GPU programming with OpenACC

October 22 – 23, 2020

CSC – IT Center for Science Ltd., Espoo

Martti Louhivuori
Georgios Markomanolis

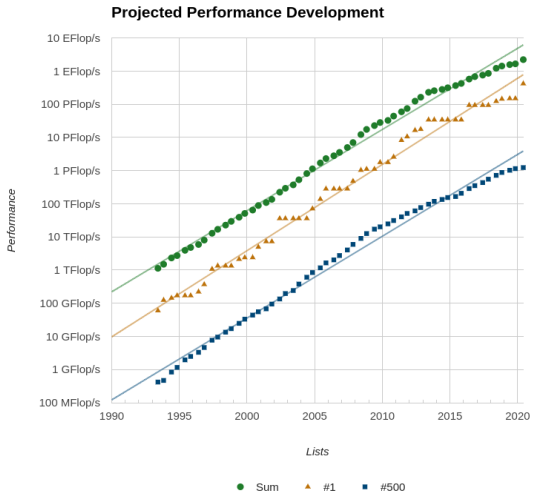
CSC
Training for Brilliant Minds

All material (C) 2011–2020 by CSC – IT Center for Science Ltd.
This work is licensed under a **Creative Commons Attribution-ShareAlike** 4.0 Unported License, http://creativecommons.org/licenses/by-sa/4.0

Introduction to GPUs in HPC

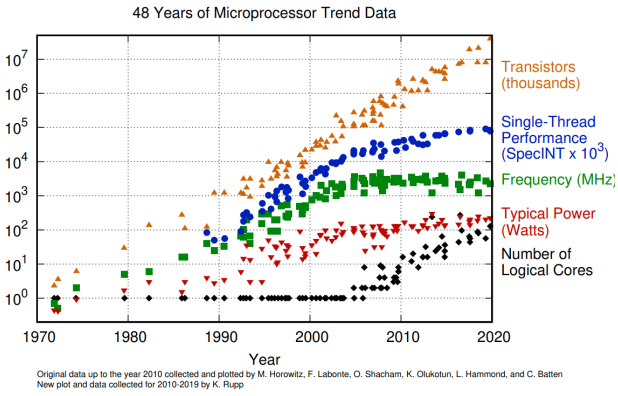
High-performance computing

- High performance computing is fueled by ever increasing performance
- Increasing performance allows breakthroughs in many major challenges that humankind faces today
- Not only hardware performance, algorithmic improvements have also added ordered of magnitude of real performance



HPC through the ages

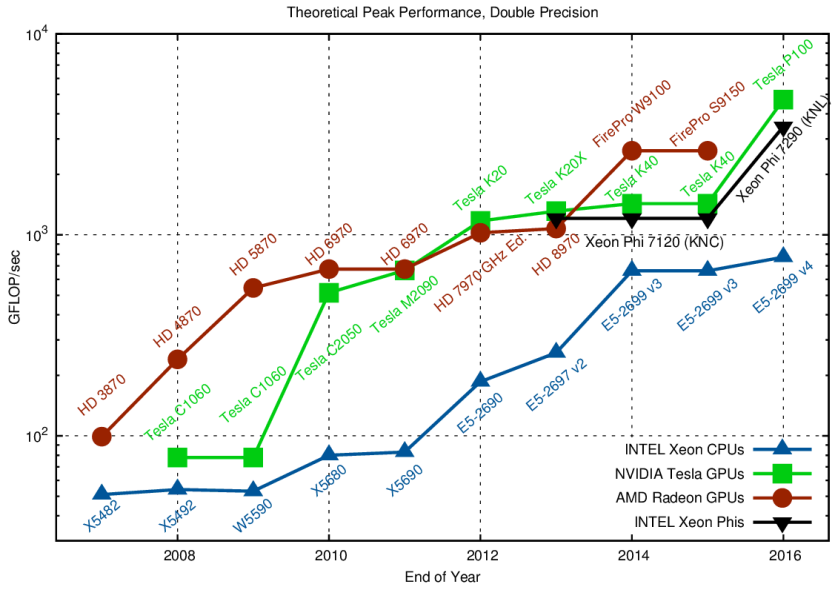
- Achieving performance has been based on various strategies throughout the years
  - Frequency, vectorization, multinode, multicore …
  - Now performance is mostly limited by power consumption
- Accelerators provide compute resources based on a very high level of parallelism to reach high performance at low relative power consumption



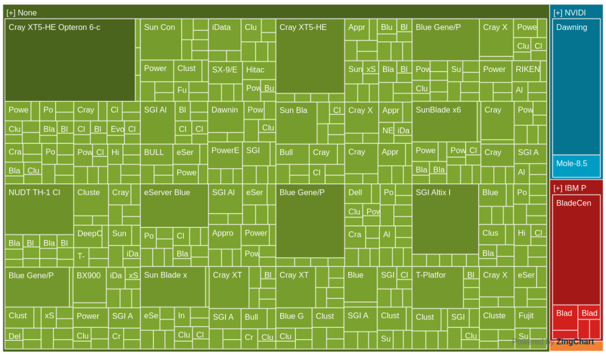
Accelerators

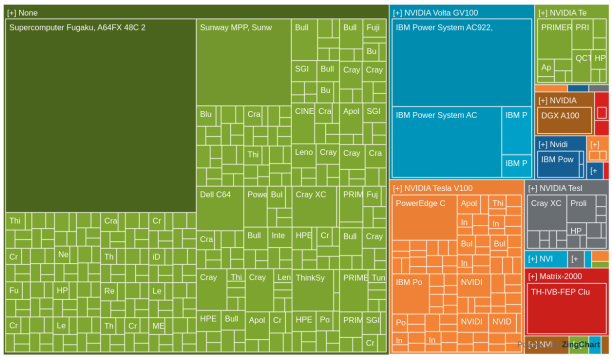
- Specialized parallel hardware for floating point operations
  - Co-processors for traditional CPUs
  - Based on highly parallel architectures
  - Graphics processing units (GPU) have been the most common accelerators during the last few years
- Promises
  - Very high performance per node
- Usually major rewrites of programs required

