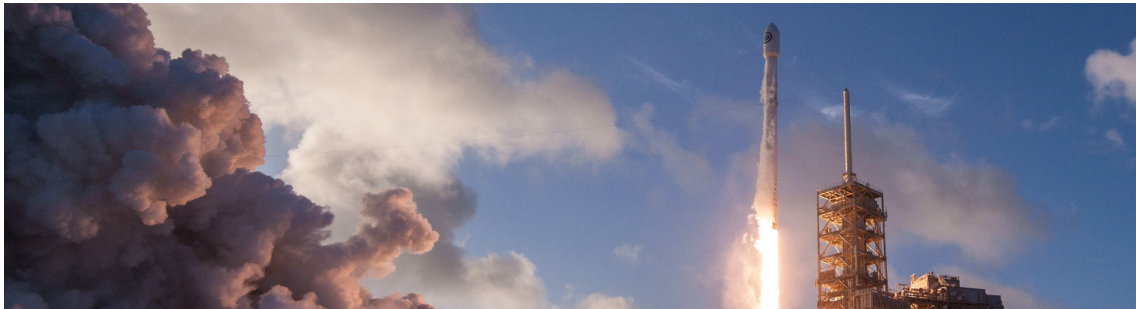




OPENACC TUTORIAL GRIDKA SCHOOL 2019

29 August 2019 | Andreas Herten | Forschungszentrum Jülich

Outline

The GPU Platform

- Introduction

- Threading Model

- App Showcase

- Parallel Models

OpenACC

- History

- OpenMP

- Modus Operandi

- OpenACC's Models

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`

- Optimize

 - Levels of Parallelism

 - Clause: `gang`

 - Memory Coalescing

 - Pinned

Interoperability

- The Keyword

- Tasks

 - Task 1

 - Task 2

 - Task 3

 - Task 4

- Conclusions

- List of Tasks

Now: Download and install
PGI Community Edition

The GPU Platform

CPU vs. GPU

A matter of specialties



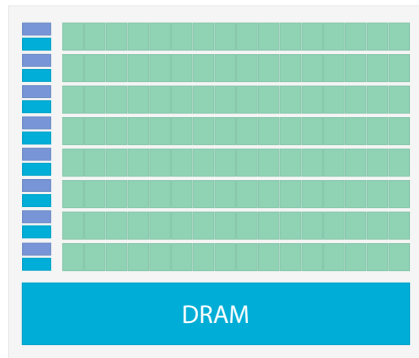
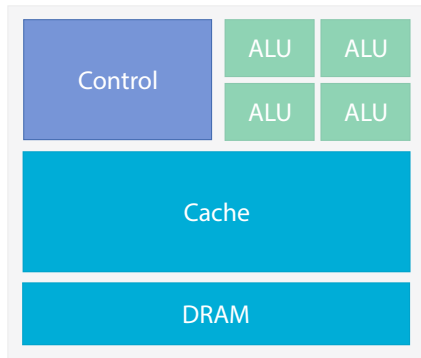
Transporting one



Transporting many

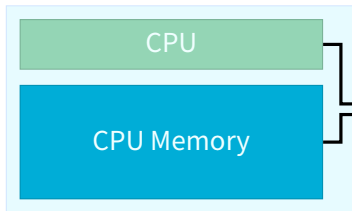
CPU vs. GPU

Chip

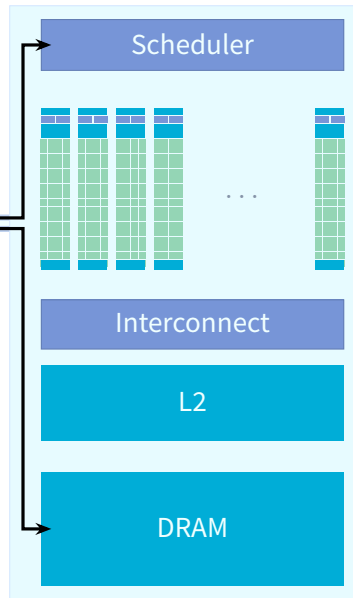


Processing Flow

CPU → GPU → CPU

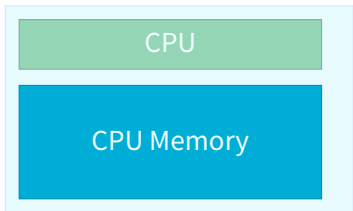


- 1 Transfer data from CPU memory to GPU memory, transfer program
- 2 Load GPU program, execute on SMs, get (cached) data from memory; write back

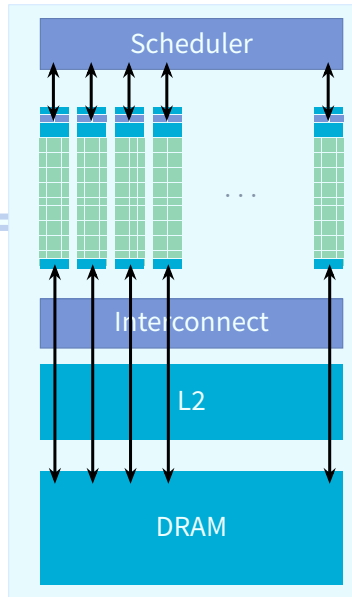


Processing Flow

CPU → GPU → CPU

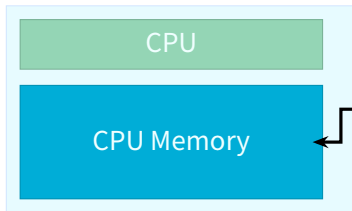


- 1 Transfer data from CPU memory to GPU memory, transfer program
- 2 Load GPU program, execute on SMs, get (cached) data from memory; write back

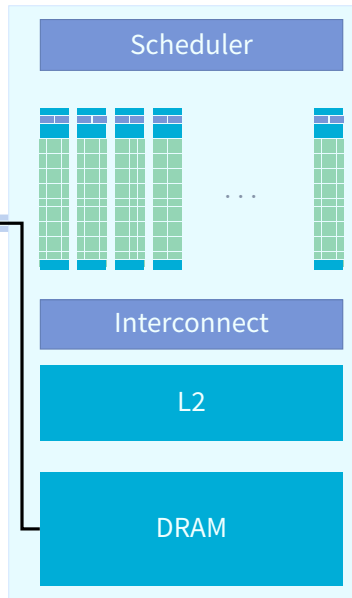


Processing Flow

CPU → GPU → CPU

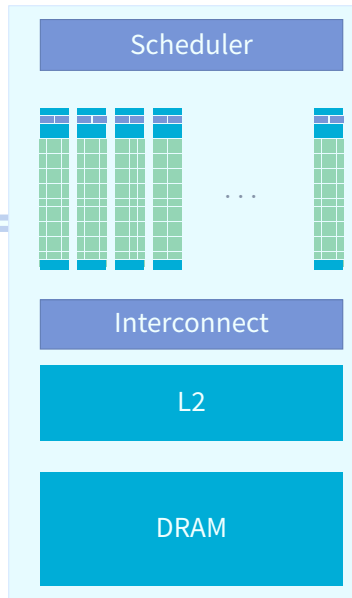
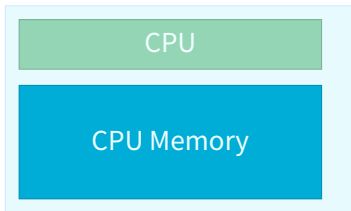


