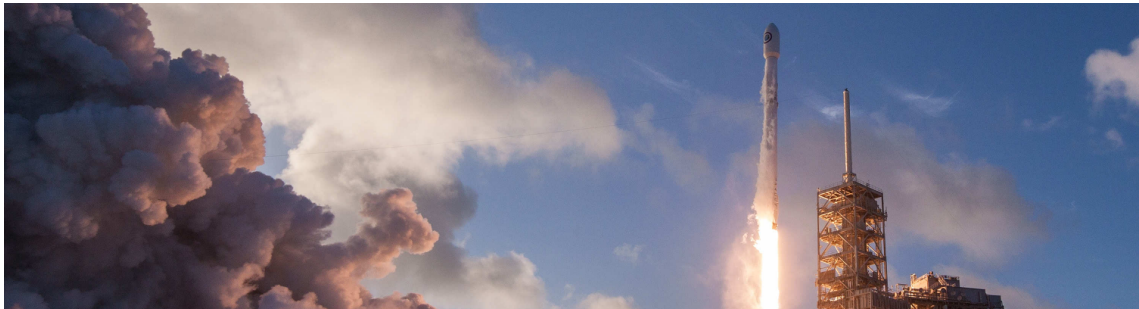




PRODUCTIVE GPU PROGRAMMING WITH OPENACC

GSP GPU COURSE 2018

9 August 2018 | Andreas Herten | Forschungszentrum Jülich, Jülich Supercomputing Centre

Overview, Outline

What you will learn today

- Parallelization strategies with OpenACC
- OpenACC on CPU, GPU

What you will not learn today

- Analyze program in-detail
- Strategies for complex programs
- How to leave the matrix

Overview, Outline

What you will learn today

- Parallelization strategies with OpenACC
- OpenACC on CPU, GPU

What you will not learn today

- Analyze program in-detail
- Strategies for complex programs
- How to leave the matrix

OpenACC Introduction

OpenACC on CPU

OpenACC: GPU Optimizations

OpenACC with GPUs

MPI 101

OpenACC, GPUs, and MPI

Hands-on

Lecture

Overview, Outline

What you will learn today

- Parallelization strategies with OpenACC
- OpenACC on CPU, GPU

What you will not learn today

- Analyze program in-detail
- Strategies for complex programs
- How to leave the matrix

