University of Zurich[UZH]

High Performance Computing Lecture 12

Douglas Potter
Jozef Bucko
Noah Kubli

The OpenACC Library


Introduction


Basic principles


kernels construct


parallel construct


Performance


Synchronization

Introduction and history

GPU computing using directives:

- Goal to provide directive-based **GPU** computing like OpenMP

Release History:

- version 1.0: November 2011
- version 2.0: June 2013
- version 2.5: October 2015
- **version 2.6: November 2017**
- version 3.0: November 2019
- version 3.2: November 2021

Resources

Documentation

- http://www.openacc.org/

Specification

- https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf

Guides, Tutorials, Books

- https://www.openacc.org/resources

OpenACC structure

Compilation directives and clauses:

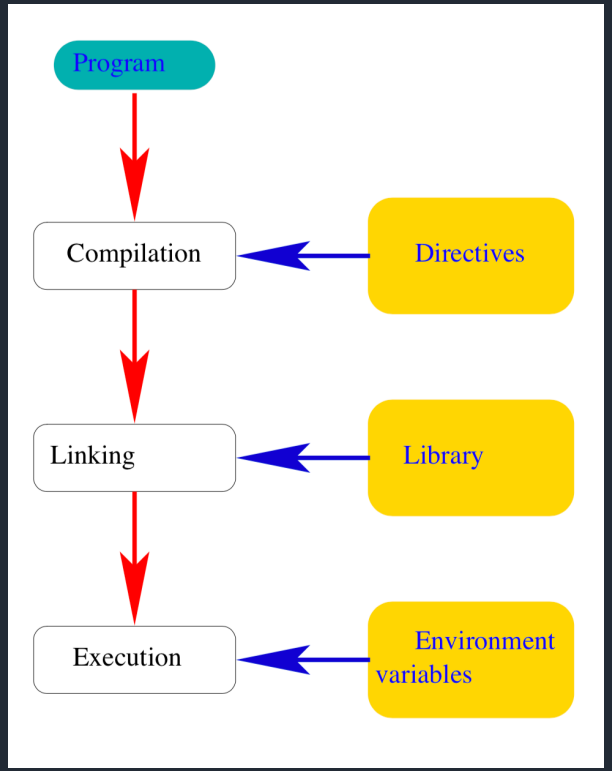
- They are put in the program to parallelize loops, define the work and data sharing strategy, and synchronize
- They are considered by the C, C++ or Fortran compilers as mere comment lines unless one specifies –acc on the compilation command line

Functions and routines:

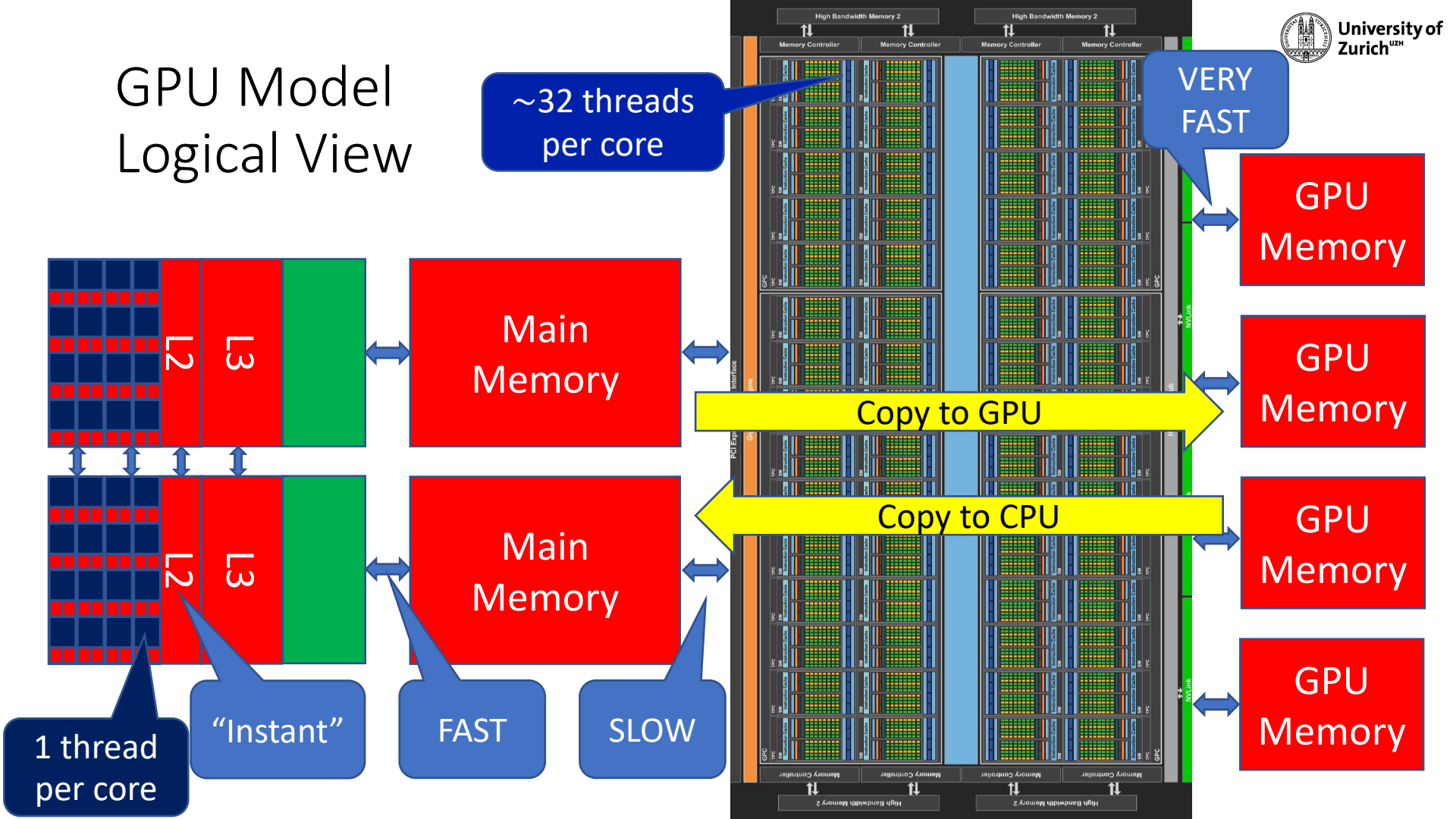
- OpenACC contains several dedicated functions (like MPI). They are part of the OpenACC library than can be linked at link time.