- 1 Transfer data from CPU memory to GPU memory, transfer program
- 2 Load GPU program, execute on SMs, get (cached) data from memory; write back
- 3 Transfer results back to host memory



Processing Flow

CPU → GPU → CPU



- 1 Transfer data from CPU memory to GPU memory, transfer program
 - 2 Load GPU program, execute on SMs, get (cached) data from memory; write back
 - 3 Transfer results back to host memory
- Old: Manual data transfer invocations – UVA
 - New: Driver automatically transfers data – UM

CUDA Threading Model

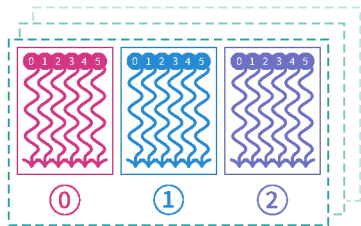
Warp the kernel, it's a thread!

- Methods to exploit parallelism:

- Thread → Block

- Block → Grid

- Threads & blocks in 3D



- Execution entity: **threads**

- Lightweight → fast switching!

- 1000s threads execute simultaneously → order non-deterministic!

- Parallel function: **kernel**

- Create **account** in Jülich via JuDoor
 - <https://judoor.fz-juelich.de/projects/join/training1926>
 - Join project1926
 - Accept *Usage Agreement*
 - Upload SSH Key
- **Login** to JURON
 - Either via <https://jupyter-jsc.fz-juelich.de/> and JuDoor web login credentials
 - or via ssh `user1@juron.fz-juelich.de` *Recommended!*
- **Bootstrap** environment

```
source $PROJECT_training1926/bootstrap.sh
```

Getting GPU-Acquainted

Some Applications

TASK 0

GEMM

N-Body

Task 0: Getting Started

- Change to GPU/Tasks/Task0/ directory
- Read Instructions.rst

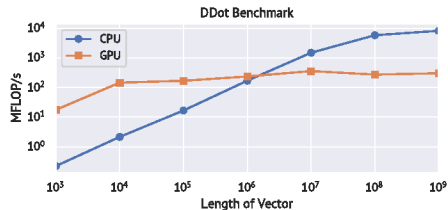
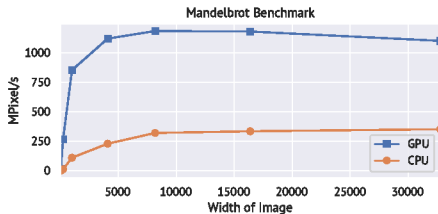
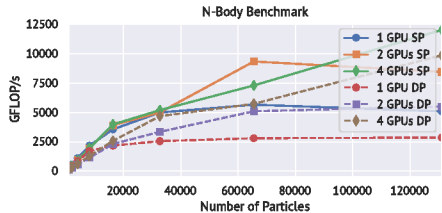
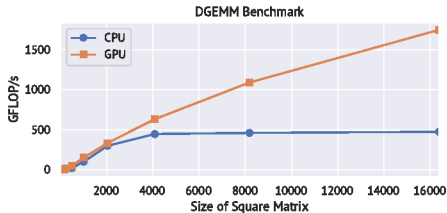
Mandelbrot

Dot Product

Getting GPU-Acquainted

Some Applications

TASK 0



Primer on Parallel Scaling

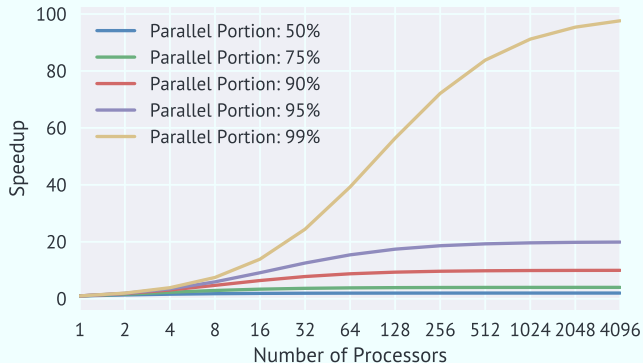
Amdahl's Law

Possible maximum speedup for
 N parallel processors

Total Time $t = t_{\text{serial}} + t_{\text{parallel}}$

N Processors $t(N) = t_s + t_p/N$

Speedup $s(N) = t/t(N) = \frac{t_s + t_p}{t_s + t_p/N}$

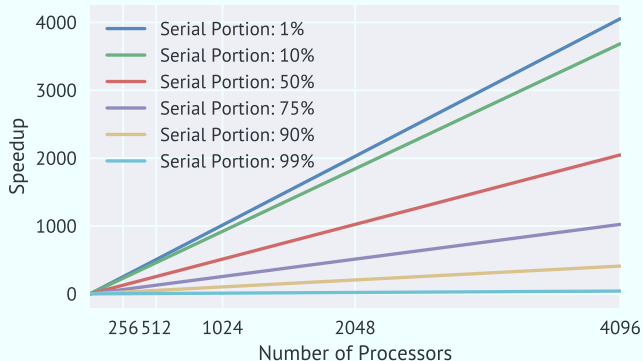


Primer on Parallel Scaling II

Gustafson-Barsis's Law

[...] speedup should be measured by scaling the problem to the number of processors, not fixing problem size.

– John Gustafson



Parallelism

Parallel programming is not easy!

Things to consider:

- Is my application **computationally intensive** *enough*?
- What are the levels of **parallelism**?
- How much **data** needs to be **transferred**?
- Is the **gain** worth the **pain**?

