INTRODUCTION TO OPENACC

JSC OPENACC COURSE 2018

29 October 2018 | Andreas Herten | Forschungszentrum Jülich

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Outline

- OpenACC by Example
  - OpenACC Workflow
  - Identify Parallelism
  - Parallelize Loops
    - `parallel`
    - `loops`
    - `pgprof`
    - `kernels`
  - Data Transfers
    - GPU Memory Spaces
    - Portability
    - Clause: `copy`
    - Visual Profiler
  - Data Locality
    - Analyse Flow
    - `data`
    - `enter data`

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

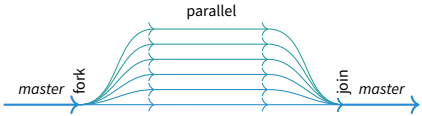
- Compiler: PGI, GCC, Cray, *Sunway*
- Languages: C/C++, Fortran

Open{MP↔ACC}

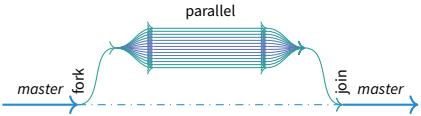
Everything's connected

- OpenACC modeled after OpenMP …
- … but specific for accelerators
- By now, OpenMP 4.0/4.5 has also *offloading* feature; but compiler support slow (Clang, XL)
- OpenACC more descriptive, OpenMP more prescriptive
- Basic principle same: Fork/join model

  Master thread launches parallel child threads; merge after execution



OpenMP



OpenACC

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

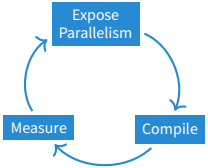
Simple and abstracted

- 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 $success

Iteration is key



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

⇒ **Productivity**

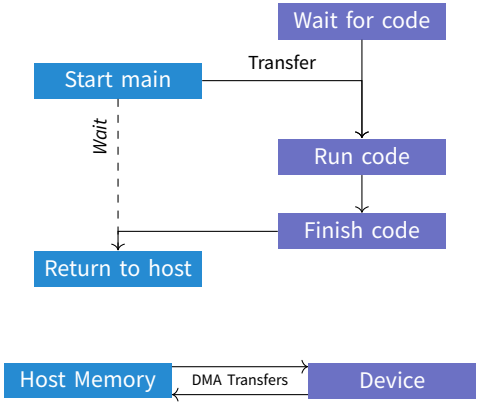
- Because of *generalness*: Sometimes not last bit of hardware performance accessible
- But: Use OpenACC together with other accelerator-targeting techniques (CUDA, libraries, …)

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:cc37`	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];
    }
}
```

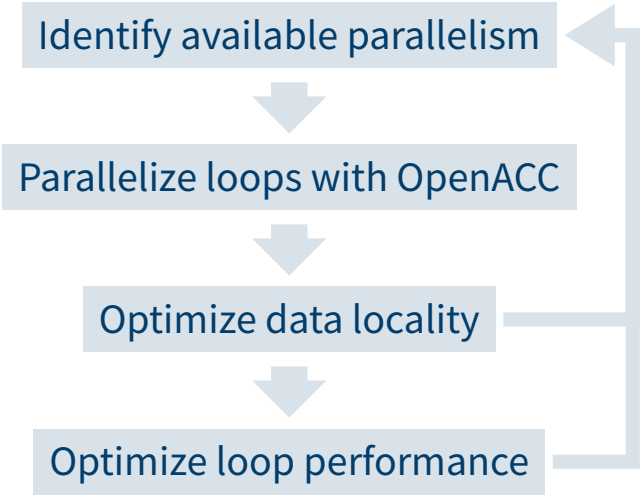
```
!$acc data copy(x(1:N),y(1:N))
!$acc parallel loop