Environment variables:

- OpenACC has several environment variables that can be set at execution time and changed the parallel computing behavior.



GPU Model Logical View



Important GPU Operations

Allocate or Free GPU memory

Copy data from “host” to GPU

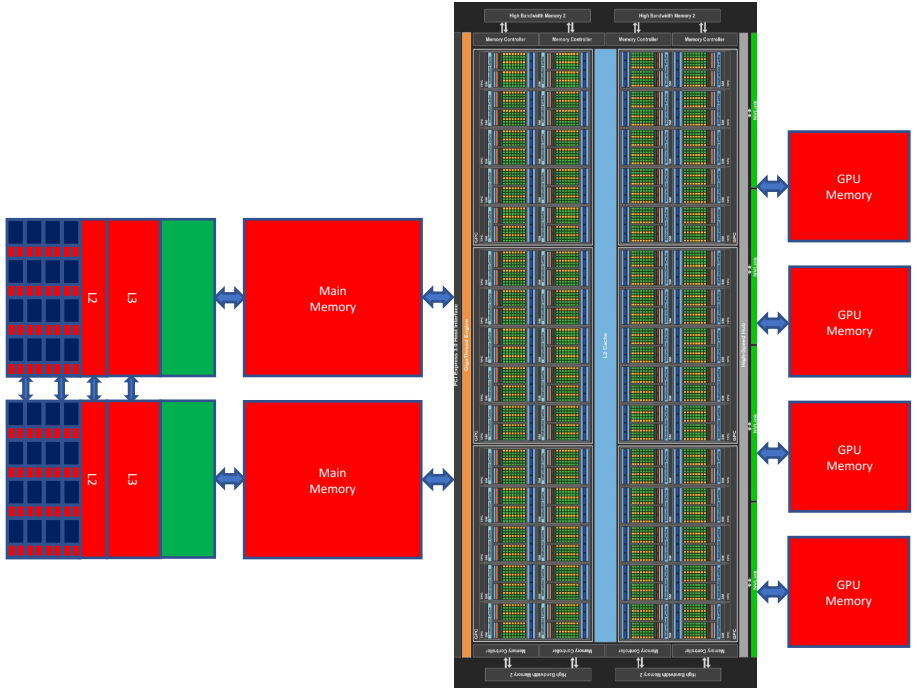
Launch a “kernel”

Copy data from GPU to host

You can request these in any order

OpenACC handles this for you

Transfers and Kernels can overlap



SAXPY – single-precision aX + Y

- **X** and **Y** are vectors
  - array of floating point numbers
- "a" is a scalar (a single float)
- Result is stored in **Y**
- Part of BLAS / LAPACK
- **Flops to Byte Ratio?**