Possibilities

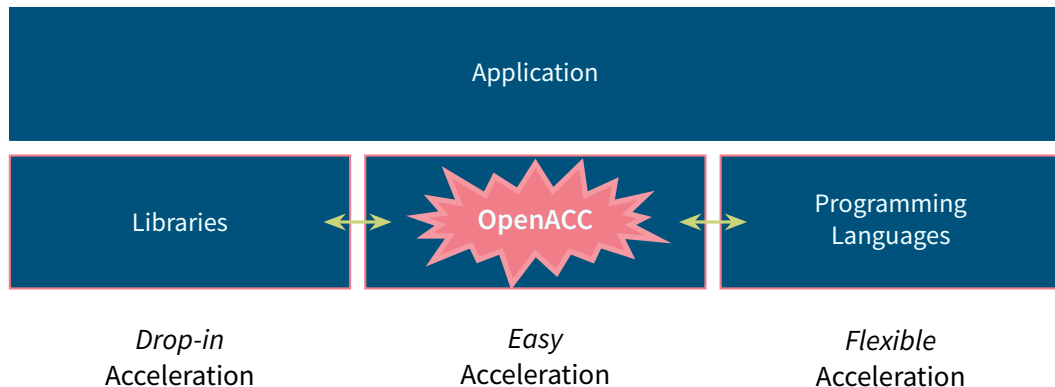
Different levels of *closeness* to GPU when GPU-programming, which **can** ease the *pain*...

- OpenACC, OpenMP
- Thrust, Kokkos, SYCL
- PyCUDA, Cupy, Numba

Other alternatives (for completeness)

- CUDA Fortran
- HIP
- OpenCL

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, ... 

2018 OpenACC 2.7: Clarifications, more host, ...  

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

Support

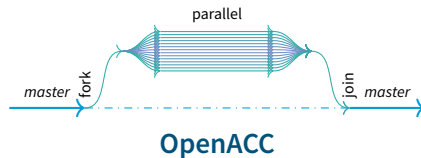
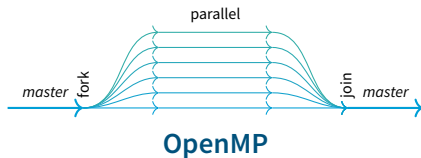
- Compiler: PGI, GCC, Clang, *Sunway*
- Languages: C/C++, Fortran

Open{MP ↔ ACC}

Everything's connected

- OpenACC modeled after OpenMP ...
- ... but specific for accelerators
- 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 *Actively performance-improved, OSS*
 - Clang *First alpha version*
- 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*

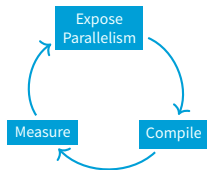
3 \$uccess

Iteration is key

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

⇒ Productivity

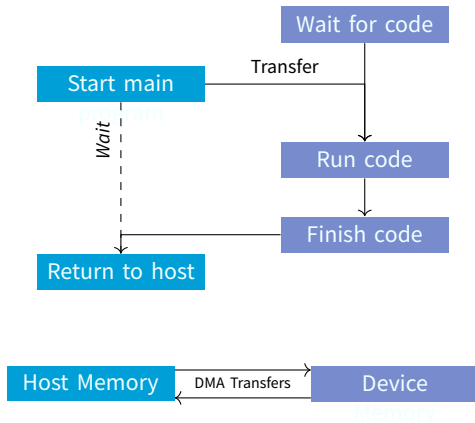
- Because of *generallness*: Sometimes not last bit of hardware performance accessible
- But: Use OpenACC together with other accelerator-targeting techniques (CUDA, libraries, ...)



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:cc70` Use compute capability 7.0

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

`-ta=tesla:managed` Use unified memory

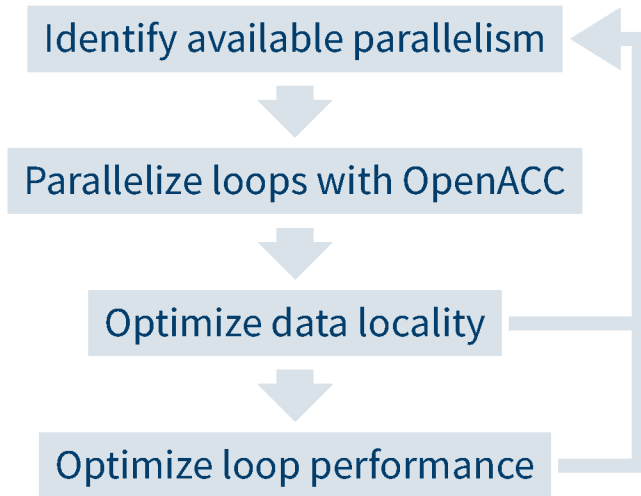
`-fopenacc-dim=geom` Use *geom* configuration for threads

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];
    }
}
```

OpenACC by Example

Parallelization Workflow

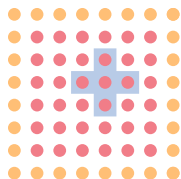
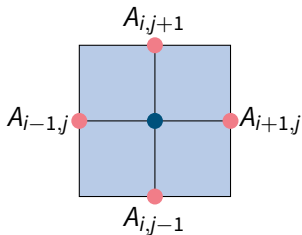


→ Conclusion

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++;  
}
```

Iterate until converged

Iterate across
matrix elements

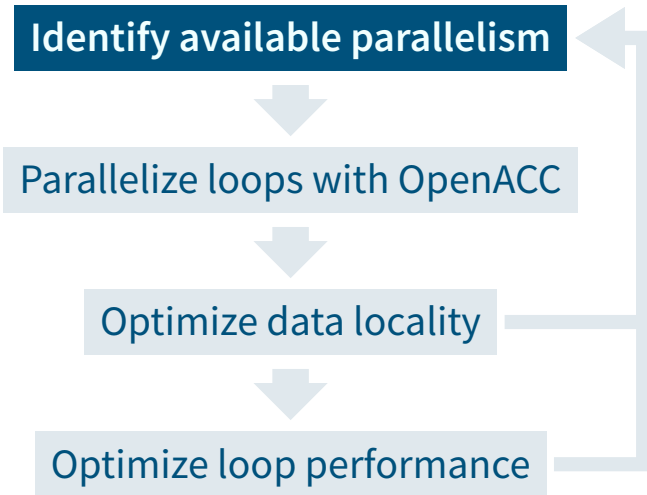
Calculate new value
from neighbors

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 [6]

- 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

Generate Profile

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

TASK 1

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 bsub 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
26.62%       370ms  main (poisson2d.c:128 0x5c8)
22.30%       310ms  main (poisson2d.c:116 0x554)
17.99%       250ms  main (poisson2d.c:123 0x5b8)
...
1.44%        20ms  __c_mcopy8 (0xffac0054)
1.44%        20ms  _xlmass_exp (0xffbe008c)
...
===== Data collected at 100Hz frequency
```

- Since everything is in main – limited helpfulness, hard to dissect
- 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++;  
}
```