Accelerator performance growth

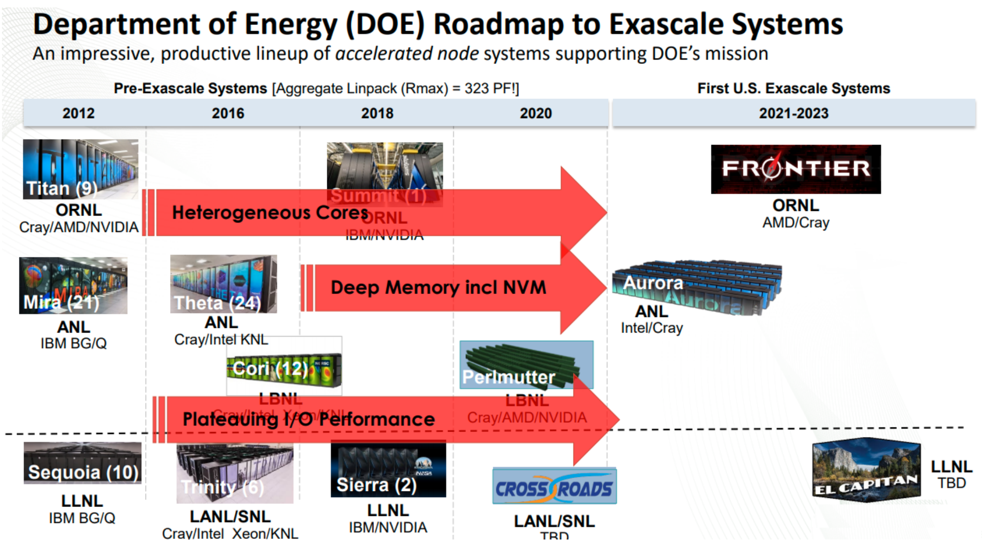


Accelerators share of 500 fastest systems (Top500) 2010 vs 2020

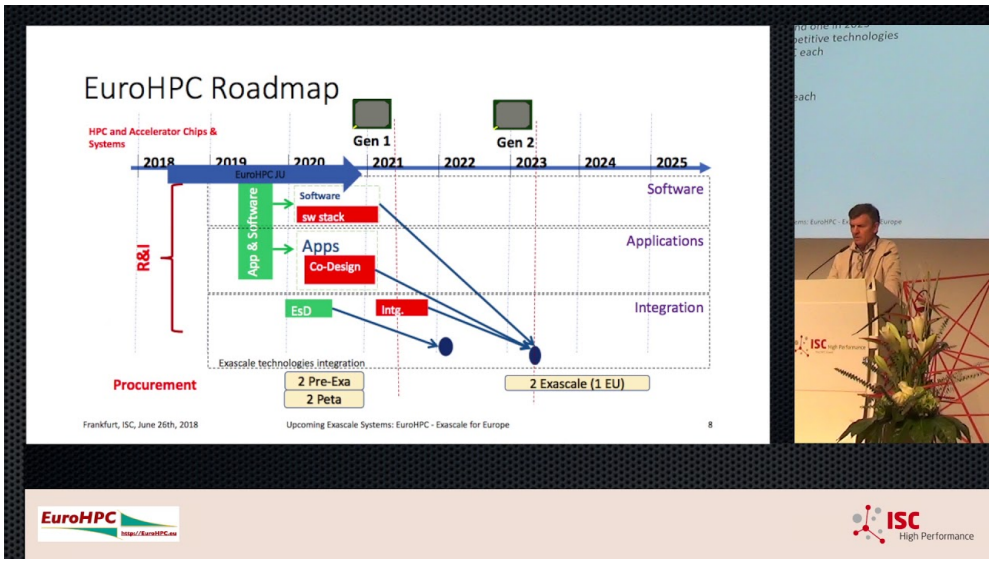




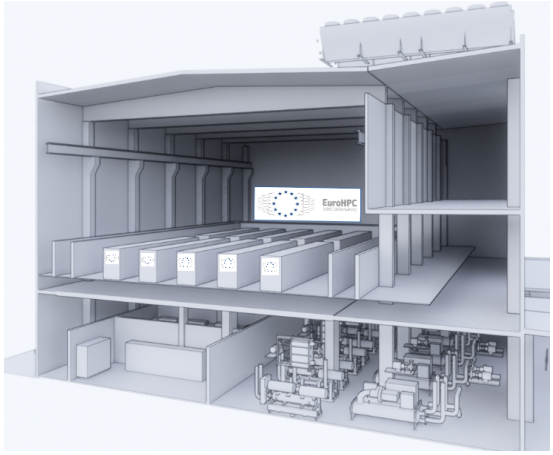
US roadmap to Exascale

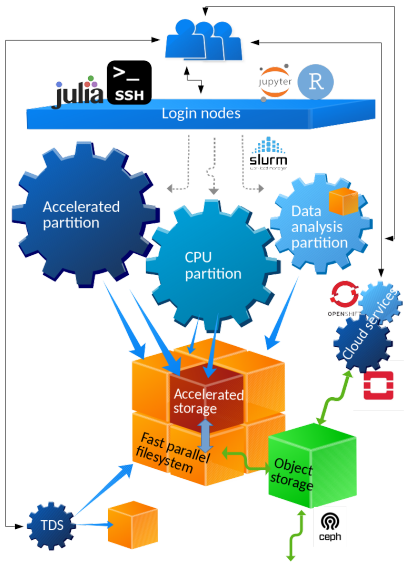


EU roadmap to Exascale



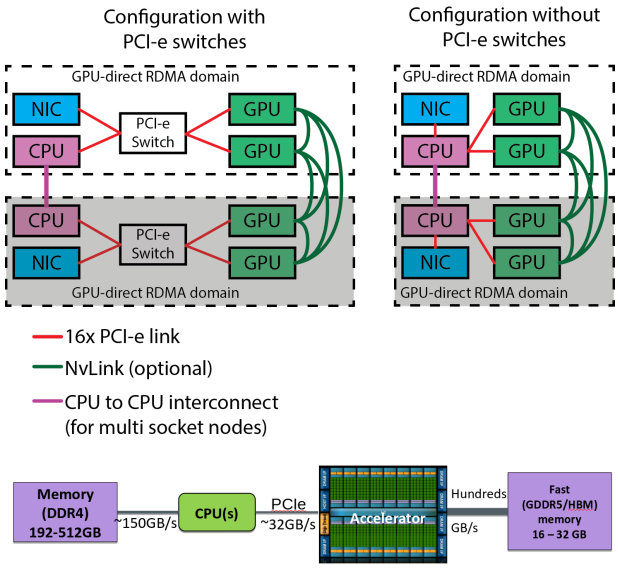
Lumi - Pre-exascale system in Finland





Accelerator model today

- Connected to CPUs via PCIe
- Local memory
  - Smaller than main memory (32 GB in Puhti)
  - Very high bandwidth (up to 900 GB/s)
  - Latency high compared to compute performance
- Data must be copied over the PCIe bus



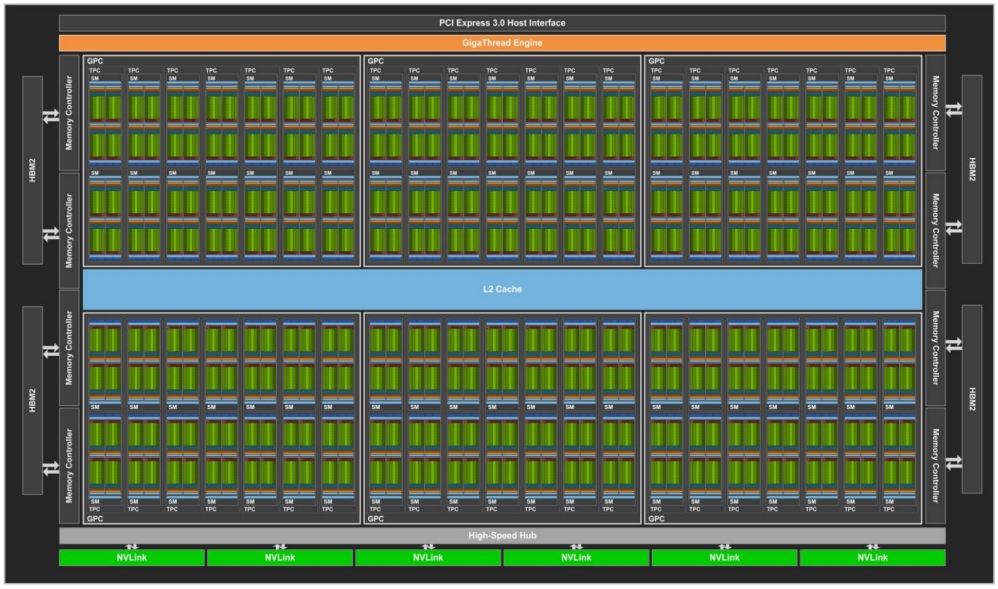
GPU architecture

- Designed for running tens of thousands of threads simultaneously on thousands of cores
- Very small penalty for switching threads
- Running large amounts of threads hides memory access penalties
- Very expensive to synchronize all threads
- Now Nvidia GPUs have close to monopoly in HPC - will change in next few years