OpenACC Introduction

About OpenACC

Modus Operandi

OpenACC's Models

Parallelization Workflow

First Steps in OpenACC

Example Program

Identify Parallelism

Parallelize Loops

parallel

loops

kernels

OpenACC on the GPU

Compiling on GPU

Data Transfers

Portability

Clause: copy

Data Locality

Analyse Flow

data

enter data

Pinned

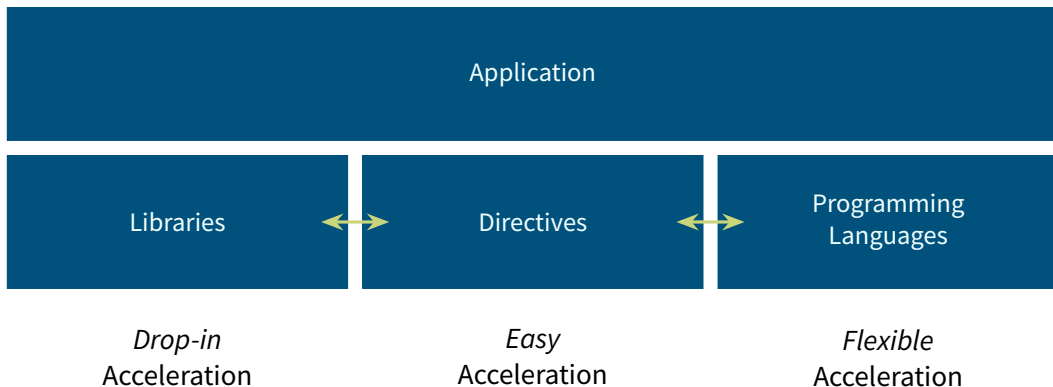
Appendix

List of Tasks

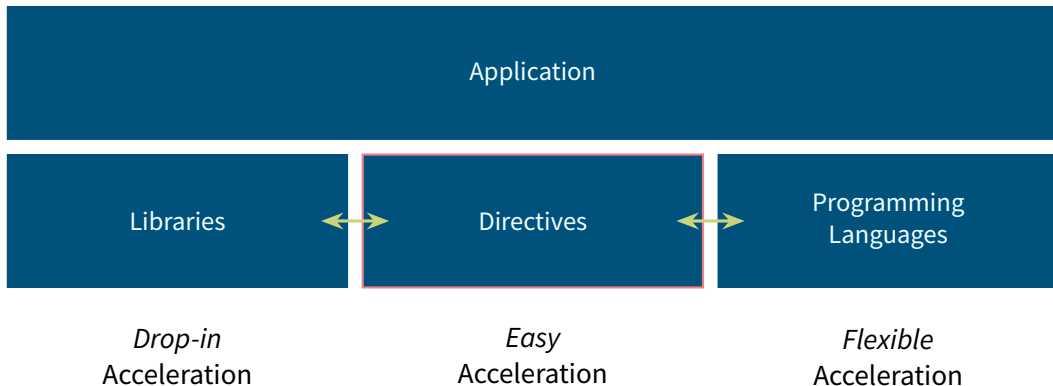


OpenACC Introduction

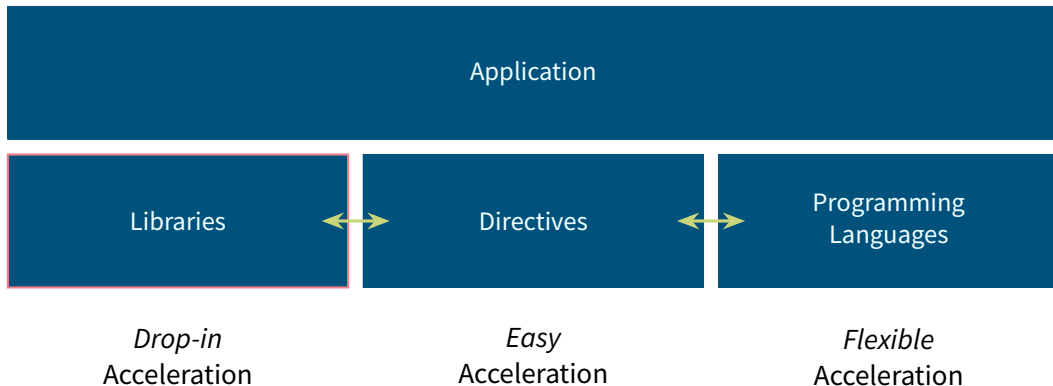
Primer on GPU Computing



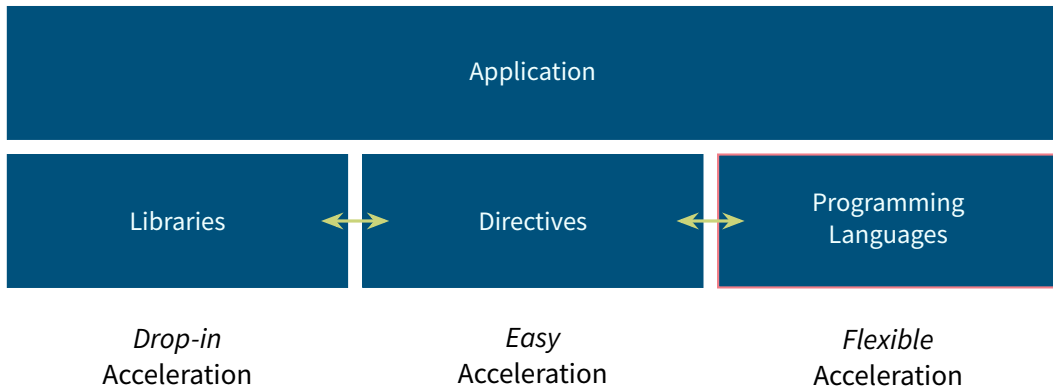
Primer on GPU Computing



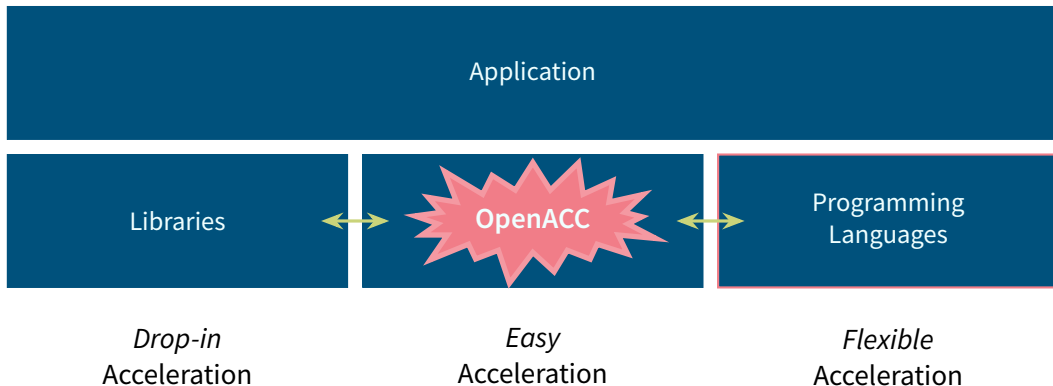
Primer on GPU Computing



Primer on GPU Computing



Primer on GPU Computing



About OpenACC

History

2011 OpenACC 1.0 specification is released 

NVIDIA, Cray, PGI, CAPS

2013 OpenACC 2.0: More functionality, portability 

2015 OpenACC 2.5: Enhancements, clarifications 

2017 OpenACC 2.6: Deep copy, ... 

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

Support

- Compiler: PGI, GCC, Cray, *Sunway*
- Languages: C/C++, Fortran

Open{MP $\leftrightarrow$ ACC}

Everything's connected

- OpenACC modeled after OpenMP ...
- ... but specific for accelerators
- Might eventually be absorbed into OpenMP

But OpenMP >4.0 also has offloading feature

- OpenACC more descriptive, OpenMP more prescriptive
- Basic principle same: Fork/join model

Master thread launches parallel child threads; merge after execution



Open{MP↔ACC}

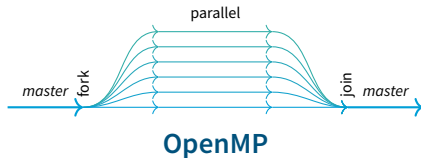
Everything's connected

- OpenACC modeled after OpenMP ...
- ... but specific for accelerators
- Might eventually be absorbed into OpenMP

But OpenMP >4.0 also has offloading feature

- OpenACC more descriptive, OpenMP more prescriptive
- Basic principle same: Fork/join model

Master thread launches parallel child threads; merge after execution



Open{MP↔ACC}

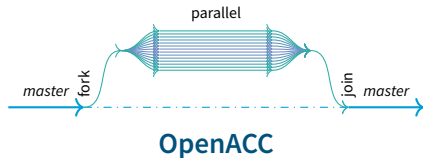
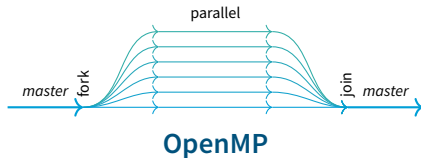
Everything's connected

- OpenACC modeled after OpenMP ...
- ... but specific for accelerators
- Might eventually be absorbed into OpenMP

But OpenMP >4.0 also has offloading feature

- OpenACC more descriptive, OpenMP more prescriptive
- Basic principle same: Fork/join model

Master thread launches parallel child threads; merge after execution



Modus Operandi

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
- High level programming model for many-core machines, especially accelerators
- OpenACC: Compiler directives, library routines, environment variables
- Portable across host systems and accelerator architectures

2 Compiler

Simple and abstracted

- Compiler support
 - PGI *Best performance, great support, free*
 - GCC *Beta, limited coverage, OSS*
 - Cray ???
- Trust compiler to generate intended parallelism; always check status output!
- No need to know ins'n'outs of accelerator; leave it to expert compiler engineers*
- One code can target different accelerators: GPUs, or even multi-core CPUs → **Portability**

**: Eventually you want to tune for device; but that's possible*

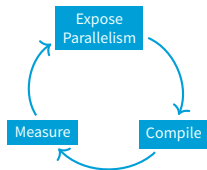
3 \$uccess

Iteration is key

- Serial to parallel: fast
- Serial to fast parallel: more time needed
- Start simple → refine

⇒ Productivity

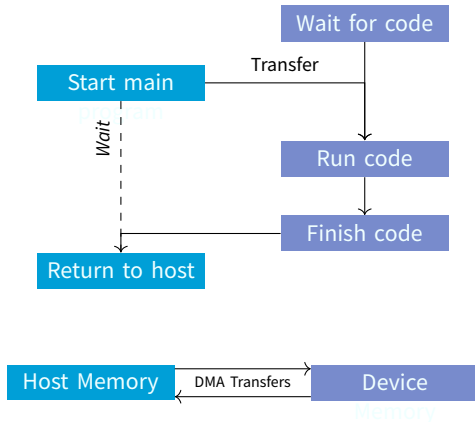
- Because of *generalness*: Sometimes not last bit of hardware performance accessible
- But: Use OpenACC together with other accelerator-targeting techniques (CUDA, libraries, ...)



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



OpenACC Programming Model

A binary perspective

- OpenACC interpretation needs to be activated as compile flag

PGI `pgcc -acc [-ta=tesla|-ta=multicore]`

GCC `gcc -fopenacc`

→ Ignored by incapable compiler!