$$\boldsymbol{Y}_i = a\boldsymbol{X}_i + \boldsymbol{Y}_i$$

```
void saxpy(int n, float a,
           float * restrict x,
           float * restrict y) {
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

Serial Code Solution

Calculate $\boldsymbol{Y}_0 = a\boldsymbol{X}_0 + \boldsymbol{Y}_0$

Calculate $\boldsymbol{Y}_1 = a\boldsymbol{X}_1 + \boldsymbol{Y}_1$

Calculate $\boldsymbol{Y}_2 = a\boldsymbol{X}_2 + \boldsymbol{Y}_2$

Calculate $\boldsymbol{Y}_3 = a\boldsymbol{X}_3 + \boldsymbol{Y}_3$

… and so on to e.g. $\boldsymbol{Y}_{3999}$

One possible OpenMP Solution

Thread

0: Calculate $\boldsymbol{Y}_0$ through $\boldsymbol{Y}_{999}$

1: Calculate $\boldsymbol{Y}_{1000}$ through $\boldsymbol{Y}_{1999}$

2: Calculate $\boldsymbol{Y}_{2000}$ through $\boldsymbol{Y}_{2999}$

3: Calculate $\boldsymbol{Y}_{3000}$ through $\boldsymbol{Y}_{3999}$

GPU Code Solution

Thread

0: Calculate $\boldsymbol{Y}_0 = a\boldsymbol{X}_0 + \boldsymbol{Y}_0$

1: Calculate $\boldsymbol{Y}_1 = a\boldsymbol{X}_1 + \boldsymbol{Y}_1$

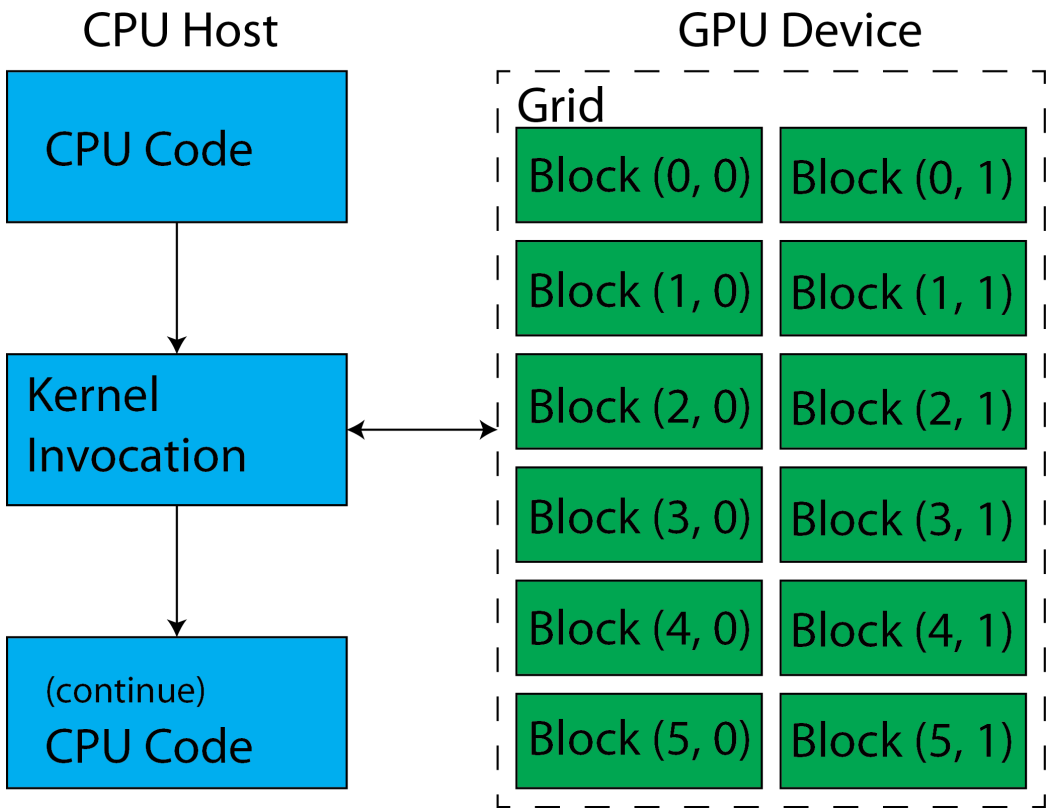
2: Calculate $\boldsymbol{Y}_2 = a\boldsymbol{X}_2 + \boldsymbol{Y}_2$

3: Calculate $\boldsymbol{Y}_3 = a\boldsymbol{X}_3 + \boldsymbol{Y}_3$

… and so on to **thread** 3999

Basic principles of OpenACC

- **OpenACC uses directives**
- The developer introduces directives as with OpenMP.
- At execution time, when the program hits a directive, the operating system creates a parallel regions, and offloads calculations to the GPU.
- Data transfers can be implicit (managed by OpenACC) or explicit.
- Careful: kernel invocations are expensive. Similar to the latency in MPI.



Directive Syntax

- The directives are ignored by the compiler if OpenACC is disabled
  - The C compiler ignores unknown pragmas
  - The ”sentinel” in Fortran looks like a comment
- Disabled if unsupported or if “-acc” not specified

In C and C++, OpenACC directives are specified with the **#pragma** mechanism. The syntax of an OpenACC directive is:

#pragma acc *directive-name* [*clause-list*] *new-line*

Each directive starts with **#pragma acc**. The remainder of the directive follows the C and C++ conventions for pragmas. White space may be used before and after the **#**; white space may be required to separate words in a directive. Preprocessing tokens following the **#pragma acc** are subject to macro replacement. Directives are case-sensitive.

Compiling

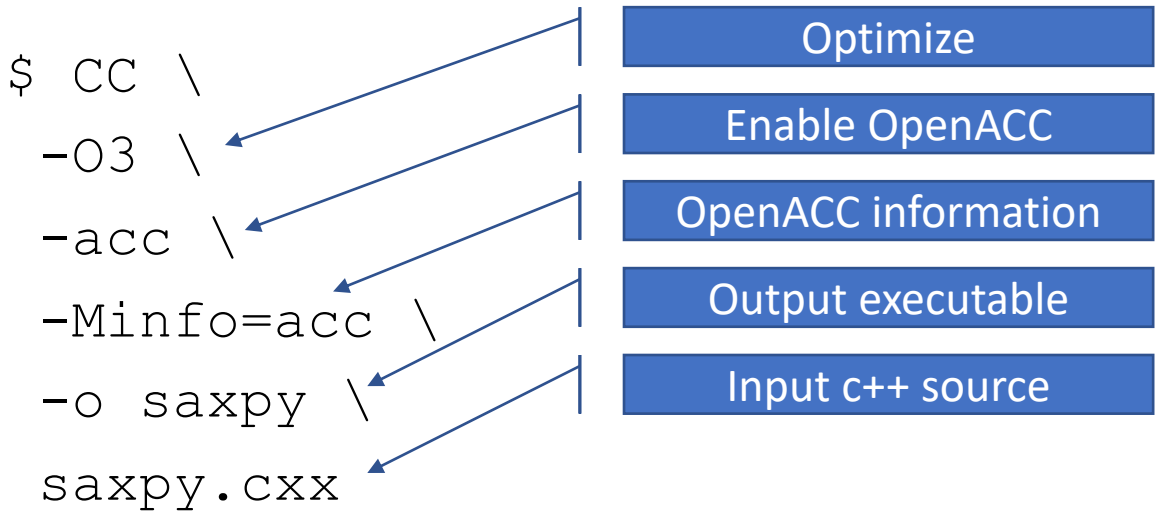
nVidia HPC SDK

