounproject.com/term/ask/1221810.