- Additional flags possible to improve/modify compilation

`-ta=tesla:cc60` Use compute capability 6.0

`-ta=tesla:lineinfo` Add source code correlation into binary

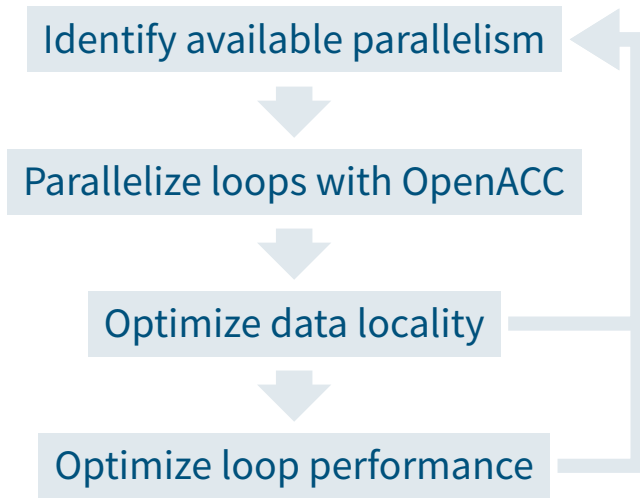
`-ta=tesla:managed` Use unified memory

`-fopenacc-dim=geom` Use *geom* configuration for threads

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];
    }
}
```

Parallelization Workflow

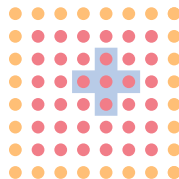
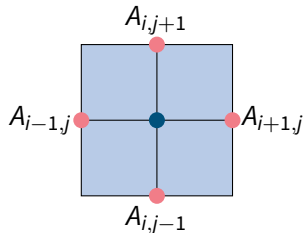


First Steps in OpenACC

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: $\nabla^2 A(x, y) = B(x, y)$



- Data Point
- Boundary Point
- Stencil

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

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

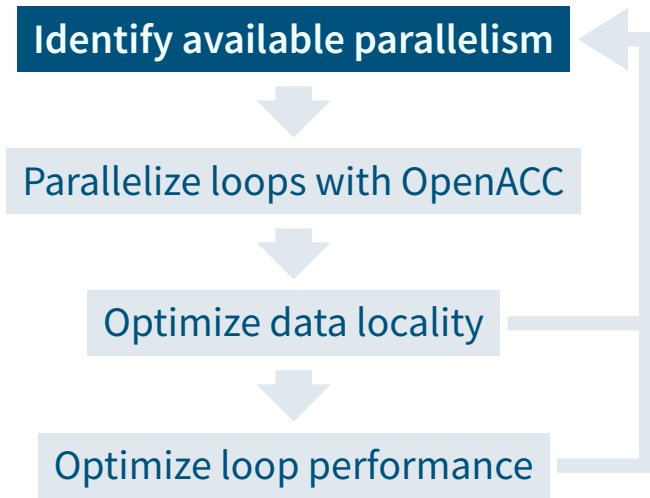
Calculate new value from neighbors

Accumulate error

Swap input/output

Set boundary conditions

Parallelization Workflow



Profiling

Profile

[...] premature optimization is the root of all evil.

– Donald Knuth [2]

- Investigate hot spots of your program!
- Profile!
- Many tools, many levels: perf, PAPI, Score-P, Intel Advisor, NVIDIA Visual Profiler, ...
 - Here: Examples from PGI

Profiling

Profile

[...] premature optimization is the root of all evil.

Yet we should not pass up our [optimization] opportunities [...]

– Donald Knuth [2]

- Investigate hot spots of your program!
- Profile!
 - Many tools, many levels: perf, PAPI, Score-P, Intel Advisor, NVIDIA Visual Profiler, ...
 - Here: Examples from PGI

Profile of Application

Info during compilation

```
$ pgcc -DUSE_DOUBLE -Minfo=all,intensity -fast -Minfo=ccff -Mprof=ccff
poisson2d_reference.o poisson2d.c -o poisson2d
poisson2d.c:
main:
    68, Generated vector simd code for the loop
        FMA (fused multiply-add) instruction(s) generated
    98, FMA (fused multiply-add) instruction(s) generated
   105, Loop not vectorized: data dependency
   123, Loop not fused: different loop trip count
        Loop not vectorized: data dependency
        Loop unrolled 8 times
```

- Automated optimization of compiler, due to -fast
- Vectorization, FMA, unrolling

Profile of Application

Info during run

```
$ pgprof --cpu-profiling on [...] ./poisson2d
===== CPU profiling result (flat):
Time(%)      Time   Name
77.52%      999.99ms  main (poisson2d.c:148 0x6d8)
 9.30%       120ms   main (0x704)
 7.75%      99.999ms  main (0x718)
 0.78%      9.9999ms  main (poisson2d.c:128 0x348)
 0.78%      9.9999ms  main (poisson2d.c:123 0x398)
 0.78%      9.9999ms  __xlmass_expd2 (0xffcc011c)
 0.78%      9.9999ms  __c_mcopy8 (0xffcc0054)
 0.78%      9.9999ms  __xlmass_expd2 (0xffcc0034)
===== Data collected at 100Hz frequency
```

- 78 % in main()
- Since everything is in main – limited helpfulness
- Let's look into main!

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++;  
}
```

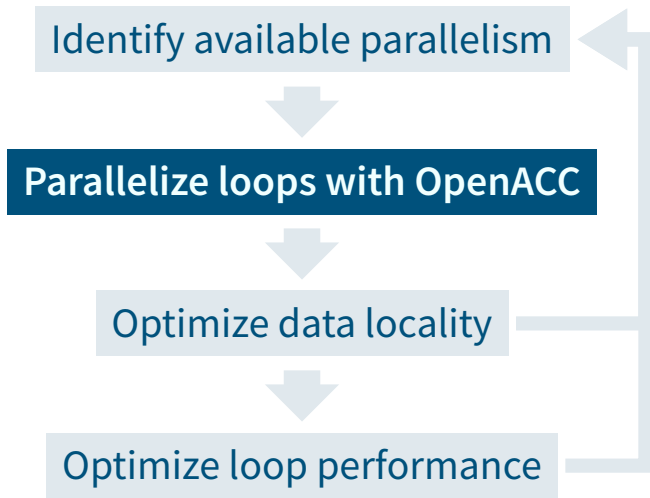
Data dependency
between iterations

Independent loop
iterations

Independent loop
iterations

Independent loop
iterations

Parallelization Workflow



Parallel Loops: Parallel

Maybe the second most important directive

- Programmer identifies block containing parallelism
→ compiler generates parallel code (*kernel*)
- 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

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

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 be 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

Maybe the third most important directive

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

🔗 OpenACC: loop

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

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`

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

Parallel Loops Example

```
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];
    }
}
```

Kernel 1

Kernel 2

Add parallelism

- Add OpenACC parallelism to main loop in Jacobi solver source code (CPU parallelism)
- Congratulations, you are a parallel developer!

Task 2: A First Parallel Loop

- Change to Task2/ directory
 - Compile: `make`; see `README.md`
 - Submit run to the batch system: `make run`
Adapt the `bsub` call and run with other number of iterations, matrix sizes
 - Change number of CPU threads via `$ACC_NUM_CORES` or `$OMP_NUM_THREADS`
- ? What's your speed-up? What's the best configuration for cores?