    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

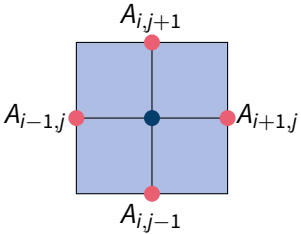
Parallelization Workflow

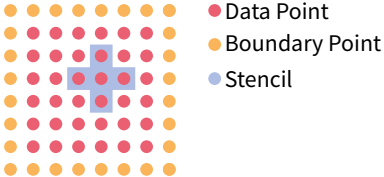


Jacobi Solver

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++;
}
```

Iterate until converged

Iterate across matrix elements

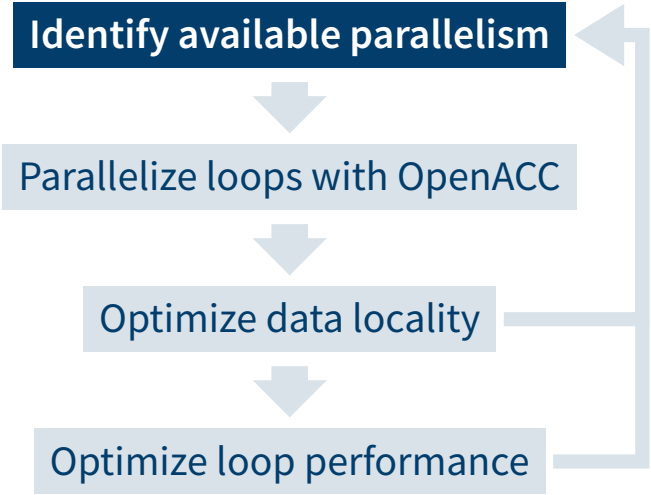
Calculate new value from neighbors

Accumulate error

Swap input/output

Set boundary conditions

Parallelization Workflow



Profiling

Profile

> *[…] premature optimization is the root of all evil.*
> ***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 Visual Profiler, …
- Here: Examples from PGI

Identify Parallelism

TASK 1

Generate Profile

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

Task 1: Analyze Application

- 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 `pgprof` call; try to understand

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

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
  23.53%  327.11ms  main (./poisson2d.c:128 0xfcb)
  22.79%  316.89ms  main (./poisson2d.c:128 0xfc1)
  21.32%  296.45ms  main (./poisson2d.c:128 0xfc5)
  15.44%  214.67ms  main (./poisson2d.c:128 0xfde)
======== 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++;
}
```

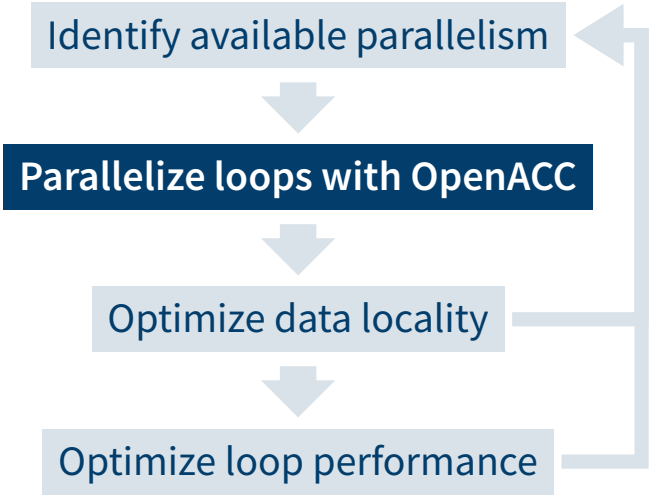
Data dependency between iterations

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
  $\rightarrow$ 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` C

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

Parallel Loops: Parallel

Maybe the second most important directive

- Programmer identifies block containing parallelism
  $\rightarrow$ 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` 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

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` C

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

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` 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`

```
#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
```

Kernel 1

Kernel 2

Parallel Jacobi

TASK 2

Add parallelism

- Add OpenACC parallelism to main double loop in Jacobi solver source code
- Profile 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
- Profile: `make profile`
  `pgprof` or `nvprof` is prefix to call to `poisson2d`

Parallel Jacobi

Source Code

```
#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]));
    }
}
```

Parallel Jacobi

Compilation result

```
$ make
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60,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, Generating implicit copyin(A[:],rhs[:])
         Generating implicit copyout(Anew[:])
    112, 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
PGI_ACC_POOL_ALLOC=0 srun ./poisson2d
Job <38143> is submitted to default queue <normal.i>.
<<Waiting for dispatch ...>>
<<Starting on juronc11>>
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:  70.6675 s, This:  10.1112 s, speedup:     6.9
```

pgprof / nvprof

NVIDIA's command line profiler

- Profiles applications, mainly for NVIDIA GPUs, but also CPU code
- More: This afternoon / tomorrow morning