GPU architecture: Nvidia Volta

- 80 streaming multi processor units (SM), each comprising many smaller Cuda cores
  - 5120 single precision cores
  - 2560 double precision cores
  - 640 tensor cores
- Common L2 cache (6144 KB) for all multi processors
- HBM2 memory, typically 16 GB or 32 GB

GPU architecture: Nvidia Volta



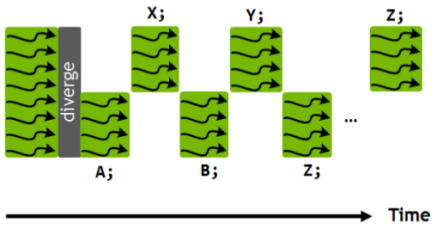
GPU architecture: Nvidia Volta SM

- 64 single precision cores
- 32 double precision cores
- 64 integer cores
- 8 Tensore cores
- 128 KB memory block for L1 and shared memory
  - 0 - 96 KB can be set to user managed shared memory
  - The rest is L1
- 65536 registers - enables the GPU to run a very large number of threads

GPU architecture: warps

- All execution is done in terms of 32 threads, a warp
- In a warp 32 threads compute the same instruction on different data (SIMT)
  - Warps are further collected into thread blocks; each executed on one SM
  - In case of divergence (if...) computation is done one branch at a time

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



Challenges in using Accelerators

Applicability: Is your algorithm suitable for GPU?

Programmability: Is the programming effort acceptable?

Portability: Rapidly evolving ecosystem and incompatibilities between vendors.

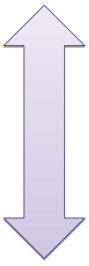
Availability: Can you access a (large scale) system with GPUs?

Scalability: Can you scale the GPU software efficiently to several nodes?

Using GPUs

1. Use existing GPU applications
2. Use accelerated libraries
3. Directive based methods
   - OpenMP
   - **OpenACC**
4. Use lower level language
   - CUDA
   - HIP
   - OpenCL

Easier, but more limited



More difficult, but more opportunities

Directive-based accelerator languages

- Annotating code to pinpoint accelerator-offloadable regions
- OpenACC standard created in Nov 2011
  - Focus on optimizing productivity (reasonably good performance with minimal effort)
  - Current standard is 3.0 (November 2019)
  - Mostly Nvidia only
- OpenMP
  - Earlier only threading for CPUs
  - 4.5 also includes for the first time some support for accelerators
  - 5.0 standard vastly improved
  - Dominant directive approach in the future?

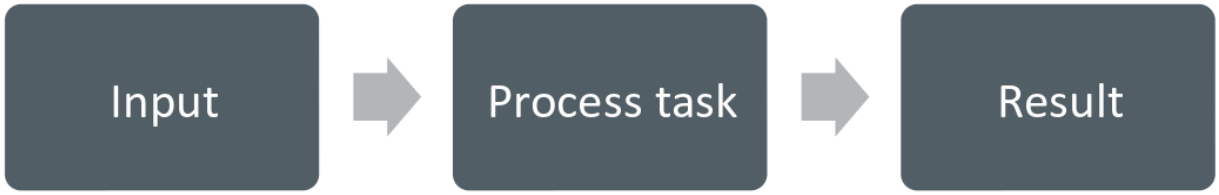
GPUs at CSC - Puhti-AI

- In total 80 nodes with a total peak performance of 2.7 Petaflops
- Each node has
  - Two latest generation Intel Xeon processors, code name Cascade Lake, with 20 cores each running at 2.1 GHz (Xeon Gold 6230)
  - Four Nvidia Volta V100 GPUs with 32 GB of memory each
  - 384 GB of main memory
  - 3.2 TB of fast local storage
  - Dual rail HDR100 interconnect network connectivity providing 200Gbps aggregate bandwidth

Parallel computing concepts

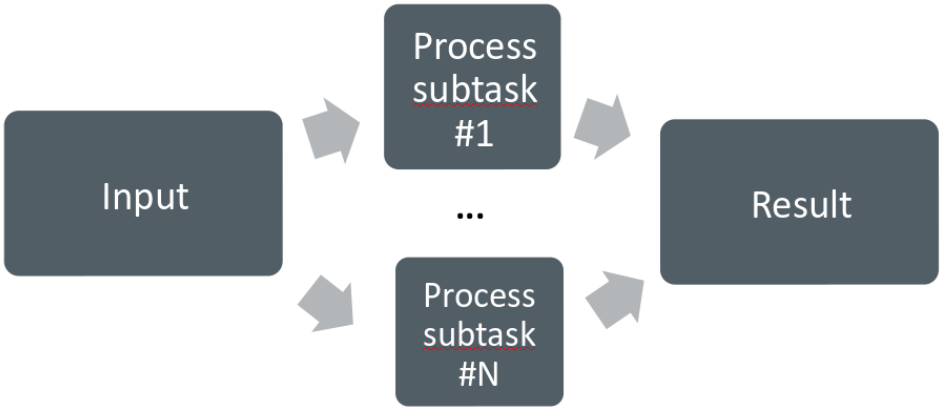
Computing in parallel

- Serial computing
  - Single processing unit ("core") is used for solving a problem



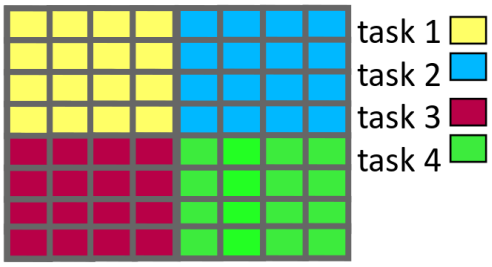
Computing in parallel

- Parallel computing
  - A problem is split into smaller subtasks
  - Multiple subtasks are processed *simultaneously* using multiple cores



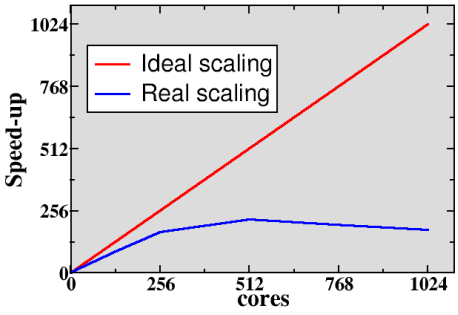
Exposing parallelism

- Data parallelism
  - Data is distributed to processor cores
  - Each core performs simultaneously (nearly) identical operations with different data
  - Especially good on GPUs(!)
- Task parallelism
  - Different cores perform different operations with (the same or) different data
- These can be combined



Parallel scaling

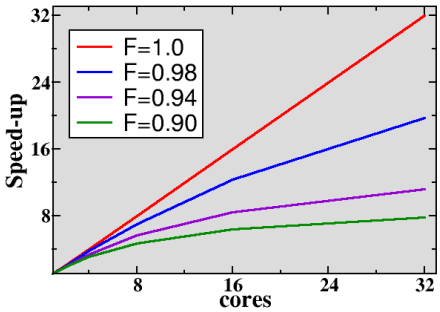
- Strong parallel scaling
  - Constant problem size
  - Execution time decreases in proportion to the increase in the number of cores
- Weak parallel scaling
  - Increasing problem size
  - Execution time remains constant when number of cores increases in proportion to the problem size



Amdahl's law

- Parallel programs often contain sequential parts
- *Amdahl's law* gives the maximum speed-up in the presence of non-parallelizable parts
- Main reason for limited scaling
- Maximum speed-up is $$S=\frac{1}{(1-F) + F/N}$$

where F is the parallel fraction and N is the number of cores



Parallel computing concepts

- Load balance
  - Distribution of workload to different cores
- Parallel overhead
  - Additional operations which are not present in serial calculation
  - Synchronization, redundant computations, communications

Summary

- HPC throughout the ages -- performance through parellelism
- Programming GPUs
  - CUDA, HIP
  - Directive based methods

OpenACC: introduction

What is OpenACC ?