E Compare it to OpenMP

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] - ( A[iy*nx+ix+1] + A[iy*nx+ix-1]
116                                                    + A[(iy-1)*nx+ix] +
117                                                    ↪ 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
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=multicore poisson2d.c poisson2d_reference.o
-o poisson2d
poisson2d.c:
main:
110, Generating Multicore code
111, #pragma acc loop gang
110, Generating reduction(max:error)
113, Accelerator restriction: size of the GPU copy of A,rhs,Anew is unknown
Complex loop carried dependence of Anew-> prevents parallelization
Loop carried dependence of Anew-> prevents parallelization
Loop carried backward dependence of Anew-> prevents vectorization
```

Parallel Jacobi

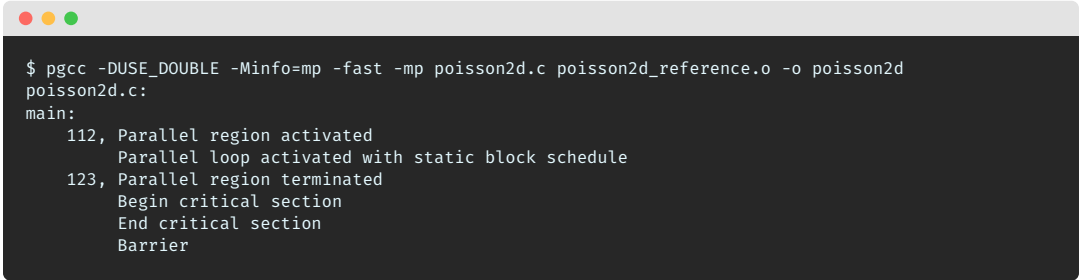
Run result

```
$ 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: 60.5136 s, This: 8.2483 s, speedup: 7.34
```

- OpenMP pragma is quite similar

```
#pragma acc parallel loop reduction(max:error)
#pragma omp parallel for reduction(max:error)
for (int ix = ix_start; ix < ix_end; ix++) { ... }
```

- PGI's compiler output is a bit different (but states the same)

A terminal window with a grey title bar and three colored window control buttons (red, yellow, green) on the left. The terminal has a dark background and displays the output of a PGI compiler command. The output shows the activation and termination of a parallel region, followed by critical sections and a barrier.

```
$ pgcc -DUSE_DOUBLE -Minfo=mp -fast -mp poisson2d.c poisson2d_reference.o -o poisson2d
poisson2d.c:
main:
  112, Parallel region activated
      Parallel loop activated with static block schedule
  123, Parallel region terminated
      Begin critical section
      End critical section
      Barrier
```

- Run time should be very similar!

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 created here

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

OpenACC on the GPU

Changes for GPU-OpenACC

Immensely complicated changes

- Necessary for previous code to run on GPU: `-ta=tesla` instead of `-ta=multicore`

Changes for GPU-OpenACC

Immensely complicated changes

- Necessary for previous code to run on GPU: `-ta=tesla` instead of `-ta=multicore`

⇒ **That's it!**

Changes for GPU-OpenACC

Immensely complicated changes

- Necessary for previous code to run on GPU: `-ta=tesla` instead of `-ta=multicore`

⇒ **That's it!**

- But we can optimize!

Changes for GPU-OpenACC

Immensely complicated changes

- Necessary for previous code to run on GPU: `-ta=tesla` instead of `-ta=multicore`

⇒ **That's it!**

- But we can optimize!
- First: A task

Parallel Jacobi on GPU

TASK 3

More parallelism, more corres

- Add OpenACC parallelism to other loops of **while** (L:123 – L:141)
Use either `kernel`s or `parallel`
- Make sure to use `-ta=tesla` when compiling.

Task 3: More Parallel Loops on GPU

- Change to Task3/ directory
 - Compile: `make`
 - Submit parallel run to the batch system: `make run`
- ? What's your speed-up?

Parallel Jacobi II

Compilation result

```
$ make
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc70,managed poisson2d.c poisson2d_reference.o -o poisson2d
poisson2d.c:
main:
    109, Accelerator kernel generated
        Generating Tesla code
    109, Generating reduction(max:error)
    110, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    112, #pragma acc loop seq
    109, ...
    121, Accelerator kernel generated
        Generating Tesla code
    124, #pragma acc loop gang /* blockIdx.x */
    126, #pragma acc loop vector(128) /* threadIdx.x */
    121, Generating implicit copyin(Anew[:])
        Generating implicit copyout(A[:])
    126, Loop is parallelizable
    133, Accelerator kernel genera...
```



Parallel Jacobi II

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:  40.3299 s, This:  0.3243 s, speedup:  124.36
```

Parallel Jacobi II

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:  40.3299 s, This:  0.3243 s, speedup:  124.36
```

Done?!

Parallel Jacobi

```
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++;
}
```

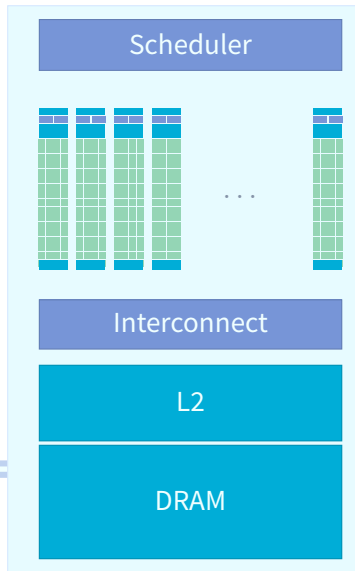
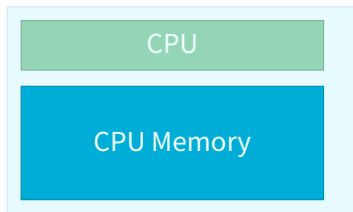
Automatic Data Transfers

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

NVIDIA GPU Memory Spaces

Location, location, location

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

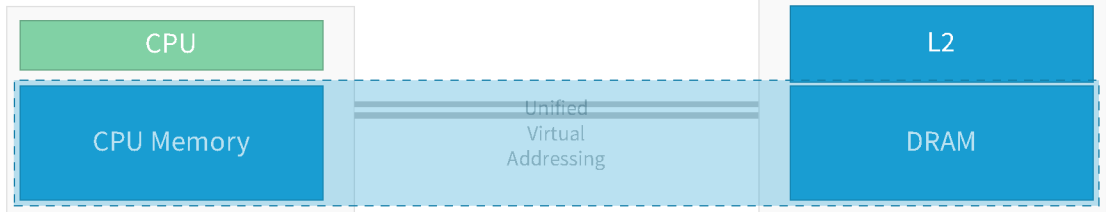


NVIDIA GPU Memory Spaces

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



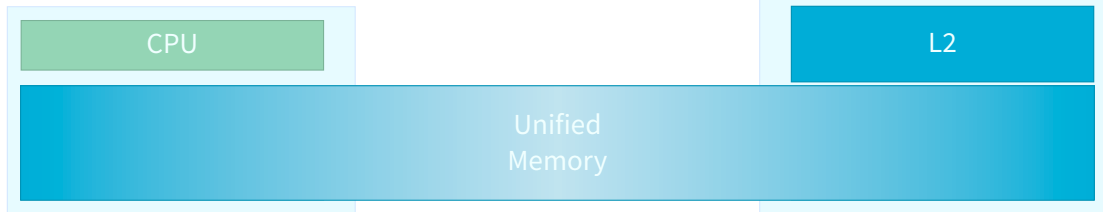
NVIDIA GPU Memory Spaces

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 (Kepler)



NVIDIA GPU Memory Spaces

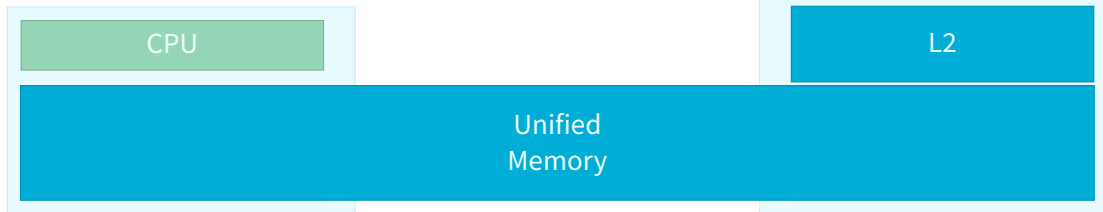
Location, location, location

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 (Kepler)

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



Portability

- Managed memory: Only NVIDIA GPU feature
 - Great OpenACC features: Portability
- Code should also be fast without `-ta=tesla:managed!`
- Let's remove it from compile flags!

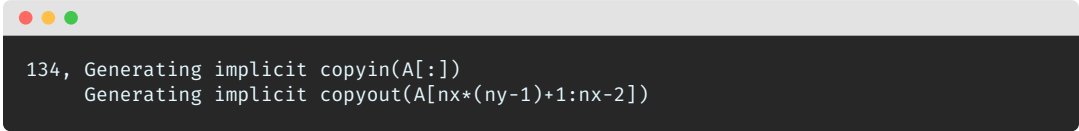
Portability

- Managed memory: Only NVIDIA GPU feature
 - Great OpenACC features: Portability
- Code should also be fast without `-ta=tesla:managed!`
- Let's remove it from compile flags!