```
$ nvc++ --version

nvc++ 21.3-0 LLVM 64-bit target on x86-64 Linux -tp haswell
NVIDIA Compilers and Tools
Copyright (c) 2020, NVIDIA CORPORATION.  All rights reserved.
```

Piz Daint

```
> module load daint-gpu
> module load cudatoolkit
> module swap PrgEnv-cray PrgEnv-nvidia
> cc --version

nvc++ 21.3-0 LLVM 64-bit target on x86-64 Linux -tp haswell
NVIDIA Compilers and Tools
Copyright (c) 2020, NVIDIA CORPORATION.  All rights reserved.
```

nVidia (formerly PGI) Compiler Flags




```
$ CC -O3 -acc -Minfo=acc -o saxpy saxpy.cxx
```

“kernels” construct

```
#pragma acc kernels
{
    // parallel block
}
```

- The “kernels” construct is used to identify blocks of code that **may** contain parallelism.
- The compiler will analyze the block and **if appropriate** generate one or more kernels and data transfer operations.
- You are relying completely on the compiler knowing the best way to parallelize the block.

“kernels” construct (and pointer aliasing)

```
int main() {
    const int N = 1'000'000'000;
    auto x = new float[N];
    auto y = new float[N];
    #pragma acc kernels
    {
        for (auto i=0; i<N; i++) {
            y[i] = 0.0f;
            x[i] = (float)(i+1);
        }
        for (auto i=0; i<N; i++) {
            y[i] = 2.0f * x[i] + y[i];
        }
    }
    delete [] x;
    delete [] y;
}
```

```
$ CC -O3 -o saxpy1 -Minfo=acc -acc saxpy1.cxx
main:
      6, Generating implicit copyout(y[:1000000000],x[:1000000000]) [if not
already present]
      7, Complex loop carried dependence of y-> prevents parallelization
         Loop carried dependence of x-> prevents parallelization
         Loop carried backward dependence of x-> prevents vectorization
         Accelerator serial kernel generated
         Generating Tesla code
          7, #pragma acc loop seq
      7, Loop carried dependence of x-> prevents parallelization
     11, Complex loop carried dependence of x-> prevents parallelization
         Loop carried dependence of y-> prevents parallelization
         Loop carried backward dependence of y-> prevents vectorization
         Accelerator serial kernel generated
         Generating Tesla code
         11, #pragma acc loop seq
     11, Loop carried dependence of y-> prevents parallelization
         Loop carried backward dependence of y-> prevents vectorization
```

$$\boldsymbol{Y}_i = a\boldsymbol{X}_i + \boldsymbol{Y}_i$$

“kernels” construct and “restrict” keyword

```
int main() {
    const int N = 1'000'000'000;
    auto restrict x = new float[N];
    auto restrict y = new float[N];
    #pragma acc kernels
    {
        for (auto i=0; i<N; i++) {
            y[i] = 0.0f;
            x[i] = (float)(i+1);
        }
        for (auto i=0; i<N; i++) {
            y[i] = 2.0f * x[i] + y[i];
        }
    }
}
```

```
$ CC -O3 -o saxpy1 -Minfo=acc -acc saxpy1.cxx
main:
      6, Generating implicit copyout(x[:1000000000],y[:1000000000]) [if not already present]
      7, Loop is parallelizable
         Generating Tesla code
          7, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     11, Loop is parallelizable
         Generating Tesla code
         11, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
$ nvprof ./saxpy1
==824== NVPROF is profiling process 824, command: ./saxpy1
==824== Profiling application: ./saxpy1
==824== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 92.05%  1.10528s       478  2.3123ms  686.14us  3.9924ms  [CUDA memcpy DtoH]
  4.32%  51.828ms         1  51.828ms  51.828ms  51.828ms  main_11_gpu
  3.63%  43.646ms         1  43.646ms  43.646ms  43.646ms  main_7_gpu
```

“parallel” construct

```
#pragma acc parallel
{
    // parallel block
}
```

- Identifies a block of code that will be parallized.
- Programmer is responsible for making sure it is “safe” to do so.
- Limited use unless paired with the “loop” directive. This is different from OpenMP.

“loop” construct

```
#pragma acc parallel
{
    #pragma acc loop
    for(…) {}
}

#pragma acc parallel loop
for(…) {}
```

- Indicates that the loop immediately following should be parallelized (like in OpenMP).
- May be combined with the parallel construct.

“parallel” construct with “loop”

```
int main() {
    const int N = 1'000'000'000;
    auto x = new float[N];
    auto y = new float[N];
    #pragma acc parallel
    {
        #pragma acc loop
        for (auto i=0; i<N; i++) {
            y[i] = 0.0f;
            x[i] = (float)(i+1);
        }
        #pragma acc loop
        for (auto i=0; i<N; i++) {
            y[i] = 2.0f * x[i] + y[i];
        }
    }
}
```