- OpenACC defines a set of compiler directives that allow code regions to be offloaded from a host CPU to be computed on a GPU
  - High level GPU programming
  - Large similarity to OpenMP directives
- Supports for both C/C++ and Fortran bindings
- More about OpenACC standard: http://www.openacc.org

OpenACC vs. CUDA/HIP

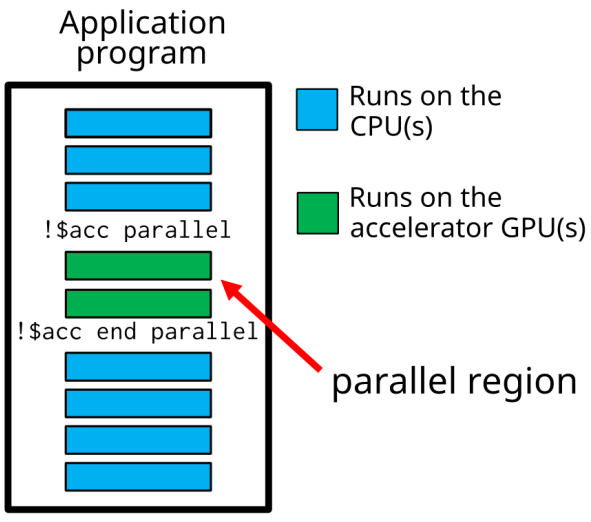
- Why OpenACC and not CUDA/HIP?
  - Easier to work with
  - Porting of existing software requires less work
  - Same code can be compiled to CPU and GPU versions easily
- Why CUDA/HIP and not OpenACC?
  - Can access all features of the GPU hardware
  - More optimization possibilities

OpenACC execution model

- Host-directed execution with an attached accelerator
  - Large part of the program is usually executed by the host
  - Computationally intensive parts are *offloaded* to the accelerator that executes *parallel regions*
- Accelerator can have a separate memory
  - OpenACC exposes the separate memories through *data environment* that defines the memory management and needed copy operations

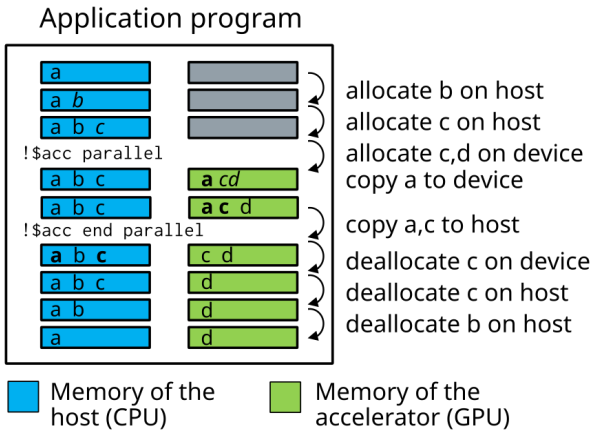
OpenACC execution model

- Program runs on the host CPU
- Host offloads compute-intensive regions (*kernels*) and related data to the accelerator GPU
- Compute kernels are executed by the GPU



OpenACC data model

- If host memory is separate from accelerator device memory
  - host manages memory of the device
  - host copies data to/from the device
- When memories are not separate, no copies are needed (difference is transparent to the user)



OpenACC directive syntax

	sentinel	construct	clauses
C/C++	`#pragma acc`	`kernels`	`copy(data)`
Fortran	`!$acc`	`kernels`	`copy(data)`

- OpenACC uses compiler directives for defining compute regions (and data transfers) that are to be performed on a GPU
- Important constructs
  - `parallel`, `kernels`, `data`, `loop`, `update`, `host_data`, `wait`
- Often used clauses
  - `if (condition)`, `async(handle)`

Compiling an OpenACC program

- Compilers that support OpenACC usually require an option that enables the feature
  - PGI: `-acc`
  - Cray: `-h acc`
  - GNU (partial support): `-fopenacc`
- Without these options a regular CPU version is compiled!

OpenACC conditional compilation

- Conditional compilation with `_OPENACC` macro:

```
#ifdef _OPENACC
device specific code
#else
host code
#endif
```

- `_OPENACC` macro is defined as *yyyymm*, where *yyyy* and *mm* refer to the year and month of when the specification supported by the compiler was released

OpenACC internal control variables

- OpenACC has internal control variables
  - `ACC_DEVICE_TYPE` controls which type of accelerator device is to be used.
  - `ACC_DEVICE_NUM` controls which accelerator of the selected type is used.
- During runtime, values can be modified or queried with `acc_<set|get>_device_<type|num>`
- Values are always re-read before a kernel is launched and can be different for different kernels

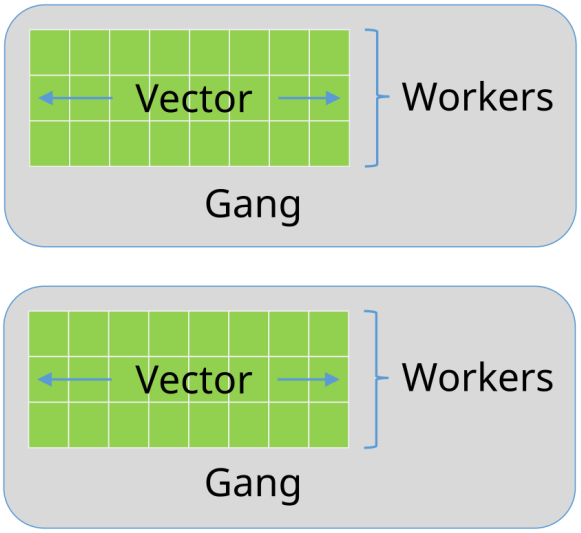
Runtime API functions

- Low-level runtime API functions can be used to
  - Query the number and type of devices in the system
  - Initialize/shutdown the device(s)
  - Allocate/deallocate memory on the device(s)
  - Transfer data to/from the device(s)
- Function definitions are in
  - C/C++ header file `openacc.h`
  - `openacc` Fortran module (`openacc_lib.h` header in some implementations)

OpenACC compute constructs

OpenACC levels of parallelism

- OpenACC has three levels of parallelism
  - **Vector** threads work in SIMT (SIMD) fashion
  - **Workers** compute a vector
  - **Gangs** have one or more workers that share resources, such as streaming multiprocessor
  - Multiple gangs work independently