```
$ make
pgcc -c -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60
poisson2d_reference.c -o poisson2d_reference.o
poisson2d.c:
PGC-S-0155-Compiler failed to translate accelerator region
(see -Minfo messages): Could not find allocated-variable index for
symbol (poisson2d.c: 110)
...
PGC/power Linux 17.4-0: compilation completed with severe errors
```

Copy Statements

- Compiler implicitly created copy clauses to copy data to device



```
134, Generating implicit copyin(A[:])  
      Generating implicit copyout(A[nx*(ny-1)+1:nx-2])
```

- 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

```
134, Generating implicit copyin(A[:])  
      Generating implicit copyout(A[nx*(ny-1)+1:nx-2])
```

- It couldn't determine length of copied data ...
- ...but before: no problem – Unified Memory!
- Now: Problem! We need to give that information! (see also [later](#))

🔑 OpenACC: copy

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

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

Data Copies

Get that data!

TASK 4

- Add copy clause to parallel regions
- Check correctness with Visual Profiler

Task 4: Data Copies

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

? What's your speed-up?



Data Copies

Compiler Output

```
$ make
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc70 poisson2d.c poisson2d_reference.o -o poisson2d
poisson2d.c:
main:
  109, Generating copy(A[:ny*nx],Anew[:ny*nx],rhs[:ny*nx])
      ...
  121, Generating copy(Anew[:ny*nx],A[:ny*nx])
      ...
  131, Generating copy(A[:ny*nx])
      Accelerator kernel generated
      Generating Tesla code
  132, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
  137, Generating copy(A[:ny*nx])
      Accelerator kernel generated
      Generating Tesla code
  138, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

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: 60.0229 s, This: 69.5278 s, speedup: 0.86
```

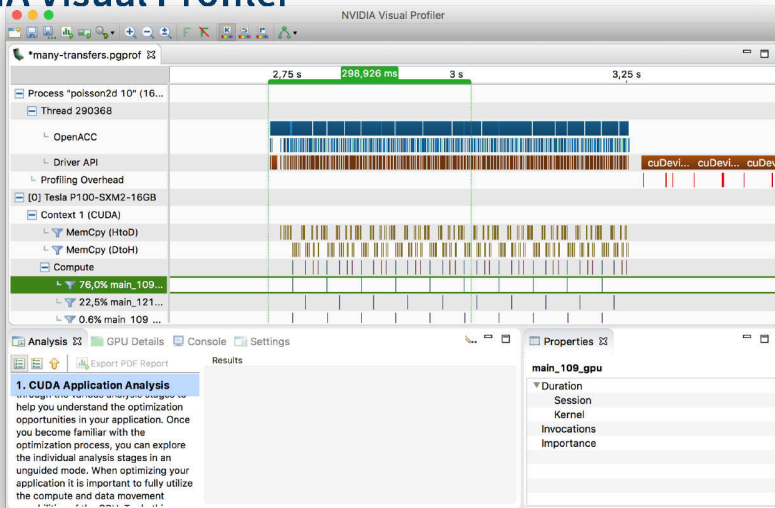
Data Copies

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 execution.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref: 60.0229 s, This: 69.5278 s, speedup: 0.86
```

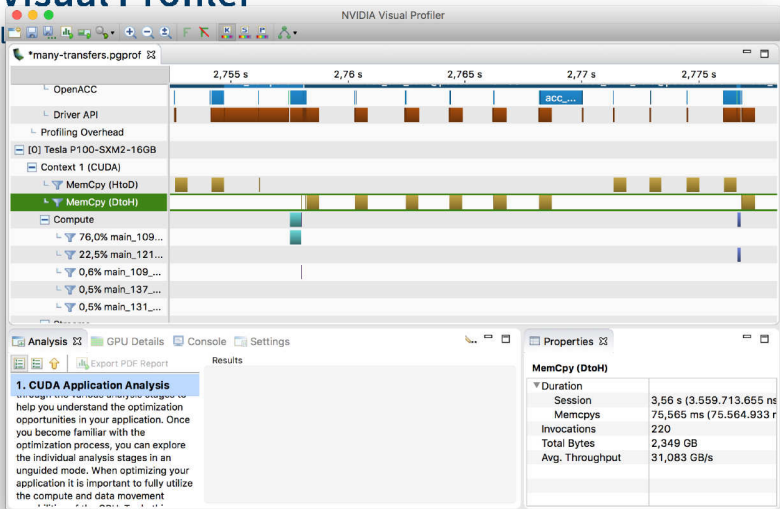
*Slower?!
Why?*

PGI/NVIDIA Visual Profiler

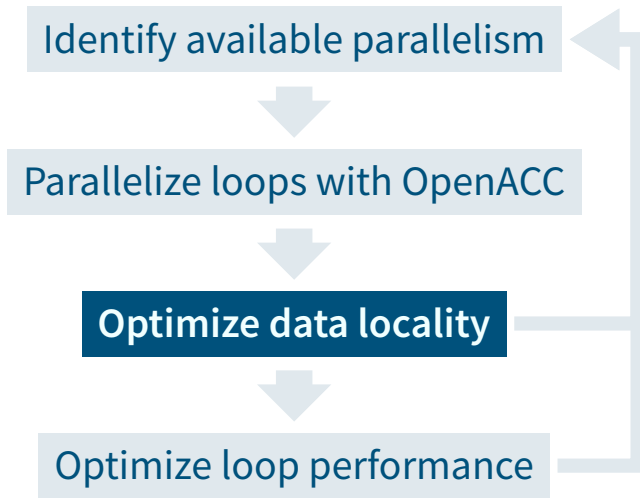


Jacobi in Visual Profiler

Zoom in to kernel



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

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++) {  
            // ...  
        }  
    }
```

```
        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

```
        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 host

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 host

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++) {  
            // ...  
        }  
    }
```

Copies are done
in each iteration!