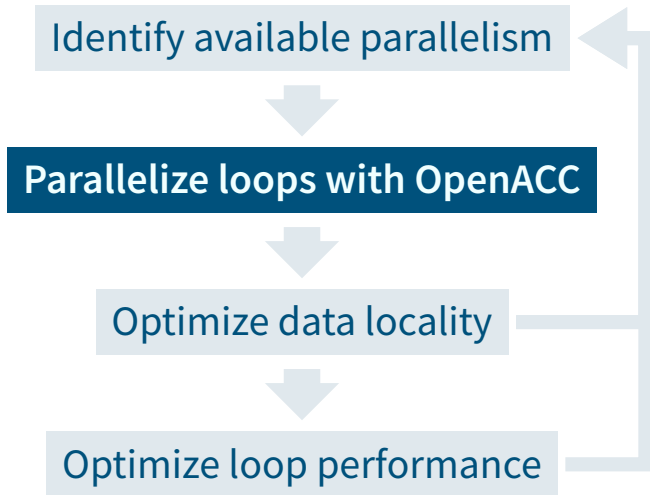
Data dependency
between iterations

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
→ 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

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 `bsub` 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

```
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=tesla:cc60,managed poisson2d.c
  poisson2d_reference.o -o poisson2d
poisson2d.c:
main:
  109, Generating Tesla code
    110, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
        Generating reduction(max:error)
    112, #pragma acc loop seq
  109, Generating implicit copyin(A[:],rhs[:])
        Generating implicit copy(error)
        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 bsub -gpu 'num=1:j_exclusive=yes' -Is -U training1926 ./poisson2d
<<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:  55.5590 s, This:  8.7508 s, speedup:    6.35
```

pgprof / nvprof

PGI's and NVIDIA's command line profiler

- Profiles applications, mainly for NVIDIA GPUs, but also CPU code
- GPU: CUDA kernels, API calls, OpenACC
- pgprof vs nvprof: Twins with other configurations
- Generate concise performance reports, full timelines; measure events and metrics (hardware counters)

⇒ Powerful tool for GPU application analysis

→ <http://docs.nvidia.com/cuda/profiler-users-guide/>

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
==163431== Profiling result:
      Type  Time(%)      Time       Calls      Avg      Min      Max   Name
GPU activities: 99.95% 98.397ms        10 9.8397ms 8.2994ms 15.225ms main_109_gpu
                0.03% 27.170us         10 2.7170us 2.5280us 4.0960us main_109_gpu__red
                0.01% 10.944us         10 1.0940us 1.0560us 1.2480us [CUDA memcpy DtoH]
                0.01% 10.816us         10 1.0810us 1.0560us 1.1200us [CUDA memset]

==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: 0.8378 s, This: 0.2716 s
```

```
==163431== Profiling result:
```

```
Type Time
```

```
GPU activities:
```

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

```
0.000000s [copy to H]
0.000000s [copy to D]
0.000000s [CUDA memset]
```

```
==116606== Unified Memory
```

```
Device "Tesla P100-SXM2"
```

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
```

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 branching into or out
- Program must not depend on order of evaluation of clauses
- At most: One if clause



Parallel Jacobi II

TASK 3

Add more parallelism

- Add OpenACC parallelism to other loops of `while` (L:123 – L:141)
- Use either `kernel`s 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];
    }
    // ...
}
```

Parallel Jacobi II

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, Generating Tesla code
    110, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
        Generating reduction(max:error)
    112, #pragma acc loop seq
  109, Generating implicit copyin(A[:])
    ...
  121, Generating Tesla code
    124, #pragma acc loop gang /* blockIdx.x */
    126, #pragma acc loop vector(128) /* threadIdx.x */
  121, Generating implicit copyin(Anew[:])
    ...
  126, Loop is parallelizable
  133, Accelerator kernel genera...
```

Parallel Jacobi II

Run result

```
$ make run
PGI_ACC_POOL_ALLOC=0 bsub -gpu 'num=1:j_exclusive=yes' -Is -U training1926 ./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: 60.2520 s, This: 0.6151 s, speedup: 97.95
```

Parallel Jacobi II

Run result

```
$ make run
PGI_ACC_POOL_ALLOC=0 bsub -gpu 'num=1:j_exclusive=yes' -Is -U training1926 ./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 for 200 iterations.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref: 60.2520 s, This: 0.6151 s, speedup: 97.95
```

Done?!

OpenACC by Example

Data Transfers

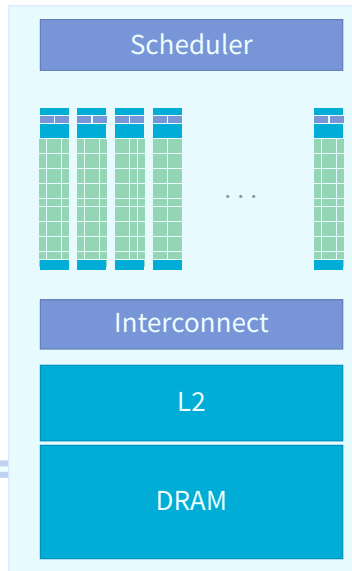
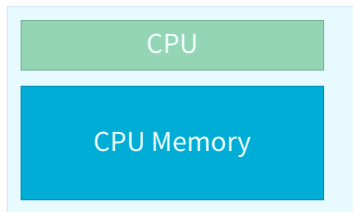
Automatic Data Transfers

- Up to now: We did not care about **data transfers**
- Compiler and runtime care
- Magic keyword: `-ta=tesla:managed`
- Be more explicit for full portability and full performance

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

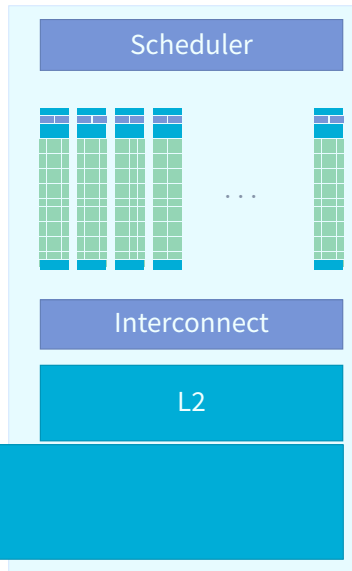
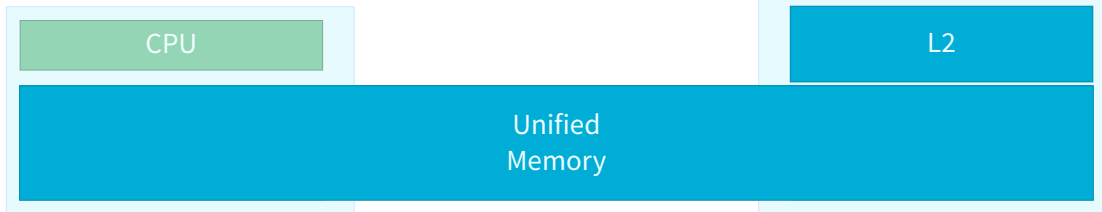
At the Beginning CPU and GPU memory very distinct, own addresses