OpenACC compute constructs

- OpenACC includes two different approaches for defining parallel regions
  - `parallel` defines a region to be executed on an accelerator. Work sharing *parallelism* has to be defined *manually*. Good tuning prospects.
  - `kernels` defines a region to be transferred into a series of kernels to be executed in *sequence* on an accelerator. Work sharing parallelism is defined *automatically* for the separate kernels, but tuning prospects limited.
- With similar work sharing, both can perform equally well

Compute constructs: `kernels`

- Define a region to be transferred to a sequence of kernels for execution on the accelerator device
  - C/C++: `#pragma acc kernels [clauses]`
  - Fortran: `!$acc kernels [clauses]`
- Each separate *loop nest* inside the region will be converted into a separate *parallel kernel*
- The *kernels* will be executed in a *sequential* order

Example: `kernels`

C/C++

```c
/* Compute y=a*x+y */
void accdaxpy(int n, double a,
              const double * restrict x,
              double * restrict y)
{
    #pragma acc kernels
    for (int j=0; j<n; ++j)
        y[j] += a * x[j];
}

...

/* An example call to accdaxpy */
accdaxpy(1<<16, 3.14, x, y);
```

Fortran

```fortran
! Compute y=a*x+y
subroutine accdaxpy(n, a, x, y)
  integer :: n, j
  real(kind=8) :: a, x(n), y(n)

  !$acc kernels
  do j = 1,n
     y(j) = y(j) + a * x(j)
  end do
  !$acc end kernels
end subroutine accdaxpy

...

! An example call to accdaxpy
call accdaxpy(65536, 3.14D0, x, y)
```

Compute constructs: `parallel`

- Define a region to be executed on the accelerator device
  - C/C++: `#pragma acc parallel [clauses]`
  - Fortran: `!$acc parallel [clauses]`
- Without any *work sharing* constructs, the whole region is executed *redundantly* multiple times
  - Given a sequence of loop nests, each loop nest may be executed simultaneously

Work sharing construct: `loop`

- Define a loop to be parallelized
  - C/C++: `#pragma acc loop [clauses]`
  - Fortran: `!$acc loop [clauses]`
  - Must be followed by a C/C++ or Fortran loop construct.
  - Combined constructs with `parallel` and `kernels`
    - `#pragma acc kernels loop` / `!$acc kernels loop`
    - `#pragma acc parallel loop` / `!$acc parallel loop`
- Similar in functionality to OpenMP `for/do` construct
- Loop index variables are `private` variables by default

Example: `parallel`

C/C++

```c
/* Compute y=a*x+y */
void accdaxpy(int n, double a,
              const double * restrict x,
              double * restrict y)
{
    #pragma acc parallel loop
    for (int j=0; j<n; ++j)
        y[j] += a * x[j];
}

...

/* An example call to accdaxpy */
accdaxpy(1<<16, 3.14, x, y);
```

Fortran

```fortran
! Compute y=a*x+y
subroutine accdaxpy(n, a, x, y)
  integer :: n, j
  real(kind=8) :: a, x(n), y(n)

  !$acc parallel loop
  do j = 1,n
     y(j) = y(j) + a * x(j)
  end do
  !$acc end parallel loop
end subroutine accdaxpy

...

! An example call to accdaxpy
call accdaxpy(65536, 3.14D0, x, y)
```

Compiler diagnostics

Compiler diagnostics

- Compiler diagnostics is usually the first thing to check when starting the OpenACC work
  - It can tell you what operations were actually performed
  - Data copies that were made
  - If and how the loops were parallelized
- The diagnostics is very compiler dependent
  - Compiler flags
  - Level and formatting of information

PGI compiler

- Diagnostics is controlled by compiler flag `-Minfo=option`
- Useful options:
  - `accel` -- operations related to the accelerator
  - `all` -- print all compiler output
  - `intensity` -- print loop computational intensity info
  - `ccff` -- add extra information to the object files for use by tools

Example: `-Minfo`

```
$ pgcc -fast -Minfo=all -c util.c
malloc_2d:
     28, Loop not vectorized: data dependency
         Loop unrolled 8 times
         Generated 1 prefetches in scalar loop
eval_point:
     38, Loop not vectorized/parallelized: potential early exits

$ pgcc -fast -Minfo=intensity -c util.c
malloc_2d:
     28, Intensity = 3.00
eval_point:
     38, Intensity = 8.00
```

Example: `-Minfo`

```
$ pgcc -acc -Minfo=all doubleloops.c
init:
     38, Memory zero idiom, loop replaced by call to __c_mzero8
     44, Memory set idiom, loop replaced by call to __c_mset8
main:
     74, Generating Tesla code
         77, #pragma acc loop gang /* blockIdx.x */
         79, #pragma acc loop vector(128) /* threadIdx.x */
     74, Generating implicit copyin(u[:1024][:1024]) [if not already present]
         Generating implicit copyout(unew[1:1022][1:1022]) [if not already present]
     79, Loop is parallelizable
     84, Generating Tesla code
         87, #pragma acc loop gang /* blockIdx.x */
         89, #pragma acc loop vector(128) /* threadIdx.x */
     84, Generating implicit copyin(unew[:1024][:1024]) [if not already present]
         Generating implicit copyout(u[1:1022][1:1022]) [if not already present]
     89, Loop is parallelizable
```

Summary

- OpenACC is an directive-based extension to C/Fortran programming languages for accelerators
- Supports separate memory on the accelerator
- Compute constructs: parallel and kernels
- Compiler diagnostics

OpenACC: data management

OpenACC data environment

- OpenACC supports devices which either share memory with or have a separate memory from the host
- Constructs and clauses for
  - defining the variables on the device
  - transferring data to/from the device
- All variables used inside the `parallel` or `kernels` region will be treated as *implicit* variables if they are not present in any data clauses, i.e. copying to and from to the device is automatically performed

Motivation for optimizing data movement

- When dealing with an accelerator / GPU device attached to a PCIe bus, **optimizing data movement** is often **essential** to achieving good performance
- The four key steps in porting to high performance accelerated code:
  1. Identify parallelism
  2. Express parallelism
  3. Express data movement
  4. Optimise loop performance
  5. Go back to 1!

Data lifetimes

- Typically data on the device has the same lifetime as the OpenACC construct (`parallel`, `kernels`, `data`) it is declared in
- It is possible to declare and refer to data residing statically on the device until deallocation takes place
- If the accelerator has a separate memory from the host, any modifications to the data on the host are not visible to the device before an explicit update has been made and vice versa

Data constructs: data

- Define a region with data declared in the device memory
  - C/C++: `#pragma acc data [clauses]`
  - Fortran: `!$acc data [clauses]`
- Data transfers take place
  - from the **host** to the **device** upon entry to the region
  - from the **device** to the **host** upon exit from the region
- Functionality defined by *data clauses*
- *Data clauses* can also be used in `kernels` and `parallel` constructs

Data construct: example

C/C++

```c
float a[100];
int iter;
int maxit=100;

#pragma acc data create(a)
{
    /* Initialize data on device */
    init(a);
    for (iter=0; iter < maxit; iter++) {
        /* Computations on device */
        acc_compute(a);
    }
}
```

Fortran

```fortran
real :: a(100)
integer :: iter
integer, parameter :: maxit = 100

!$acc data create(a)
! Initialize data on device
call init(a)
do iter=1,maxit
  ! Computations on device
  call acc_compute(a)
end do
!$acc end data
```

Data constructs: data clauses

`present(var-list)`

- **on entry/exit:** assume that memory is allocated and that data is present on the device