A, Anew resident on host

A, Anew resident on device



iter++

}

Data Regions

To manually specify data locations: `data` construct

- 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 copied automatically to GPU but considered present

Data Region Example

```
#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];
  }
}
```

Data Regions II

Looser regions: `enter` data directive

- Define data regions, but not for structured block
- Closest to `cudaMemcpy()`
- Still, explicit data transfers

🔗 OpenACC: `enter` data

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

Data Region

TASK 5

More parallelism, Data locality

- Add data regions such that all data resides on device during iterations
- Optional: See your success in Visual Profiler

Task 5: Data Region

- Change to Task5/ directory
- Work on TODOs
- Compile: make
- Submit parallel run to the batch system: make run
- ? What's your speed-up?
- Generate profile with make profile_tofile



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 }
```

Parallel Jacobi II

Compilation result

```
$ make
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc70 poisson2d.c poisson2d_reference.o
-o poisson2d
poisson2d.c:
poisson2d.c:
main:
105, Generating copyin(rhs[:ny*nx])
    Generating create(Anew[:ny*nx])
    Generating copy(A[:ny*nx])
111, Accelerator kernel generated
    Generating Tesla code
111, Generating reduction(max:error)
112, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
114, #pragma acc loop seq
114, Complex loop carried dependence of Anew-> prevents parallelization
    Loop carried dependence of Anew-> prevents parallelization
```

Parallel Jacobi II

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:  59.5508 s, This:  0.3328 s, speedup:  178.95
```

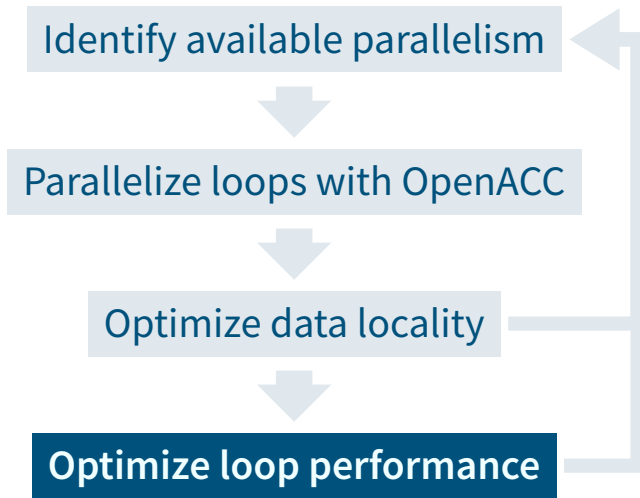
Parallel Jacobi II

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
  0, 0.249999
 100, 0.249760
 200, 0...
Calculate current execution
  0, 0.249999
 100, 0.249760
 200, 0...
2048x2048: Ref: 59.5508 s, This: 0.3328 s, speedup: 178.95
```

Nice!
But can we be even better?

Parallelization Workflow



Understanding Compiler Output

```
110, Accelerator kernel generated
    Generating Tesla code
110, Generating reduction(max:error)
111, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
114, #pragma acc loop seq
114, Complex loop carried dependence of Anew-> prevents parallelization
```

```
110  #pragma acc parallel loop reduction(max:error)
111  for (int ix = ix_start; ix < ix_end; ix++)
112  {
113      // Inner loop
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  }
```

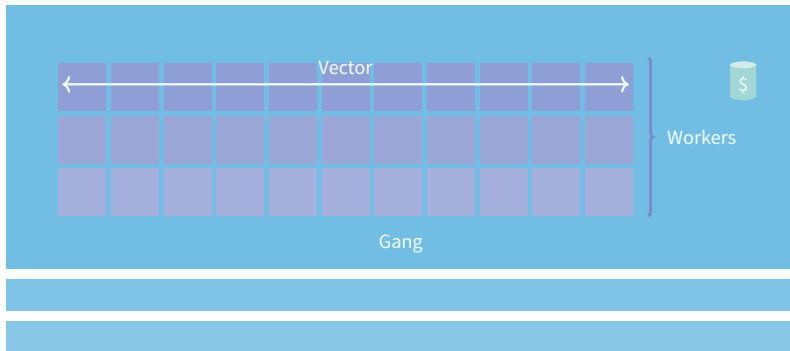
Understanding Compiler Output

```
110, Accelerator kernel generated
    Generating Tesla code
110, Generating reduction(max:error)
111, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
114, #pragma acc loop seq
114, Complex loop carried dependence of Anew-> prevents parallelization
```

- Outer loop: Parallelism with gang and vector
- Inner loop: Sequentially per thread (#pragma acc loop seq)
- Inner loop was never parallelized!
- **Rule of thumb:** Expose as much parallelism as possible

OpenACC Parallelism

3 Levels of Parallelism



Vector

Vector threads work in lockstep (SIMD/SIMT parallelism)

Worker

Has 1 or more vector; workers share common resource (*cache*)

Gang

Has 1 or more workers; multiple gangs work independently from each other

CUDA Parallelism

CUDA Execution Model

Software



Thread

Hardware



Scalar
Processor

- **Threads** executed by scalar processors (*CUDA cores*)



CUDA Parallelism

CUDA Execution Model

Software



Thread



Thread Block

Hardware



Scalar Processor

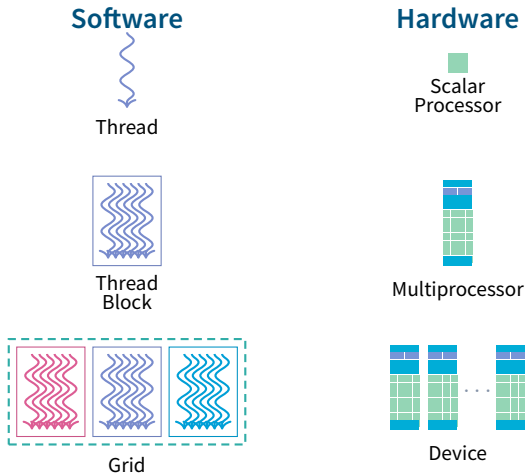


Multiprocessor

- **Threads** executed by scalar processors (*CUDA cores*)
- Thread **blocks**: Executed on multiprocessors (*SM*)
- Do not migrate
- Several concurrent thread blocks can reside on multiprocessor
Limit: Multiprocessor resources (register file; shared memory)

CUDA Parallelism

CUDA Execution Model



- **Threads** executed by scalar processors (*CUDA cores*)
- Thread **blocks**: Executed on multiprocessors (*SM*)
- Do not migrate
- Several concurrent thread blocks can reside on multiprocessor
Limit: Multiprocessor resources (register file; shared memory)
- Kernel launched as **grid** of thread blocks
- Blocks, grids: Multiple dimensions

From OpenACC to CUDA

`map(||acc, ||<<<>>>)`

- In general: Compiler free to do what it thinks is best
- Usually
 - `gang` Mapped to blocks (*coarse grain*)
 - `worker` Mapped to threads (*fine grain*)
 - `vector` Mapped to threads (*fine SIMD/SIMT*)
 - `seq` No parallelism; sequential
- Exact mapping compiler dependent
- Performance tips
 - Use vector size divisible by 32
 - Block size: `num_workers` $\times$ `vector_length`



Declaration of Parallelism

Specify configuration of threads

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

🔑 OpenACC: `gang worker vector`

```
#pragma acc parallel loop gang vector
```

Also: `worker`

Size: `num_gangs(n)`, `num_workers(n)`, `vector_length(n)`

Understanding Compiler Output II

```
110, Accelerator kernel generated
    Generating Tesla code
110, Generating reduction(max:error)
111, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
114, #pragma acc loop seq
114, Complex loop carried dependence of Anew-> prevents parallelization
```

- Compiler reports configuration of parallel entities
 - Gang mapped to `blockIdx.x`
 - Vector mapped to `threadIdx.x`
 - Worker not used
- Here: 128 threads per block; as many blocks as needed

128 seems to be default for Tesla/NVIDIA

More Parallelism

TASK 6

Unsequentialize inner loop

- Add vector clause to inner loop
- Study result with profiler

Task 6: More Parallelism

- Change to Task6/ directory
 - Work on TODOs
 - Compile: `make`
 - Submit to the batch system: `make run`
 - Generate profile with `make profile_tofile`
- ? What's your speed-up?



More Parallelism

Compiler Output