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 (POWER); Heterogeneous Memory Management (Linux)



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:cc70
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 - rhs (poisson2d.c: 109)
...
PGC/x86-64 Linux 19.3-0: compilation aborted
```

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`
 - Generate profile with `make profile_tofile`
- ? 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
bsub -gpu 'num=1:j_exclusive=yes' -Is -U training1926 ./poisson2d
<<Starting on juronc13>>
Jacobi relaxation calculation: max 500 iterations for 2048 x 2048 mesh
Calculate reference solution and time for execution.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref: 105.5924 s, This: 28.5370 s, speedup: 3.70
```

slower?!
why?

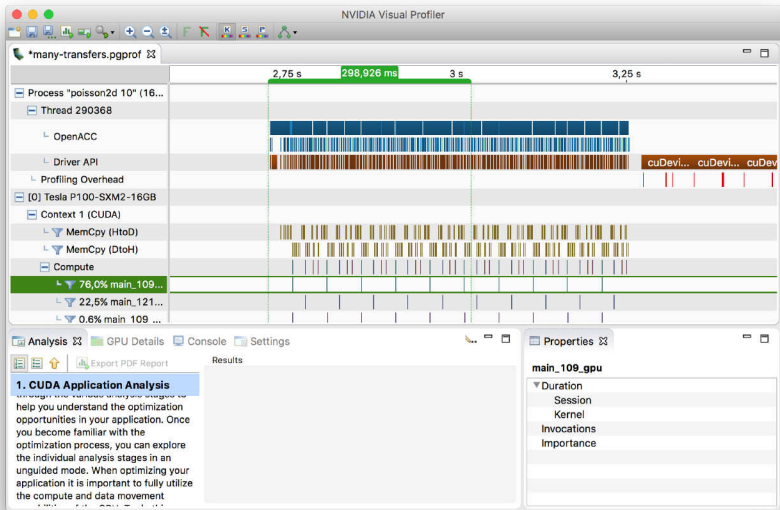
PGI/NVIDIA Visual Profiler

- GUI tool accompanying pgprof / nvprof
 - PGI Start pgprof without parameters
 - NVIDIA Start nvvp
- Timeline view of all things GPU
 - Study stages and interplay of application
- Interactive or with input from command line profilers
- View launch and run configurations
- Guided and unguided analysis

→ <https://developer.nvidia.com/nvidia-visual-profiler>

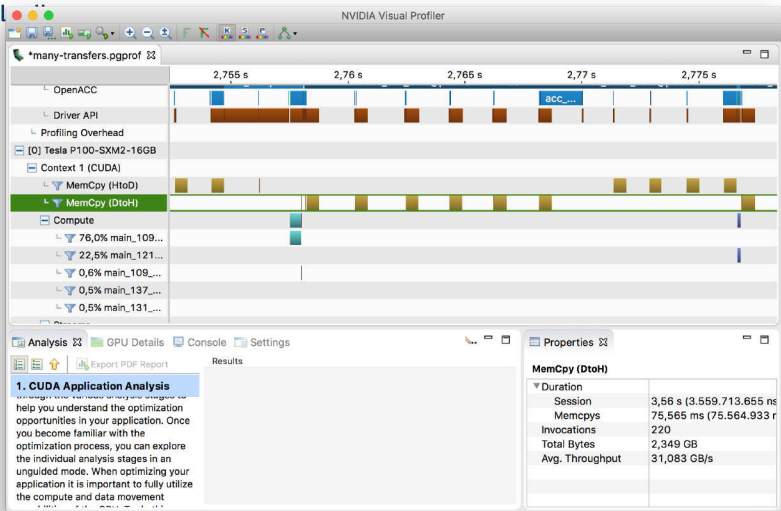
PGI/NVIDIA Visual Profiler

Overview

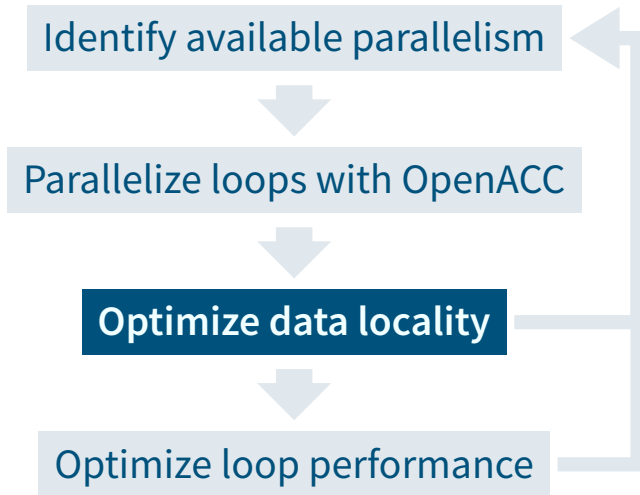


PGI/NVIDIA Visual Profiler

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

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

```
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
between **each** loop and
in **each** iteration!

A, Anew resident on host

A, Anew resident on device

iter++

}

Analyze Jacobi Data Flow

Summary

- By now, whole algorithm is using GPU
- At beginning of **while** loop, data copied to device; at end of loop, copied 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 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 }
```

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

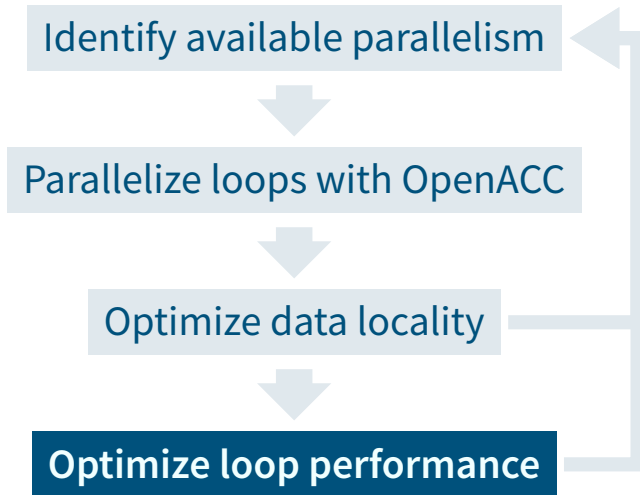
```
$ make run
<<Starting on juronc12>>
Jacobi relaxation calculation: max 500 iterations
Calculate reference solution and
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref: 104.0426 s, This:  0.5861 s, speedup:  177.53
```

Nice!
But can we be even better?