```
$ CC -O3 -o saxpy2 -Minfo=acc -acc saxpy2.cxx
main:
      6, Generating Tesla code
          8, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         13, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
      6, Generating implicit copyout(y[:1000000000],x[:1000000000]) [if not
already present]
$ nvprof ./saxpy2
==1172== NVPROF is profiling process 1172, command: ./saxpy2
==1172== Profiling application: ./saxpy2
==1172== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 90.10%  959.49ms       478  2.0073ms  753.47us  3.3830ms  [CUDA memcpy DtoH]
  9.90%  105.46ms         1  105.46ms  105.46ms  105.46ms  main_6_gpu
```

Single Kernel!

“parallel” construct with “loop”

```
int main() {
    const int N = 1'000'000'000;
    auto x = new float[N];
    auto y = new float[N];
    #pragma acc parallel loop
    for (auto i=0; i<N; i++) {
        y[i] = 0.0f;
        x[i] = (float)(i+1);
    }
    #pragma acc parallel loop
    for (auto i=0; i<N; i++) {
        y[i] = 2.0f * x[i] + y[i];
    }
}
```

```
$ CC -O3 -o saxpy3 -Minfo=acc -acc saxpy3.cxx
main:
      4, Generating Tesla code
          6, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
      4, Generating implicit copyout(x[:1000000000],y[:1000000000]) [if not
already present]
      9, Generating Tesla code
         11, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
      9, Generating implicit copyin(x[:1000000000]) [if not already present]
         Generating implicit copy(y[:1000000000]) [if not already present]
$ nvprof ./saxpy3
==1310== NVPROF is profiling process 1310, command: ./saxpy2
==1310== Profiling application: ./saxpy2
==1310== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 64.68%  1.29333s       717  1.8038ms  579.42us  3.4421ms  [CUDA memcpy DtoH]
 30.50%  609.87ms       478  1.2759ms  535.39us  1.4902ms  [CUDA memcpy HtoD]
  2.64%  52.772ms         1  52.772ms  52.772ms  52.772ms  main_9_gpu
  2.18%  43.548ms         1  43.548ms  43.548ms  43.548ms  main_4_gpu
```

“data” construct

```
#pragma acc data clauses
{
    // parallel block(s)
}
```

- **copy** – Allocate and copy variable to the device and copy it back at the end
- **copyin** – Allocate and copy to device
- **copyout** – Allocate space but do not initialize. Copy to host at the end
- **create** – allocate space but do not initialize or copy back to the host
- **present** – the variable is already present on the device (when data regions are nested)
- **present_or_***, e.g., **pcreate**

“data” construct

```
int main() {
    const int N = 1'000'000'000;
    auto x = new float[N];
    auto y = new float[N];
    #pragma acc data pcreate(x[0:N]) pcopyout(y[:N])
    {
        #pragma acc parallel loop
        for (auto i=0; i<N; i++) {
            y[i] = 0.0f;
            x[i] = (float)(i+1);
        }
        #pragma acc parallel loop
        for (auto i=0; i<N; i++) {
            y[i] = 2.0f * x[i] + y[i];
        }
    }
}
```

Array Range

```
$ CC -O3 -o saxpy4 -Minfo=acc -acc saxpy4.cxx
main:
      6, Generating create(x[:1000000000]) [if not already present]
         Generating copyout(y[:1000000000]) [if not already present]
         Generating Tesla code
          8, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     11, Generating Tesla code
         13, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
$ nvprof ./saxpy4
==1728== NVPROF is profiling process 1728, command: ./saxpy4
==1728== Profiling application: ./saxpy4
==1728== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 83.55%  484.79ms       239  2.0284ms  1.3351ms  3.3359ms  [CUDA memcpy DtoH]
  8.93%  51.799ms         1  51.799ms  51.799ms  51.799ms  main_11_gpu
  7.53%  43.669ms         1  43.669ms  43.669ms  43.669ms  main_6_gpu
```

Nested data regions

```
void init(int N,float *x, float *y) {
  #pragma acc data pcopyout(x[0:N]) pcopyout(y[0:N])
  {
    #pragma acc parallel loop
    for (auto i=0; i<N; i++) {
      y[i] = 0.0f;
      x[i] = (float)(i+1);
    }
  }
}

void saxpy(int N,float a,float * restrict x,float * restrict y){
  #pragma acc data pcopyin(x[0:N]) pcopy(y[0:N])
  {
    #pragma acc parallel loop
    for (auto i=0; i<N; i++) {
      y[i] = a * x[i] + y[i];
    }
  }
}
```

```
int main() {
  const int N = 1'000'000'000;
  auto x = new float[N];
  auto y = new float[N];
  #pragma acc data pcreate(x[0:N]) pcopyout(y[0:N])
  {
    init(N,x,y);
    saxpy(N,2.0f,x,y);
  }
  // Here we would use the result 'y'
}
```

Nested data regions