`create(var-list)`

- **on entry:** allocate memory on the device, unless it was already present
- **on exit:** deallocate memory on the device, if it was allocated on entry
  - in-depth: *structured* reference count decremented, and deallocation happens if both reference counts (*structured* and *dynamic*) are zero

Data constructs: data clauses

`copy(var-list)`

- **on entry:** if data is present on the device on entry, behave as with the `present` clause, otherwise allocate memory on the device and copy data from the host to the device.
- **on exit:** copy data from the device to the host and deallocate memory on the device if it was allocated on entry

Data constructs: data clauses

`copyin(var-list)`

- **on entry:** same as `copy` on entry
- **on exit:** deallocate memory on the device if it was allocated on entry

`copyout(var-list)`

- **on entry:** if data is present on the device on entry, behave as with the `present` clause, otherwise allocate memory on the device
- **on exit:** same as `copy` on exit

Data constructs: data clauses

`reduction(operator:var-list)`

- Performs reduction on the (scalar) variables in list
- Private reduction variable is created for each gang's partial result
  - initialised to operators initial value
- After parallel region the reduction operation is applied to the private variables and the result is aggregated to the shared variable *and* the aggregated result is combined with the original value of the variable

Reduction operators in C/C++ and Fortran

Arithmetic Operator	Initial value
`+`	`0`
`-`	`0`
`*`	`1`
`max`	least
`min`	largest

Reduction operators in C/C++ only

Logical Operator	Initial value
&&	1
||	0

Bitwise Operator	Initial value
&	~0
|	0
^	0

Reduction operators in Fortran

Logical Operator	Initial value
.and.	.true.
.or.	.false.
.eqv.	.true.
.neqv.	.false.

Bitwise Operator	Initial value
iand	all bits on
ior	0
ieor	0

Data specification

- Data clauses specify functionality for different variables
- Overlapping data specifications are not allowed
- For array data, *array ranges* can be specified
  - C/C++: `arr[start_index:length]`, for instance `vec[0:n]`
  - Fortran: `arr(start_index:end_index)`, for instance `vec(1:n)`
- Note: array data **must** be *contiguous* in memory (vectors, multidimensional arrays etc.)

Default data environment in compute constructs

- All variables used inside the `parallel` or `kernels` region will be treated as *implicit* variables if they are not present in any data clauses, i.e. copying to and from the device is automatically performed
- Implicit *array* variables are treated as having the `copy` clause in both cases
- Implicit *scalar* variables are treated as having the
  - `copy` clause in `kernels`
  - `firstprivate` clause in `parallel`

`data` construct: example

C/C++

```
int a[100], d[3][3], i, j;

#pragma acc data copy(a[0:100])
{
    #pragma acc parallel loop present(a)
    for (int i=0; i<100; i++)
        a[i] = a[i] + 1;
    #pragma acc parallel loop \
          collapse(2) copyout(d)
    for (int i=0; i<3; ++i)
        for (int j=0; j<3; ++j)
            d[i][j] = i*3 + j + 1;
}
```

Fortran

```
integer a(0:99), d(3,3), i, j

!$acc data copy(a(0:99))
  !$acc parallel loop present(a)
  do i=0,99
     a(i) = a(i) + 1
  end do
  !$acc end parallel loop
  !$acc parallel loop collapse(2) copyout(d)
  do j=1,3
     do i=1,3
        d(i,j) = i*3 + j + 1
     end do
  end do
  !$acc end parallel loop
!$acc end data
```

Unstructured data regions

- Unstructured data regions enable one to handle cases where allocation and freeing is done in a different scope
- Useful for e.g. C++ classes, Fortran modules
- `enter data` defines the start of an unstructured data region
  - C/C++: `#pragma acc enter data [clauses]`
  - Fortran: `!$acc enter data [clauses]`
- `exit data` defines the end of an unstructured data region
  - C/C++: `#pragma acc exit data [clauses]`
  - Fortran: `!$acc exit data [clauses]`

Unstructured data

```
class Vector {
    Vector(int n) : len(n) {
        v = new double[len];
        #pragma acc enter data create(v[0:len])
    }
    ~Vector() {
        #pragma acc exit data delete(v[0:len])
        delete[] v;
    }
    double v;
    int len;
};
```

Enter data clauses

`if(condition)`

- Do nothing if condition is false

`create(var-list)`

- Allocate memory on the device

`copyin(var-list)`

- Allocate memory on the device and copy data from the host to the device

Exit data clauses

`if(condition)`

- Do nothing if condition is false

`delete(var-list)`

- Deallocate memory on the device
  - in-depth: *dynamic* reference count decremented, and deallocation happens if both reference counts (*dynamic* and *structured*) are zero

`copyout(var-list)`

- Deallocate memory on the device and copy data from the device to the host
  - in-depth: *dynamic* reference count decremented, and deallocation happens if both reference counts (*dynamic* and *structured*) are zero

Data directive: update

- Define variables to be updated within a data region between host and device memory
  - C/C++: `#pragma acc update [clauses]`
  - Fortran: `!$acc update [clauses]`
- Data transfer direction controlled by `host(var-list)` or `device(var-list)` clauses
  - `self` (`host`) clause updates variables from device to host
  - `device` clause updates variables from host to device
- At least one data direction clause must be present

Data directive: update

- `update` is a single line executable directive
- Useful for producing snapshots of the device variables on the host or for updating variables on the device
  - Pass variables to host for visualization
  - Communication with other devices on other computing nodes
- Often used in conjunction with
  - Asynchronous execution of OpenACC constructs
  - Unstructured data regions

`update` directive: example

C/C++

```
float a[100];
int iter;
int maxit=100;

#pragma acc data create(a) {
    /* Initialize data on device */
    init(a);
    for (iter=0; iter < maxit; iter++) {
        /* Computations on device */
        acc_compute(a);
        #pragma acc update self(a) \
                if(iter % 10 == 0)
    }
}
```

Fortran

```
real :: a(100)
integer :: iter
integer, parameter :: maxit = 100

!$acc data create(a)
    ! Initialize data on device
    call init(a)
    do iter=1,maxit
        ! Computations on device
        call acc_compute(a)
        !$acc update self(a)
        !$acc& if(mod(iter,10)==0)
    end do
!$acc end data
```

Data directive: declare

- Makes a variable resident in accelerator memory
- Added at the declaration of a variable
- Data life-time on device is the implicit life-time of the variable
  - C/C++: `#pragma acc declare [clauses]`
  - Fortran: `!$acc declare [clauses]`
- Supports usual data clauses, and additionally
  - `device_resident`
  - `link`

Porting and managed memory

- Porting a code with complicated data structures can be challenging because every field in type has to be copied explicitly
- Recent GPUs have *Unified Memory* and support for page faults

```
typedef struct points {
    double x, y;
    int n;
}

void init_point() {
    points p;

    #pragma acc data create(p)
    {
        p.size = n;
        p.x = (double)malloc(...
        p.y = (double)malloc(...
        #pragma acc update device(p)
        #pragma acc copyin (p.x[0:n]...
```

Managed memory

- Managed memory copies can be enabled on PGI compilers
  - Pascal (P100): `--ta=tesla,cc60,managed`
  - Volta (V100): `--ta=tesla,cc70,managed`
- For full benefits Pascal or Volta generation GPU is needed
- Performance depends on the memory access patterns
  - For some cases performance is comparable with explicitly tuned versions

Summary

- Data directive
  - Structured data region
  - Clauses: `copy`, `present`, `copyin`, `copyout`, `create`
- Enter data & exit data
  - Unstructured data region
- Update directive
- Declare directive