```
$ make
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc70 poisson2d.c poisson2d_reference.o
-o poisson2d
poisson2d.c:
main:
    104, Generating create(Anew[:ny*nx])
        Generating copyin(rhs[:ny*nx])
        Generating copy(A[:ny*nx])
    110, Accelerator kernel generated
        Generating Tesla code
    110, Generating reduction(max:error)
    111, #pragma acc loop gang /* blockIdx.x */
    114, #pragma acc loop vector(128) /* threadIdx.x */
    ...
```

Data Region

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: 60.8886 s, This: 0.7658 s, speedup: 79.51
```

Data Region

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 error.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref: 60.8886 s, This: 0.7658 s, speedup: 79.51
```

*Actually slower!
Why?*

Memory Coalescing

Memory in batch

- Coalesced access *good*
 - Threads of warp (group of 32 contiguous threads) access adjacent words
 - Few transactions, high utilization
- Uncoalesced access *bad*
 - Threads of warp access scattered words
 - Many transactions, low utilization
- Best **performance**: `threadIdx.x` should access contiguously



Jacobi Access Pattern

A coalescion of data

Improve memory access pattern: Loop order in main loop

```
#pragma acc parallel loop reduction(max:error)
for (int ix = ix_start; ix < ix_end; ix++) {
    #pragma acc loop vector
    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] ));
        //...
```

ix Outer run index; accesses consecutive memory locations

iy Inner run index; accesses offset memory locations

→ Change order to optimize pattern!

Jacobi Access Pattern

A coalescion of data

Improve memory access pattern: Loop order in main loop

```
#pragma acc parallel loop reduction(max:error)
for (int iy = iy_start; iy < iy_end; iy++) {
    #pragma acc loop vector
    for (int ix = ix_start; ix < ix_end; ix++) {
        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]));
        //...
```

ix Outer run index; accesses consecutive memory locations

iy Inner run index; accesses offset memory locations

→ Change order to optimize pattern!

Fixing Access Pattern

TASK 7

Loop change

- Interchange loop order for Jacobi loops
- Also: Compare to loop-fixed CPU reference version

Task 7: Loop Ordering

- Change to Task7/ directory
 - Work on TOD0s
 - Compile: make
 - Submit to the batch system: make run
- ? What's your speed-up?

Fixing Access Pattern

Compiler output (unchanged)

```
$ make
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc70 poisson2d.c poisson2d_reference.o
-o poisson2d
poisson2d.c:
main:
    104, Generating create(Anew[:ny*nx])
        Generating copyin(rhs[:ny*nx])
        Generating copy(A[:ny*nx])
    110, Accelerator kernel generated
        Generating Tesla code
    110, Generating reduction(max:error)
    111, #pragma acc loop gang /* blockIdx.x */
    114, #pragma acc loop vector(128) /* threadIdx.x */
    ...
```

Fixing Access Pattern

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: 113.0214 s, This:  0.3284 s, speedup:  344.15
```

Fixing Access Pattern

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
  0, 0.249999
 100, 0.249760
 200, 0...
Calculate current execution
  0, 0.249999
 100, 0.249760
 200, 0...
2048x2048: Ref: 113.0214 s, This:  0.3284 s, speedup:  344.15
```

Fix also CPU version!

Fixing Access Pattern

Run Result II

```
$ 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:    6.2612 s, This:    0.2187 s, speedup:    28.63
```

Fixing Access Pattern

Run Result II

```
$ 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:    6.2612 s, This:    0.2187 s, speedup:    28.63
```

26 × is great!

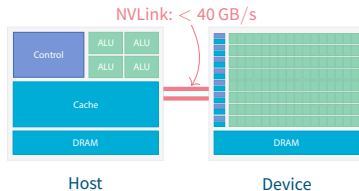
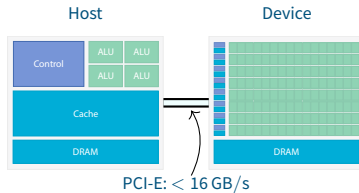
Aside: Data Transfer with NVLink

- One feature of Minsky not showcased in tutorial: NVLink between CPU and GPU
- Task 3 on P100 + PCI-E:

```
$ nvprof ./poisson2d
2048x2048: Ref: 73.1076 s, This: 0.4600 s, speedup: 158.93
Device "Tesla P100-PCI-E-12GB (0)"
Count Avg Size Min Size Max Size Total Size Total Time Name
657 149.63KB 4.0000KB 0.9844MB 96.00000MB 9.050452ms Host To Device
193 169.78KB 4.0000KB 0.9961MB 32.00000MB 2.679974ms Device To Host
```

- Task 3 on P100 + NVLink:

```
2048x2048: Ref: 49.7252 s, This: 0.5574 s, speedup: 89.21
Device "Tesla P100-SXM2-16GB (0)"
Count Avg Size Min Size Max Size Total Size Total Time Name
480 204.80KB 64.0000KB 960.00KB 96.00000MB 3.325184ms Host To Device
160 204.80KB 64.0000KB 960.00KB 32.00000MB 1.102954ms Device To Host
```



Fixing Access Pattern

Run Result II

```
$ 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:    6.8080 s, This:    0.2609 s, speedup:    26.10
```

Fixing Access Pattern

Run Result II