```
$ CC -O3 -o saxpy5 -Minfo=acc -acc saxpy5.cxx
init(int, float *, float *):
      3, Generating copyout(y[:N],x[:N]) [if not already present]
         Generating Tesla code
          5, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
saxpy(int, float, float *, float *):
     13, Generating copy(y[:N]) [if not already present]
         Generating copyin(x[:N]) [if not already present]
         Generating Tesla code
         15, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
main:
     26, Generating copyout(y[:1000000000]) [if not already present]
         Generating create(x[:1000000000]) [if not already present]
$ nvprof ./saxpy5
==2582== NVPROF is profiling process 2582, command: ./saxpy5
==2582== Profiling application: ./saxpy5
==2582== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 83.64%  487.23ms       239  2.0386ms  1.3387ms  3.2830ms  [CUDA memcpy DtoH]
  8.89%  51.783ms         1  51.783ms  51.783ms  51.783ms  saxpy_13_gpu(int, float, float*, float*)
  7.47%  43.510ms         1  43.510ms  43.510ms  43.510ms  init_3_gpu(int, float*, float*)
```

Updates inside a data region

```
#pragma acc data …
{
  do_some_gpu_work(x,…);
  #pragma acc update self(x)
  do_something_on_the_cpu(x);
  #pragma acc update device(x)
  do_more_gpu_work(x,…);
}
```

Unstructured data regions

- So far we have discussed “structured” data regions.
- The start and end of the region is clearly defined
  - Between the open and closing braces in C/C++
- There are cases where such an elegant data scope is not possible
  - For example you may want to start a data region in a C++ constructor and end it in the destructor for a class.
- Use “enter data” and “exit data”

```
class foo {
  float *v;
  foo(int n) {
    v = new float[n];
    #pragma acc enter data create(v[:n])
  }
  ~foo() {
     #pragma acc exit data delete(v)
     delete [] v;
  }
```

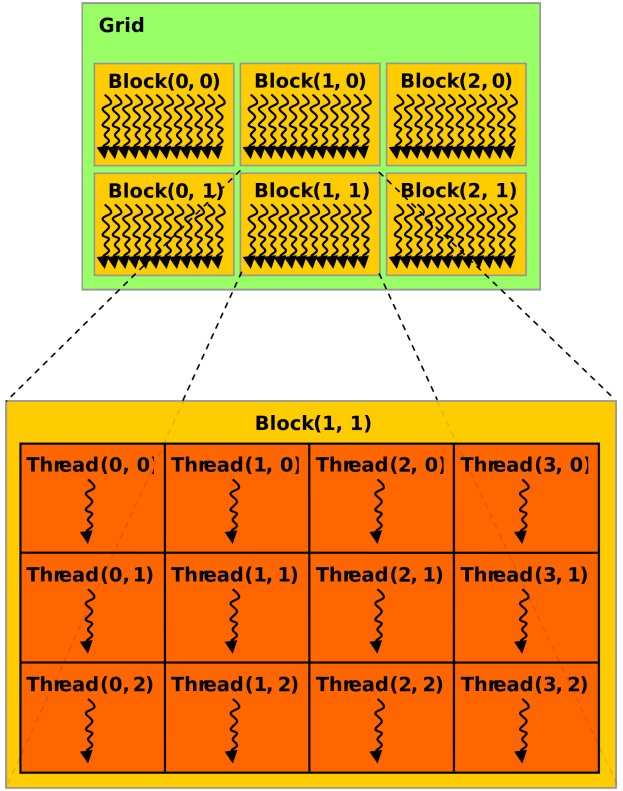
Data Scope (shared or private)

- **Scalar** and loop index variables are private by default to each thread.
  - Careful: This is different than OpenMP
- Arrays are shared by default.
- Explicit data clauses override automatic scoping decisions; for example the “private” clause can be used with parallel, kernels or loop.
- Like with OpenMP you can specify default(none)
  - You have to specify the scope of every variable (except loop indexes), otherwise the compiler will issue an error.

Block runs on a single SM

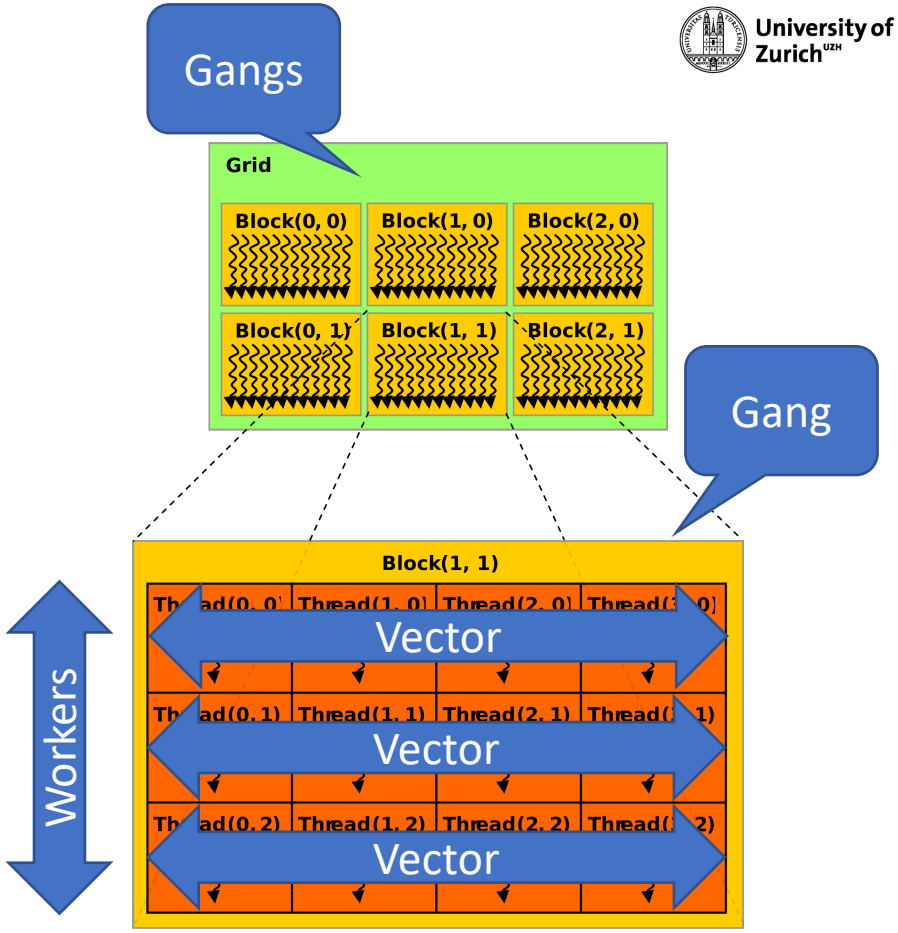
(up to 32 blocks at once per SM)

(2 x 1024 = 2048 threads maximum)



Gangs, Vectors, Workers

- OpenACC is a general abstraction
- nVidia: vectors and workers map onto a thread block.
  - X and Y dimensions of the block
  - Limit of 1024 threads
  - Only total number of threads matter
- “Vector” matters for e.g. SIMD
- Gangs are schedule by the GPU on available resources (SMs)



Vector Length

- Conceptually a “vector” executes the same instruction on each thread
- Mapped onto warps, so the vector should be 32, or a multiple of 32 threads for efficiency.
- For some problems this is not possible so a non-commensurate number (e.g., 27) can be used. Some threads (cores) will be idle/wasted.

Using Gangs and Vectors

- gang, worker and/or vector can be added to a loop clause
- The size of each can be specified with num_gangs(n), num_workers(n), vector_length(n)