OpenACC by Example

Optimize Loop Performance

Parallelization Workflow



Understanding Compiler Output

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

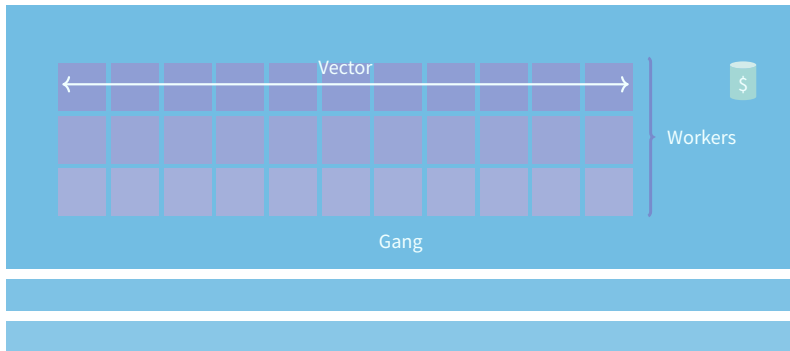
Understanding Compiler Output

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

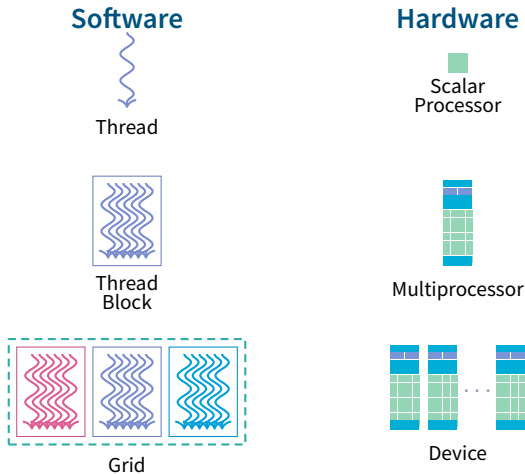
Has 1 or more vector; workers share common resource (*cache*)

Gang

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

CUDA Parallelism

CUDA Execution Model



- **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 worker vector
```

```
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

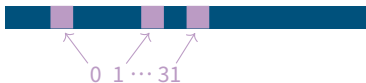
```
$ make run
srun --partition=gpus --gres=gpu:1 ./poisson2d
Jacobi relaxation calculation: max 500 iterations on 2048 mesh
Calculate reference solution and error.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref: 118.7600 s, This:  1.0367 s, speedup:  114.56
```

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 TODOs
- 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:cc60 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
<<Starting on juronc11>>
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048
Calculate reference solution and time with serial CPU version
  0, 0.249999
 100, 0.249760
 200, 0...
Calculate current
  0, 0.249999
 100, 0.249760
 200, 0...
2048x2048: Ref: 106.5172 s, This: 0.2643 s, speedup: 403.05
```

Again with proper CPU version!

Fixing Access Pattern

Run Result II

```
$ make run
<<Starting on juronc11>>
Jacobi relaxation calculation: max 500 iterations on 2048 x 2048 mesh
Calculate reference solution and time with serial execution.
    0, 0.249999
   100, 0.249760
   200, 0...
Calculate current execution.
    0, 0.249999
   100, 0.249760
   200, 0...
2048x2048: Ref:   6.3189 s, This:   0.2640 s, speedup:   23.94
```

23 × 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
- 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

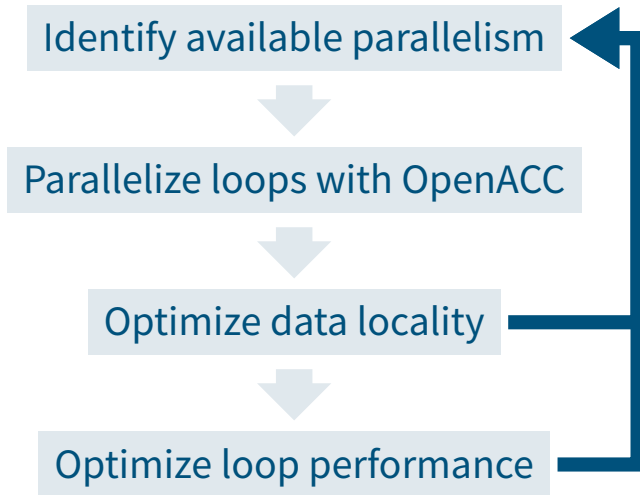
TASK 7'

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

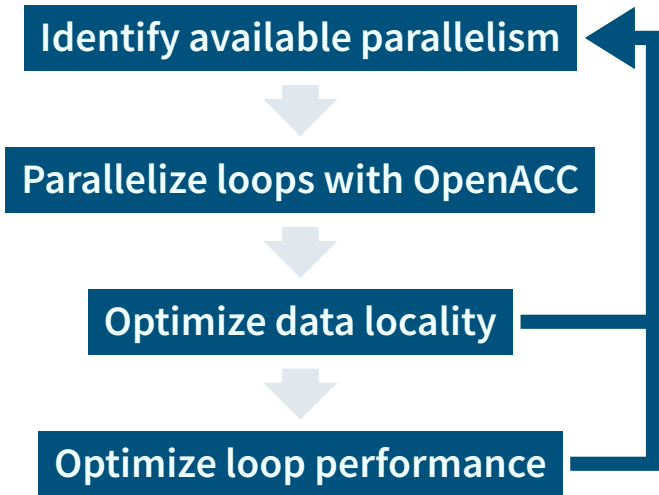
Task 7': Pinned Memory

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

Parallelization Workflow



Parallelization Workflow



Interoperability

Interoperability

- OpenACC can operate together with
 - Applications
 - Libraries
 - CUDA
- Both directions possible: Call OpenACC from others, call others from OpenACC

The Keyword

OpenACC's Rosetta Stone

host_data use_device

- Background
 - GPU and CPU are different devices, have different memory
 - Distinct address spaces
- OpenACC hides handling of addresses from user
 - For every chunk of accelerated data, **two** addresses exist
 - One for CPU data, one for GPU data
 - OpenACC uses appropriate address in accelerated kernel
- **But:** Automatic handling not working when out of OpenACC (OpenACC will default to host address)
 - **host_data use_device** uses the address of the GPU device data for scope

The host_data Construct

Example

- Usage:

```
double* foo = new double[N];           // foo on Host
#pragma acc data copyin(foo[0:N])      // foo on Device
{
    ...
    #pragma acc host_data use_device(foo)
    some_lfunc(foo);                    // Device: OK!
    ...
}
```

- Directive can be used for structured block as well

The Inverse: deviceptr

When CUDA is involved

- For the inverse case:
 - Data has been copied by CUDA or a CUDA-using library
 - Pointer to data residing on devices is returned
 - Use this data in OpenACC context

- `deviceptr` clause declares data to be on device

- Usage:

```
float * n;  
int n = 4223;  
cudaMalloc((void*)&x, (size_t)n*sizeof(float));  
// ...  
#pragma acc kernels deviceptr(x)  
for (int i = 0; i < n; i++) {  
    x[i] = i;  
}
```

Interoperability Tasks

Task 1

Introduction to BLAS

- Use case: Anything linear algebra
- **BLAS**: Basic Linear Algebra Subprograms
 - Vector-vector, vector-matrix, matrix-matrix operations
 - Specification of routines
 - Examples: SAXPY, DGEMV, ZGEMM→ <http://www.netlib.org/blas/>
- **cuBLAS**: NVIDIA's linear algebra routines with BLAS interface, readily accelerated
→ <http://docs.nvidia.com/cuda/cublas/>
- **Task 1**: Use cuBLAS for vector addition, everything else with OpenACC

Task 8-1

cuBLAS OpenACC Interaction

- cuBLAS routine used:

```
cublasDaxpy(cublasHandle_t handle, int n,  
            const double          *alpha,  
            const double          *x, int incx,  
            double                *y, int incy)
```

- handle capsules GPU auxiliary data, needs to be created and destroyed with cublasCreate and cublasDestroy
- x and y point to addresses on **device**!
- cuBLAS library needs to be linked with `-lcublas`

Task 8-1

TASK 8-1

Vector Addition with cuBLAS

- Use cuBLAS for vector addition

Task 8-1: OpenACC+cuBLAS

- Change to Task8-1/ directory
- Work on TODOs in `vecAddRed.c`
 - Use `host_data use_device` to provide correct pointer
 - Check [cuBLAS documentation](#) for details on `cublasDaxpy()`
- Compile: `make`
Make sure to have loaded CUDA before: `module load cuda`
- Submit to the batch system: `make run`

Task 8-2

CUDA Need-to-Know

- Use case:
 - Working on legacy code
 - Need the *raw* power (/flexibility) of CUDA
- CUDA need-to-knows:
 - Thread → Block → Grid
Total number of threads should map to your problem; threads are always given per block
 - A kernel is called from every thread on GPU device
Number of kernel threads: *triple chevron syntax*
`kernel<<<nBlocks, nThreads>>>(arg1, arg2, ...)`
 - Kernel: Function with `__global__` prefix
Aware of its index by global variables, e.g. `threadIdx.x`
→ <http://docs.nvidia.com/cuda/>

Task 8-2

TASK 8-2

Vector Addition with CUDA Kernel

- CUDA kernel for vector addition, rest OpenACC
- Marrying CUDA C and OpenACC:
 - All direct CUDA interaction wrapped in wrapper file `cudaWrapper.cu`, compiled with `nvcc` to object file (`-c`)
 - `vecAddRed.c` calls external function from `cudaWrapper.cu` (**extern**)

Task 8-2: OpenACC+CUDA

- Change to `Task8-2/` directory
- Work on TODOs in `vecAddRed.c` and `cublasWrapper.cu`
 - Use `host_data` `use_device` to provide correct pointer
 - Implement computation in kernel, implement call of kernel
- Compile: `make`; Submit to the batch system: `make run`

Thrust

Iterators! Iterators everywhere! 

- $\frac{\text{Thrust}}{\text{CUDA}} = \frac{\text{STL}}{\text{C++}}$
- Template library
- Based on iterators, but also works with plain C
- Data-parallel primitives (`scan()`, `sort()`, `reduce()`, ...); algorithms

→ <http://thrust.github.io/>
<http://docs.nvidia.com/cuda/thrust/>

Thrust

Code example

```
int a = 42;
int n = 10;
thrust::host_vector<float> x(n), y(n);
// fill x, y

thrust::device_vector d_x = x, d_y = y;

using namespace thrust::placeholders;
thrust::transform(d_x.begin(), d_x.end(), d_y.begin(), d_y.begin(), a * _1 + _2);

x = d_x;
```

Task 8-3

8-3

Vector Addition with Thrust

- Use Thrust for reduction, everything else of vector addition with OpenACC

Task 8-3: OpenACC+Thrust

- Change to Task8-3/ directory
- Work on TODOs in `vecAddRed.c` and `thrustWrapper.cu`
 - Use `host_data` `use_device` to provide correct pointer
 - Implement call to `thrust::reduce` using `c_ptr`
- Compile: `make`
- Submit to the batch system: `make run`

Task 8-4

Stating the Problem

- We want to solve the Poisson equation

$$\Delta\Phi(x,y) = -\rho(x,y)$$

with periodic boundary conditions in x and y

- Needed, e.g., for finding electrostatic potential Φ for a given charge distribution ρ
- Model problem

$$\begin{aligned}\rho(x,y) &= \cos(4\pi x) \sin(2\pi y) \\ (x,y) &\in [0,1)^2\end{aligned}$$

- Analytically known: $\Phi(x,y) = \Phi_0 \cos(4\pi x) \sin(2\pi y)$
- Let's solve the Poisson equation with a Fourier Transform!

Task 8-4

Introduction to Fourier Transforms

- Discrete Fourier Transform and Re-Transform:

$$\hat{f}_k = \sum_{j=0}^{N-1} f_j e^{-\frac{2\pi i k j}{N}} \Leftrightarrow f_j = \sum_{k=0}^{N-1} \hat{f}_k e^{\frac{2\pi i j k}{N}}$$

- Time for all $\hat{f}_k$: $\mathcal{O}(N^2)$
- Fast Fourier Transform: Recursively splitting $\rightarrow \mathcal{O}(N \log(N))$
- Find derivatives in Fourier space:

$$f'_j = \sum_{k=0}^{N-1} i k \hat{f}_k e^{\frac{2\pi i j k}{N}}$$

It's just multiplying by $ik!$

Task 8-4

Plan for FFT Poisson Solution

Start with charge density ρ

- 1 Fourier-transform ρ
 $\hat{\rho} \leftarrow \mathcal{F}(\rho)$
- 2 *Integrate* ρ in Fourier space twice
 $\hat{\phi} \leftarrow -\hat{\rho} / (k_x^2 + k_y^2)$
- 3 Inverse Fourier-transform $\hat{\phi}$
 $\phi \leftarrow \mathcal{F}^{-1}(\hat{\phi})$

cuFFT

OpenACC

cuFFT

Task 8-4

cuFFT

- cuFFT: NVIDIA's (Fast) Fourier Transform library
 - 1D, 2D, 3D transforms; complex and real data types
 - Asynchronous execution
 - Modeled after FFTW library (API)
 - Part of CUDA Toolkit

→ <https://developer.nvidia.com/cufft>