```
$ 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:    6.8080 s, This:    0.2609 s, speedup:    26.10
```

26 × is great!

Page-Locked Memory

Pageability

- Host memory allocated with `malloc()` is pageable
 - Memory pages of memory can be moved by kernel, e.g. swapped to disk
 - Additional indirection



Page-Locked Memory

Pageability

- Host memory allocated with `malloc()` is pageable
 - Memory pages of memory can be moved by kernel, e.g. swapped to disk
 - Additional indirection
- NVIDIA GPUs can allocate **page-locked memory** (*pinned* memory)
 - + Faster (safety guards are skipped)
 - + Interleaving of execution and copy (asynchronous)
 - + Directly map into GPU memory*
 - Scarce resource; OS performance could degrade



Page-Locked Memory

Pageability

- Host memory allocated with `malloc()` is pageable
 - Memory pages of memory can be moved by kernel, e.g. swapped to disk
 - Additional indirection
- NVIDIA GPUs can allocate **page-locked memory** (*pinned* memory)
 - + Faster (safety guards are skipped)
 - + Interleaving of execution and copy (asynchronous)
 - + Directly map into GPU memory*
 - Scarce resource; OS performance could degrade
- OpenACC: Very easy to use pinned memory
 - ta=tesla:pinned

Page-Locked Memory

Loop change

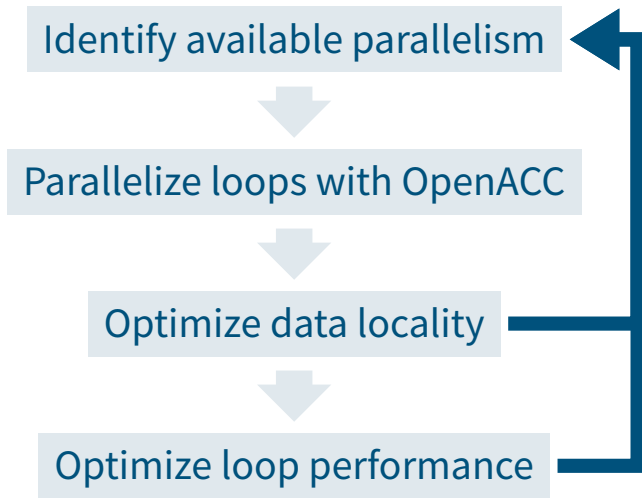
TASK 7'

- Compare performance with and without pinned memory
- Also test unified memory again

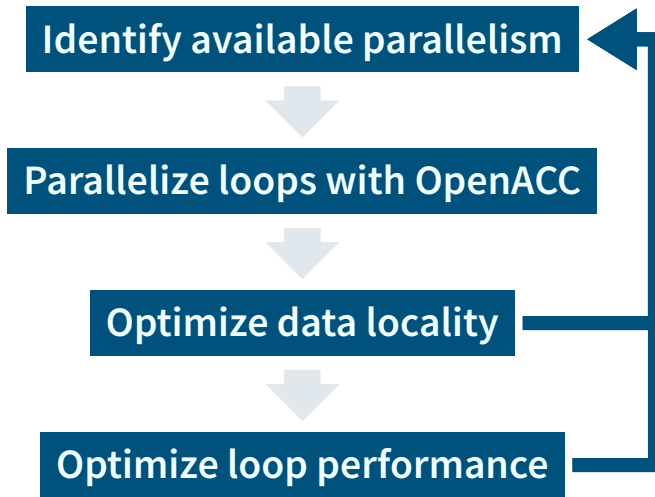
Task 7': Pinned Memory

- Like in Task 7, but change compilation to include pinned or managed
- Submit to the batch system: `make run`

Parallelization Workflow



Parallelization Workflow



Conclusions & Summary

- **OpenACC** can be used to efficiently exploit parallelism
 - ... on the CPU, similar to OpenMP,
 - ... on the GPU, for which it is specially designed for,
 - ... on multiple GPUs, working well together with MPI (not shown today).
 - It can work well with other GPU-leveraging tools
 - There are still many more tuning possibilities and keywords not mentioned (*time...*)
- Great online resources to **deepen your knowledge** (see appendix)

Conclusions & Summary

- **OpenACC** can be used to efficiently exploit parallelism
 - ... on the CPU, similar to OpenMP,
 - ... on the GPU, for which it is specially designed for,
 - ... on multiple GPUs, working well together with MPI (not shown today).
 - It can work well with other GPU-leveraging tools
 - There are still many more tuning possibilities and keywords not mentioned here (see e.g. [1] for more details...)
- Great online resources to **deepen your knowledge** (see e.g. [1] for more details...)

**Thank you
for your attention!**
a.herten@fz-juelich.de

APPENDIX

Appendix

List of Tasks

Further Reading

Glossary

References

List of Tasks

Task 2: A First Parallel Loop

Task 3: More Parallel Loops on GPU

Task 4: Data Copies

Task 5: Data Region

Task 6: More Parallelism

Task 7: Loop Ordering

Task 7': Pinned Memory

Further Reading

Further Resources on OpenACC

- www.openacc.org: Official home page of OpenACC
- developer.nvidia.com/openacc-courses: OpenACC courses, upcoming (live) and past (recorded)
- <https://nvidia.qwiklab.com/quests/3>: Qwiklabs for OpenACC; various levels
- Book: **Chandrasekaran and Juckeland** *OpenACC for Programmers: Concepts and Strategies* <https://www.amazon.com/OpenACC-Programmers-Strategies-Sunita-Chandrasekaran/dp/0134694287> [3]
- Book: **Farber** *Parallel Programming with OpenACC* <https://www.amazon.com/Parallel-Programming-OpenACC-Rob-Farber/dp/0124103979> [4]

Glossary

Glossary I

CUDA Computing platform for **GPUs** from NVIDIA. Provides, among others, CUDA C/C++. [18](#), [67](#), [68](#), [69](#), [70](#), [104](#)

GCC The GNU Compiler Collection, the collection of open source compilers, among others for C and Fortran. [17](#), [20](#)

MPI The Message Passing Interface, a API definition for multi-node computing. [129](#), [130](#)

NVIDIA US technology company creating **GPUs**. [11](#), [66](#), [67](#), [68](#), [69](#), [70](#), [71](#), [72](#), [79](#), [123](#), [124](#), [125](#), [137](#), [138](#), [139](#)

NVLink **NVIDIA**'s communication protocol connecting **CPU** ↔ **GPU** and **GPU** ↔ **GPU** with high bandwidth. [120](#), [138](#)

Glossary II

- OpenACC** Directive-based programming, primarily for many-core machines. [2](#), [3](#), [4](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#), [16](#), [18](#), [19](#), [20](#), [21](#), [22](#), [32](#), [39](#), [40](#), [42](#), [44](#), [46](#), [51](#), [57](#), [58](#), [59](#), [60](#), [61](#), [71](#), [72](#), [73](#), [74](#), [81](#), [88](#), [91](#), [97](#), [100](#), [104](#), [105](#), [123](#), [124](#), [125](#), [127](#), [128](#), [129](#), [130](#), [135](#)
- OpenMP** Directive-based programming, primarily for multi-threaded machines. [12](#), [13](#), [14](#), [46](#), [50](#), [53](#), [54](#), [55](#), [129](#), [130](#)
- P100** A large GPU with the Pascal architecture from NVIDIA. It employs NVLink as its interconnect and has fast HBM2 memory. [120](#)
- PAPI** The Performance API, a C/C++ API for querying performance counters. [33](#), [34](#)
- Pascal** GPU architecture from NVIDIA (announced 2016). [67](#), [68](#), [69](#), [70](#), [138](#)
- perf** Part of the Linux kernel which facilitates access to performance counters; comes with command line utilities. [33](#), [34](#)

Glossary III

PGI Compiler creators. Formerly *The Portland Group, Inc.*; since 2013 part of **NVIDIA**.
17, 20, 33, 34, 50

POWER CPU architecture from IBM, earlier: PowerPC. See also POWER8. 139

POWER8 Version 8 of IBM's **POWER** processor, available also under the OpenPOWER Foundation. 139

CPU Central Processing Unit. 2, 3, 4, 17, 46, 67, 68, 69, 70, 114, 117, 120, 129, 130, 137, 139

GPU Graphics Processing Unit. 2, 3, 4, 6, 7, 8, 9, 10, 17, 19, 57, 58, 59, 60, 61, 66, 67, 68, 69, 70, 71, 72, 89, 120, 123, 124, 125, 129, 130, 137, 138

References

References I

- [2] 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](https://doi.org/10.1145/356635.356640). URL: <http://doi.acm.org/10.1145/356635.356640> (pages 33, 34).
- [3] Sunita Chandrasekaran and Guido Juckeland. *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley Professional, 2017. ISBN: 0134694287. URL: <https://www.amazon.com/OpenACC-Programmers-Strategies-Sunita-Chandrasekaran/dp/0134694287> (page 135).
- [4] Rob Farber. *Parallel Programming with OpenACC*. Morgan Kaufmann, 2016. ISBN: 0124103979. URL: <https://www.amazon.com/Parallel-Programming-OpenACC-Rob-Farber/dp/0124103979> (page 135).

References: Images, Graphics I

- [1] SpaceX. *SpaceX Launch*. Freely available at Unsplash. URL: <https://unsplash.com/photos/uj3hvdfQujI>.