```
#pragma acc parallel loop gang
for(int i=0; i<n; ++i)
  #pragma acc loop vector
  for(int j=0; j<n; ++j)
    …
```

```
#pragma acc parallel vector_length(32)
#pragma acc loop gang worker
for(int i=0; i<n; ++i)
  #pragma acc loop vector
  for(int j=0; j<n; ++j)
    …
```

General rule: use vectors for the inner loop and gangs/workers for the outer loops

Gangs and Vectors

```
#pragma acc parallel loop gang
for(int i=0; i<n; ++i)
  #pragma acc loop vector
  for(int j=0; j<m; ++j)
    …
```

Consider the following nested loops:

Assume n=32000 and m=28.

1. What is the best choice for the vector size?
   vector_length(32) is a good choice (although 4 threads are wasted)
2. What is the optimal choice for the number of gangs?
   As many as possible! However, each SM has a limit on how many blocks (gangs) it can manage at once. For the P100, V100 and A100 this is 32. The number of active threads would be 32 x 32 = 1024. This is less than the maximum number of threads which is 2048 which leads to reduced occupancy (50%).

Workers

Remember that threads are used to handle latency and OCCUPANCY (number of active threads) is a measure of how well this is achieved. Occupancy is a indication of efficiency, but not a direct measure of it. We can use workers to help increase efficiency.

The compiler will probably define 2 or 4 workers per gang resulting in 100% occupancy.

```
#pragma acc parallel vector_length(32)
#pragma acc loop gang worker
for(int i=0; i<n; ++i)
  #pragma acc loop vector
  for(int j=0; j<m; ++j)
    …
```

The number of workers can be set explicitly with num_workers(2).

A gang is a pool of workers running on the same SM.

Sequential and independent loops

Sometime loops need to be executed sequentially, for example if the result depends on a previous iteration. For this you can use seq.

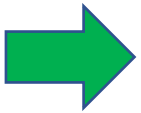
To indicate the loops are independent from loops that follow you can use the independent clause.

```
#pragma acc data copyin(a,b) copy(c)
#pragma acc kernels
#pragma acc loop independent
for(int i=0; i<n; ++i) {
  #pragma acc loop independent
  for(int j=0; j<n; ++j) {
    #pragma acc loop seq
    for(int k=0; k<n; ++k) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

collapse

- The collapse(n) clause will combine the following n loops

```
#pragma acc parallel loop collapse(2)
for(int i=0; i<n; ++i)
  for(int j=0; j<m; ++j)
    …
```



```
#pragma acc parallel loop
for(int ij=0; ij<n*m; ++ij)
  …
```

- Collapse outer loops to create more gangs
- Collapse inner loops to enable longer vector lengths
- It’s good practice to collapse all loops when possible

Reduction

```
#pragma acc parallel
#pragma acc loop reduction(+:sum) private(x)
for (i=0; i < steps; i++) {
  x = (i+0.5)*step;
  sum += 4.0 / (1.0+x*x);
}
```

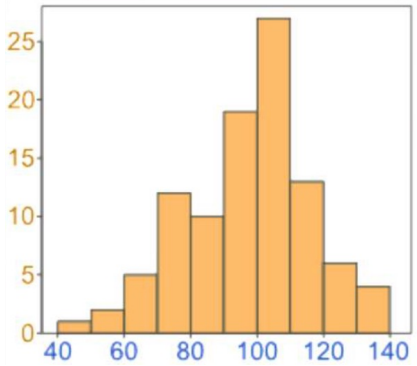
- Same idea as OpenMP
- Avoids race conditions
- Improves performance
- In the “cpi” example
- Clause can be used with parallel, kernels and loop constructs.
- Usual set of possible operations (e.g., product, sum, min, max, etc.)