```
cufftDoubleComplex *src, *tgt;           // Device data!
cufftHandle plan;
// Setup 2d complex-complex trafo w/ dimensions (Nx, Ny)
cufftCreatePlan(plan, Nx, Ny, CUFFT_Z2Z);
cufftExecZ2Z(plan, src, tgt, CUFFT_FORWARD); // FFT
cufftExecZ2Z(plan, tgt, tgt, CUFFT_INVERSE); // iFFT
// Inplace trafo ^----^
cufftDestroy(plan);                      // Clean-up
```

Task 8-4

Synchronizing cuFFT

- CUDA Streams enable interleaving of computational tasks
- cuFFT uses streams for asynchronous execution
- cuFFT runs in default CUDA stream;
OpenACC not → trouble

⇒ Force cuFFT on OpenACC stream

```
#include <openacc.h>  
// Obtain the OpenACC default stream id  
cudaStream_t accStream = (cudaStream_t) acc_get_cuda_stream(acc_async_sync);  
// Execute all cufft calls on this stream  
cufftSetStream(accStream);
```

Task 8-4

TASK 8-4

OpenACC and cuFFT

- Use case: Fourier transforms
- Use cuFFT and OpenACC to solve Poisson's Equation

Task 8-4: OpenACC+cuFFT

- Change to Task8-4/ directory
- Work on TODOs in `poisson.c`
 - `solveRSpace` Force cuFFT on correct stream; implement data handling with
`host_data use_device`
 - `solveKSpace` Implement data handling and parallelism
- Compile: `make`
- Submit to the batch system: `make run`



Conclusions

Conclusions

- OpenACC directives and clauses
`#pragma acc parallel loop copyin(A[0:N]) reduction(max:err) vector`
- Start easy, optimize from there
- PGI / NVIDIA Visual Profiler help to find bottlenecks
- OpenACC is interoperable to other GPU programming models
- Don't forget the CPU version!

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

Appendix

- List of Tasks
- Glossary
- References

List of Tasks

Task 2: A First Parallel Loop

Task 3: More Parallel Loops

Task 4: Data Copies

Task 5: Data Region

Task 6: More Parallelism

Task 7: Loop Ordering

Task 7': Pinned Memory

Task 8-1: OpenACC+cuBLAS

Task 8-2: OpenACC+CUDA

Task 8-3: OpenACC+Thrust

Task 8-4: OpenACC+cuFFT

Glossary I

API A programmatic interface to software by well-defined functions. Short for application programming interface. [49](#)

CUDA Computing platform for [GPUs](#) from NVIDIA. Provides, among others, CUDA C/C++. [10](#), [17](#), [24](#), [49](#), [62](#), [63](#), [90](#), [108](#), [111](#), [115](#), [116](#), [117](#), [124](#), [125](#), [130](#), [132](#)

GCC The GNU Compiler Collection, the collection of open source compilers, among others for C and Fortran. [23](#), [26](#)

JURON One of the two HBP pilot system in Jülich; name derived from Juelich and Neuron. [11](#)

NVIDIA US technology company creating [GPUs](#). [19](#), [49](#), [64](#), [69](#), [70](#), [71](#), [103](#), [113](#), [124](#), [128](#), [131](#), [132](#), [133](#)

Glossary II

- OpenACC** Directive-based programming, primarily for many-core machines. [2](#), [17](#), [18](#), [19](#), [20](#), [21](#), [22](#), [24](#), [25](#), [26](#), [27](#), [28](#), [29](#), [32](#), [38](#), [39](#), [41](#), [43](#), [45](#), [49](#), [52](#), [55](#), [60](#), [64](#), [65](#), [72](#), [76](#), [79](#), [84](#), [85](#), [88](#), [90](#), [91](#), [103](#), [105](#), [106](#), [108](#), [109](#), [111](#), [113](#), [114](#), [115](#), [117](#), [120](#), [123](#), [125](#), [126](#), [128](#), [130](#)
- OpenCL** The *Open Computing Language*. Framework for writing code for heterogeneous architectures ([CPU](#), [GPU](#), DSP, FPGA). The alternative to [CUDA](#). [17](#)
- OpenMP** Directive-based programming, primarily for multi-threaded machines. [2](#), [17](#), [20](#), [54](#)
- PAPI** The Performance API, a C/C++ API for querying performance counters. [33](#)
- Pascal GPU** architecture from [NVIDIA](#) (announced 2016). [62](#), [63](#)

Glossary III

perf Part of the Linux kernel which facilitates access to performance counters; comes with command line utilities. [33](#)

PGI Compiler creators. Formerly *The Portland Group, Inc.*; since 2013 part of [NVIDIA](#). [2](#), [23](#), [26](#), [33](#)

POWER CPU architecture from IBM, earlier: PowerPC. See also POWER8. [62](#), [63](#), [133](#)

POWER8 Version 8 of IBM's [POWER](#) processor, available also under the OpenPOWER Foundation. [133](#)

Thrust A parallel algorithms library for (among others) GPUs. See <https://thrust.github.io/>. [17](#), [118](#), [119](#), [120](#), [130](#)

CPU Central Processing Unit. [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [23](#), [49](#), [62](#), [63](#), [99](#), [101](#), [109](#), [128](#), [132](#), [133](#)

Glossary IV

GPU Graphics Processing Unit. 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 17, 18, 23, 25, 45, 49, 62, 63, 64, 69, 75, 77, 103, 109, 114, 116, 128, 131, 132, 133

HBP Human Brain Project. 131

References I

- [4] John L. Gustafson. “Reevaluating Amdahl’s Law”. In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: [10.1145/42411.42415](https://doi.org/10.1145/42411.42415). URL: <http://doi.acm.org/10.1145/42411.42415>.
- [6] 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> (page 33).

References: Images, Graphics

- [1] SpaceX. *SpaceX Launch*. Freely available at Unsplash. URL: <https://unsplash.com/photos/uj3hvdfQujI>.
- [2] Mark Lee. *Picture: kawasaki ninja*. URL: <https://www.flickr.com/photos/pochacco20/39030210/> (page 4).
- [3] Shearings Holidays. *Picture: Shearings coach 636*. URL: <https://www.flickr.com/photos/shearings/13583388025/> (page 4).
- [5] Setyo Ari Wibowo. *Ask*. URL: <https://thenounproject.com/term/ask/1221810>.