Profiling and performance optimisation

Outline

- NVidia profiling tools
- Tuning compute region performance
  - Enabling vectorization
  - Loop optimizations
  - Branches
  - Memory access

Profiling tools

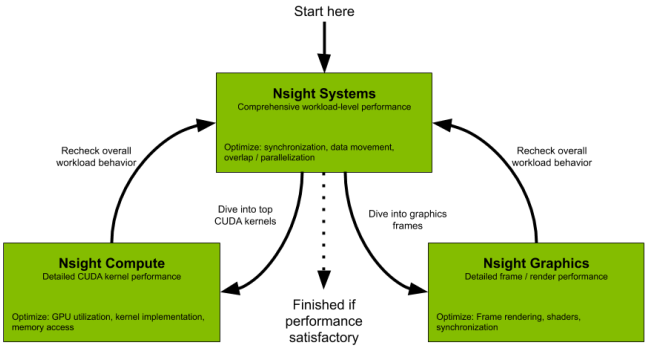
NVIDIA NVPROF profiler

- NVIDIA released the Nsight recently for profiling. One of the main reasons for the new tool is scalability and, of course, new features
  - It is included with CUDA since 10.x
- GPU profiling capabilities
  - High-level usage statistics
  - Timeline collection
  - Analysis metrics

OpenACC example

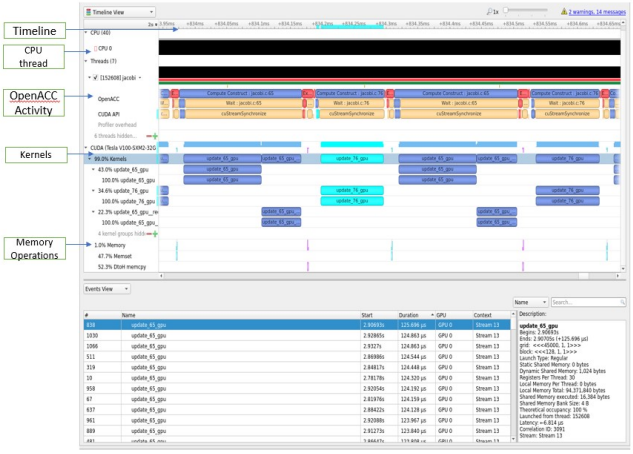
```
$ nsys profile -t nvtx,openacc --stats=true -s cpu ./jacobi
...
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  -------------
   77.1       108005341        1450         74486.4            1815          189044  cuStreamSynch
   16.6        23232332           1      23232332.0        23232332        23232332  cuMemHostAllo
    2.9         4037274        1091          3700.5            2815           24189  cuLaunchKerne
    0.8         1182112         361          3274.5            2911           21123  cuMemcpyDtoHA
    0.6          855308         361          2369.3            2065           10502  cuMemsetD32As
    0.6          813341           1        813341.0          813341          813341  cuMemAllocHos

Generating CUDA Kernel Statistics...
CUDA Kernel Statistics (nanoseconds)

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
   43.2        43864453         361        121508.2          118208          124671  update_65_gpu
```

NVIDIA Nsight workflow



Source: NVIDIA

NVIDIA Systems



NVIDIA Metrics

```
srun -n 1 nv-nsight-cu-cli --devices 0 --query-metrics  > my_metrics.txt

dram__bytes                 # of bytes accessed in DRAM
dram__bytes_read            # of bytes read from DRAM
dram__bytes_write           # of bytes written to DRAM
dram__cycles_active         # of cycles where DRAM was active
...
tpc__cycles_active          # of cycles where TPC was active
tpc__cycles_elapsed         # of cycles where TPC was active
tpc__cycles_in_frame        # of cycles in user-defined frame
tpc__cycles_in_region       # of cycles in user-defined region
...
```

Nsight Compute CLI (I)

```
srun -n 1  nv-nsight-cu-cli ./jacobi
...
  update_65_gpu, 2020-Oct-18 23:42:06, Context 1, Stream 13
   Section: GPU Speed Of Light
    ---------------------------------------------------------------------- --------------- -------
    DRAM Frequency                                                         cycle/usecond
    SM Frequency                                                           cycle/nsecond
    Elapsed Cycles                                                                 cycle
    Memory [%]                                                                         %
    SOL DRAM                                                                           %
    Duration                                                                      usecond
    SOL L1/TEX Cache                                                                   %
    SOL L2 Cache                                                                       %
    SM Active Cycles                                                               cycle
    SM [%]                                                                             %
    ---------------------------------------------------------------------- --------------- -------
    WRN   Memory is more heavily utilized than Compute: Look at the Memory Workload Analysis repor
          where the memory system bottleneck is. Check memory replay (coalescing) metrics to make
```

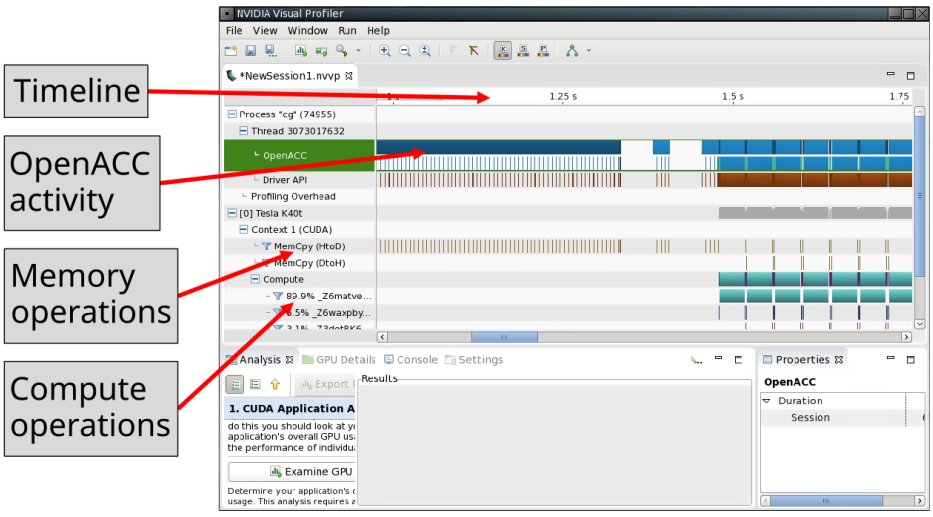
Nsight Compute CLI (II)

```
  Section: Launch Statistics
  ---------------------------------------------------------------------- --------------- ------------------------------
  Block Size                                                                                                        128
  Grid Size                                                                                                      45,000
  Registers Per Thread                                                   register/thread                             30
  Shared Memory Configuration Size                                             Kbyte                          16.38
  Driver Shared Memory Per Block                                          byte/block                              0
  Dynamic Shared Memory Per Block                                        Kbyte/block                           1.02
  Static Shared Memory Per Block                                          byte/block                              0
  Threads                                                                      thread                      5,760,000
  Waves Per SM                                                                                                  35.16
  ---------------------------------------------------------------------- --------------- ------------------------------

  Section: Occupancy
  ---------------------------------------------------------------------- --------------- ------------------------------
  Block Limit SM                                                               block                             32
  Block Limit Registers                                                        block                             16
  Block Limit Shared Mem                                                       block                             96
  Block Limit Warps                                                            block                             16
  Theoretical Active Warps per SM                                               warp                             64
  Theoretical Occupancy                                                            %                            100
  Achieved Occupancy                                                               %                          83.71
  Achieved Active Warps Per SM                                                  warp                          53.58
  ---------------------------------------------------------------------- --------------- ------------------------------
```