Profile of Jacobi

With pgprof

```
$ make profile
==116606== PGPROF is profiling process 116606, command: ./poisson2d 10
==116606== Profiling application: ./poisson2d 10
Jacobi relaxation calculation: max 10 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution.
2048x2048: Ref:   0.8378 s, This:   0.2716 s, speedup:     3.08
==116606== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.96%  129.82ms        10  12.982ms  11.204ms  20.086ms  main_109_gpu
  0.02%  30.560us        10  3.0560us  2.6240us  3.8720us  main_109_gpu_red
  0.01%  10.304us        10  1.0300us     960ns  1.2480us  [CUDA memcpy HtoD]
  0.00%  6.3680us        10     636ns     608ns     672ns  [CUDA memcpy DtoH]

==116606== Unified Memory profiling result:
Device "Tesla P100-SXM2-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    3360  204.80KB  64.000KB  960.00KB  672.0000MB  25.37254ms  Host To Device
    3200  204.80KB  64.000KB  960.00KB  640.0000MB  30.94435ms  Device To Host
    2454         -         -         -           -  66.99111ms  GPU Page fault groups
Total CPU Page faults: 2304
```

Profile of Jacobi

With pgprof

```
$ make profile
==116606== PGPROF is profiling process 116606, command: ./poisson2d 10
==116606== Profiling application: ./poisson2d 10
Jacobi relaxation calculation: max 10 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial CPU execution
2048x2048: Ref:   0.83
==116606== Profiling r
Time(%)      Time
 99.96%  129.82ms
  0.02%  30.560us
  0.01%  10.304us
  0.00%  6.3680us

==116606== Unified Memory profiling result:
Device "Tesla P100-SXM2-16GB (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    3360  204.80KB  64.000KB  960.00KB  672.0000MB  25.37254ms  Host To Device
    3200  204.80KB  64.000KB  960.00KB  640.0000MB  30.94435ms  Device To Host
    2454         -         -         -           -  66.99111ms  GPU Page fault groups
Total CPU Page faults: 2304
```

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 created here

`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 braunching 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
pgcc -c -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60,managed
 poisson2d_reference.c -o poisson2d_reference.o
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
PGI_ACC_POOL_ALLOC=0 srun --pty ./poisson2d
Job <38144> is submitted to default queue <normal.i>.
<<Waiting for dispatch ...>>
<<Starting on juronc15>>
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:  67.8660 s, This:   5.6739 s, speedup:    11.96
```

Parallel Jacobi II

Run result

```
$ make run
PGI_ACC_POOL_ALLOC=0 srun --pty ./poisson2d
Job <38144> is submitted to default queue <normal.i>.
<<Waiting for dispatch ...>>
<<Starting on juronc15>>
Jacobi relaxation calculation: max 500                    048 x 2048 mesh
Calculate reference solution and time w                   execution.
    0, 0.249999
  100, 0.249760
  200, 0...
Calculate current execution.
    0, 0.249999
  100, 0.249760
  200, 0...
2048x2048: Ref:  67.8660 s, This:   5.6739 s, speedup:    11.96
```

Done?!

OpenACC by Example

Data Transfers

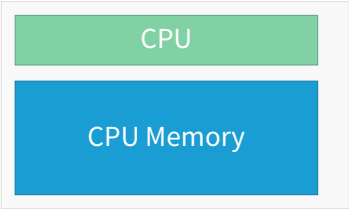
Automatic Data Transfers

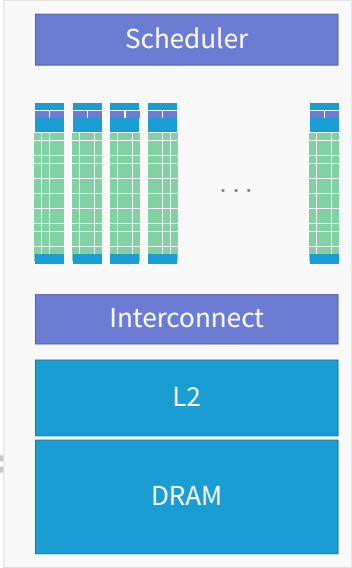
- Up to now: We did not care about **data transfers**
- Compiler and runtime care
- Magic keyword: `-ta=tesla:managed`
- Only feature of (recent) NVIDIA GPUs! (And on K80 *limited* implementation.)

CPU and GPU Memory

Location, location, location

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

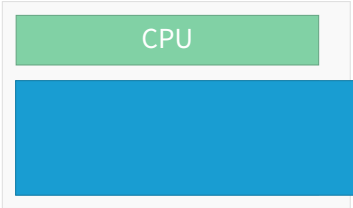


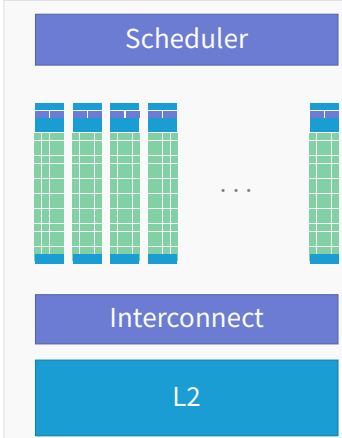


CPU and GPU Memory

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
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: Omit page faults





Portability

Only C

- 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!
- Now: Problem! We need to give that information!
  (Fortran: can often be determined by compiler)

OpenACC: `copy` C

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

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

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!
  (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!

TASK 4

- Add `copy` clause to parallel regions

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:cc60 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 --pty ./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:  65.1643 s, This:  38.9384 s, speedup:     1.67
```