Synchronization

- An implicit synchronization happens when leaving a “kernels” or “parallel” region.
  - All threads will have completed before the region ends.
- There is no implicit synchronization inside a parallel region.
  - A second loop inside a parallel region could start before all threads doing the first loop finishes.
  - The order of loop execution is not preserved – indexes are processed in any order.
- Loop execution order is preserved in the kernels contruct
  - Each loop is a separate kernel invocation on the device

Atomic

- Same idea as OpenMP
- Avoids race conditions
- Possible performance impact
- Hardware support




```
#pragma acc data copyin(a[:n]) copyout(h[:nbins])
{
  #pragma acc parallel loop
  for(int i=0; i<nbins; ++i) h[i] = 0;
  #pragma acc parallel loop
  for(int i=0; i<n; ++i) {
    #pragma acc atomic update
    ++h[a[i]];
  }
}
```

Calling functions inside parallel regions

- Originally OpenACC did not support calling functions from inside a parallel region. Instead you needed to use inlining.
- Now you can use the routine directive.
- This specifies that the compiler should generate both a GPU and a CPU version of the function.
- Specification of the level of parallization is **mandatory**. Options are:
  - gang
  - worker
  - vector
  - seq

Routine Directive

```
// foo.h
#pragma acc routine seq
double foo(int i);
```

```
// in main()
#pragma acc parallel loop
for(int i=0; i<n; ++i)
  array[i] = foo(i);
```

- At function source:
  - Function must be built for GPU
  - It will be called sequentially
    - i.e., independent from other threads
- At function call:
  - Function will be on the GPU
  - It is a sequential routine

Asyncronous Programming

- So far we have been using “synchronous” programming
  - Work is scheduled in a parallel section
  - When the section finishes, the host waits for the GPU
    - All kernels have finished executing
    - All data transfers are complete
- Most OpenACC directives can be made “asynchronous”
  - Host issues multiple parallel loops to the GPU
  - The host can then perform other calculations while the GPU is busy
  - Data transfers can happen before the data is needed

“async” and “wait”

- The async(n) clause launches work asynchonously in queue n.
- The wait(n) directive waits for all work in queue n to complete.
- This can significantly reduce launch latency and enable pipelines and concurrent operations.

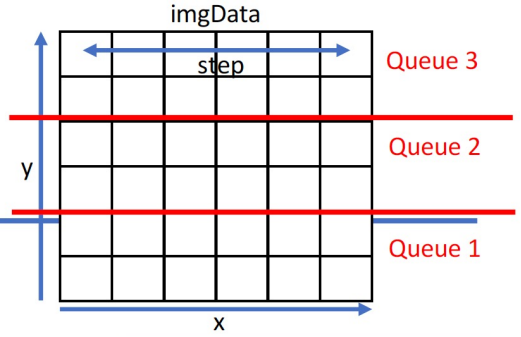
If n is not specified then work will go to the default queue, and wait will wait for all previously queued work.

```
#pragma acc parallel loop async(1)
…
#pragma acc parallel loop async(1)
for(i=0; i<N; ++i) …
// host can work independently here
#pragma acc wait(1)
for(i=0; i<N; ++i) … // Host deals with array
```

Queues are CUDA streams

- On nVidia, a “queue” maps onto a “stream”
- A stream does one thing at a time, in order (often what you want):
  - Copy data to the GPU
  - Execute a kernel
  - Execute a second kernel …
  - Copy results back to the CPU
- With multiple queues/streams you can run multiple kernels at once
- Subject to resource constraints
  - There are a fixed number of SMs
  - You can have only one copy (possibly each direction) active at a time.

async example



Data is only created (no copy)

```
#pragma acc data create(imgData[w*h*ch],out[w*h*ch])
                copyin(filter)
{
for ( long blocky = 0; blocky < nblocks; blocky++)
{
  long starty = MAX(0,blocky * blocksize - filtersize/2);
  long endy   = MIN(h,starty + blocksize + filtersize/2);
#pragma acc update device(imgData[starty*step:blocksize*step]) async(block%3+1)
  starty = blocky * blocksize;
  endy = starty + blocksize;
#pragma acc parallel loop collapse(2) gang vector async(block%3+1)
  for (y=starty; y<endy; y++) for ( x=0; x<w; x++ ) {
    <filter code ommitted>
    out[y * step + x * ch]     = 255 - (scale * blue);
    out[y * step + x * ch + 1 ] = 255 - (scale * green);
    out[y * step + x * ch + 2 ] = 255 - (scale * red);
  }
#pragma acc update self(out[starty*step:blocksize*step]) async(block%3+1)
}
#pragma acc wait
}
```

Cycle between 3 async queues by blocks.

Wait for all blocks to complete.

Number of elements between two rows

Courtesy of Claudio Gheller

Final words

- Understanding memory management is crucial for performance.
  - OpenACC manages “shared” memory and “caches” for you.
- Parallelization hierachy: Gangs, Workers, vectors, sequential.
- Be careful about order of execution (kernels versus parallel).
- Using asynchronous operation you can overlap transfers, GPU and host computations.
- Not covered
  - OpenACC and CUDA can be used together
  - Global variables (nice for static tables)