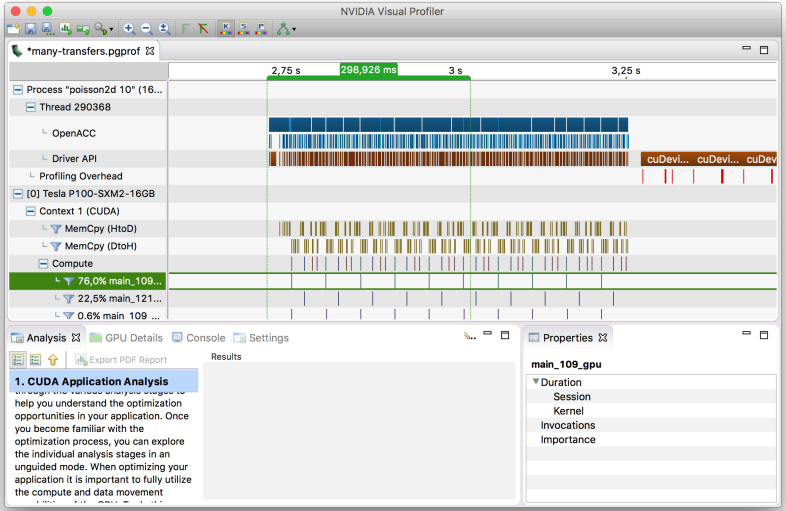
Slower?! Why?

PGI/NVIDIA Visual Profiler

- GUI tool accompanying `pgprof` / `nvprof`
- Timeline view of all things GPU
  $\rightarrow$ Study stages and interplay of application
- Dedicated session after this, now: just showing!

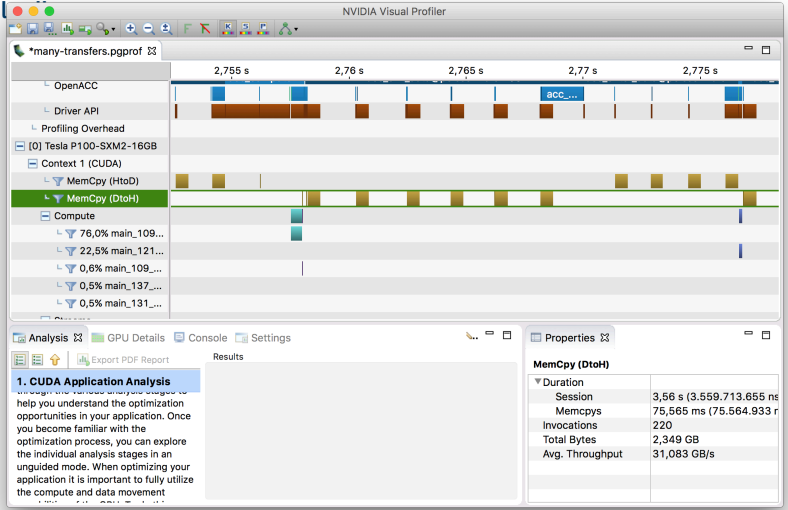
PGI/NVIDIA Visual Profiler

Overview

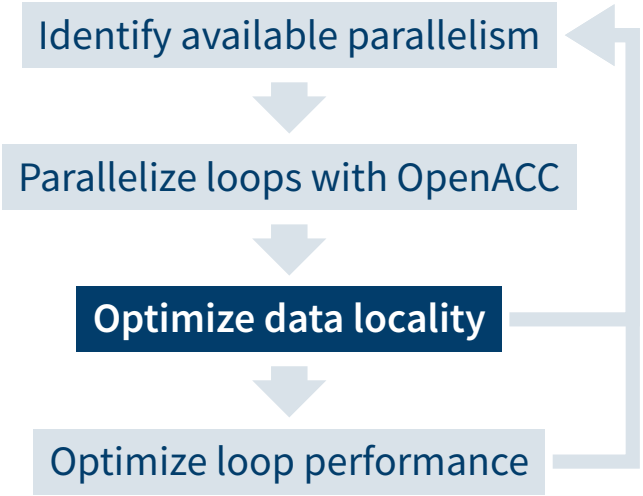


PGI/NVIDIA Visual Profiler

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**

copy

```
#pragma acc parallel loop

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 **device**

A, Anew resident on **host**

```
    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 **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, coped by 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

To manually specify data locations

- 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

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

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 TODOs
- Compile: `make`
- Submit parallel run to the batch system: `make run`

? What's your speed-up?

Parallel Jacobi II

Source Code

```
#pragma acc data copy(A[0:nx*ny]) copyin(rhs[0:nx*ny]) create(Anew[0:nx*ny])
while ( error > tol && iter < iter_max )
{
    error = 0.0;

    // Jacobi kernel
    #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]));
        }
    }

    // A <-> Anew
    #pragma acc parallel loop
    for (int iy = iy_start; iy < iy_end; iy++)
    // …
}
```

Data Region

Compiler Output

```
$ make
pgcc -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla:cc60 poisson2d.c
poisson2d_reference.o -o poisson2d
poisson2d.c:
main:
    104, Generating copyin(rhs[:ny*nx])
         Generating create(Anew[:ny*nx])
         Generating copy(A[:ny*nx])
    110, Accelerator kernel generated
         Generating Tesla code
        110, Generating reduction(max:error)
        111, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
        113, #pragma acc loop seq
        ...
```

Data Region

Run Result

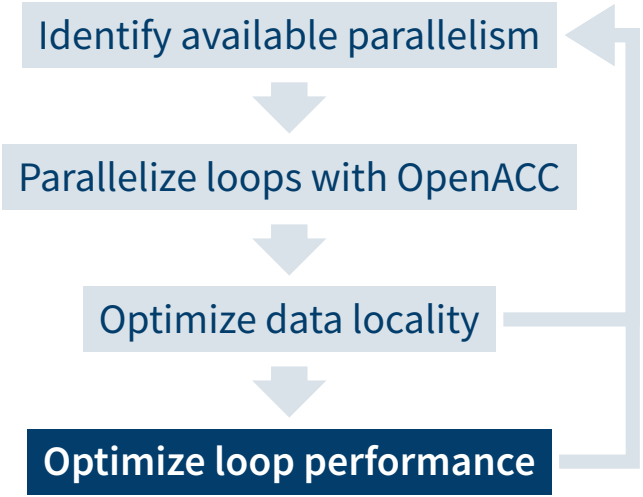
Fortran

```
$ make run
srun --pty ./poisson2d
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution and time wi                    J execution.
    0, 0.249999
  100, 0.249760
  200, 0...
Calculate current execution.
    0, 0.249999
  100, 0.249760
  200, 0...
2048x2048: Ref:  65.8519 s, This:   1.0166 s, speedup:    64.77
```

Nice!

$ make run

Parallelization Workflow



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

Thank you for your attention!

a.herten@fz-juelich.de

Appendix

List of Tasks

Glossary I

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 8, 48, 49

GCC The GNU Compiler Collection, the collection of open source compilers, among others for C and Fortran. 7, 10

NVIDIA US technology company creating GPUs. 3, 35, 47, 50, 56, 57, 58, 76, 77

OpenACC Directive-based programming, primarily for many-core machines. 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 16, 22, 23, 24, 26, 27, 29, 31, 38, 41, 46, 50, 51, 52, 59, 63, 66, 71, 73

OpenMP Directive-based programming, primarily for multi-threaded machines. 2, 4, 40

PAPI The Performance API, a C/C++ API for querying performance counters. 17

Glossary II

Pascal GPU architecture from NVIDIA (announced 2016). 48, 49

perf Part of the Linux kernel which facilitates access to performance counters; comes with command line utilities. 17

PGI Compiler creators. Formerly *The Portland Group, Inc.*; since 2013 part of NVIDIA. 7, 10, 17

CPU Central Processing Unit. 7, 35, 48, 49

GPU Graphics Processing Unit. 2, 7, 9, 31, 35, 47, 48, 49, 50, 56, 62, 64, 73, 76, 77

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 (page 17).

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.