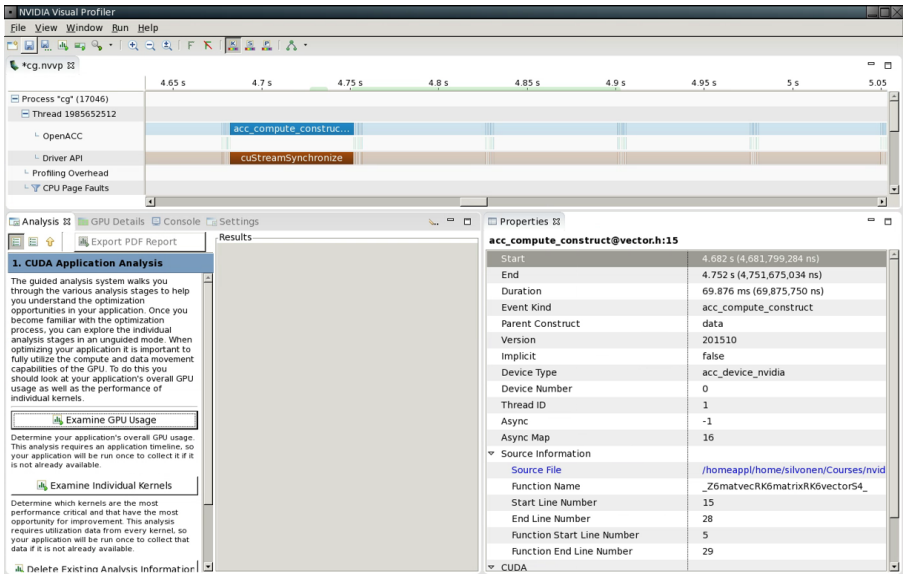
- Nsight Compute GUI does not support OpenACC

NVIDIA visual profiler

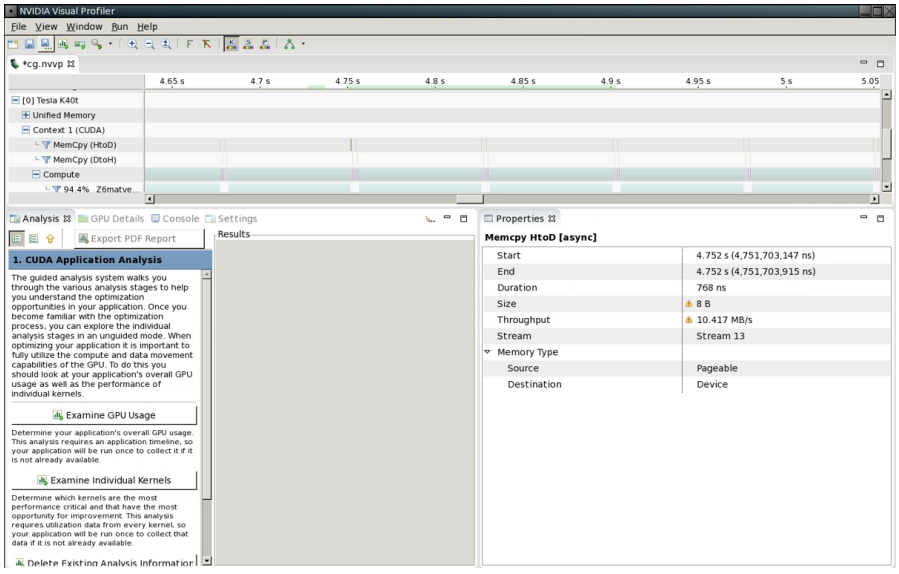
- Nvprof is an older profiling tool



Details on OpenACC compute construct



Details on memory copy



Optimization

Compute optimizations

- Data movement is and important part to optimize when using GPUs
  - Keeping data on the GPU as long as possible
- Getting the compiler to generate parallel code
  - Addressing loop dependencies
- Data access and execution divergence are important for GPU performance

Loop dependencies

```c
/* FLOW dependency, k>0 */
for (int i=0; i<N; i++)
    A[i] = A[i-k]+1;
/* ANTI dependency, k>0 */
for (int i=0; i<N; i++)
    A[i] = A[i+k]+1;
```

```fortran
! FLOW dependency, k>0
do i=0, N
    a(i) = a(i-k) + 1;
end do
! ANTI dependency, k>0
do i=0, N
    a(i) = a(i+k) + 1;
end do
```

- FLOW dependency
  - Read After Write (RAW), data is written to is read on the following iteration round(s)
- ANTI dependency
  - Write After Read (WAR), data read is written to on the following iteration rounds

Loop dependencies

- Dependencies disable vectorization, which is essential for good performance
- Rewrite the loops so that the dependency is removed
- Try to split the loop, use temporary array, etc.
- Some dependencies can not be removed
  - Try a different algorithm?

Loop dependencies and C

- C pointers are hard for the compiler to follow
  - Compiler will not know, if a loop can be vectorized safely, if a function has pointer arguments
  - Can be a *false* dependency

```
void adder(float *x, float *y, float *res) {
    for (int i=0; i < VECSIZE; i++) {
        res[i] = x[i] + y[i];
    }
}
```

- What if `res` and `x` overlap in memory?

C99 restrict keyword

- C99 standard has restrict keyword which tells the compiler that the pointer is accessed so that it does not overlap with other accesses

```
void adder(float restrict *x, float restrict *y, float restrict *res) {
    for (int i=0; i < VECSIZE; i++) {
        res[i] = x[i] + y[i];
    }
}
```

Loop independent clause

- OpenACC independent clause tells to the compiler that loop iterations are independent
  - Overrides any compiler dependency analysis
  - You have to make sure that the iterations are independent!

```
#pragma acc loop independent
void adder(float *x, float *y, float *res) {
    for (int i=0; i < VECSIZE; i++) {
        res[i] = x[i] + y[i];
    }
}
```

Loop directive

- Loop directive accepts several fine-tuning clauses
  - **gang** -- apply gang-level parallelism
  - **worker** -- apply worker-level parallelism
  - **vector** -- apply vector-level parallelism
  - **seq** -- run sequentially
- Multiple levels can be applied to a loop nest, but they have to be applied in top-down order

Optimize loops: vector length

- Tell the compiler that when using NVIDIA device it should use a vector length of 32 on the innermost loop
- Because these parameters depend on the accelerator type, it is a good practice to add **device_type** clause

```
for (int i=0; i<imax; i++) {
    ...
    #pragma acc loop device_type(nvidia) vector(32)
    for (int j=0; j<jmax; j++) {
        ... /* No further loops in this block */
    }
}
```

Optimize loops: specifying workers

```
#pragma acc loop device_type(nvidia) gang worker(32)
for (int i=0; i<imax; i++) {
    ...
    #pragma acc loop device_type(nvidia) vector(32)
    for (int j=0; j<jmax; j++) {
        ...
    }
}
```

- Tell the compiler that when using NVIDIA device, the outer loop should be broken over gangs and workers with 32 workers per gang

Additional loop optimizations

- `collapse(N)`
  - Same as in OpenMP, take the next N tightly nested loops and flatten them into a one loop
  - Can be beneficial when loops are small
  - Breaks the next loops into tiles (blocks) before parallelizing the loops
  - For certain memory access patterns this can improve data locality

What values should I try?

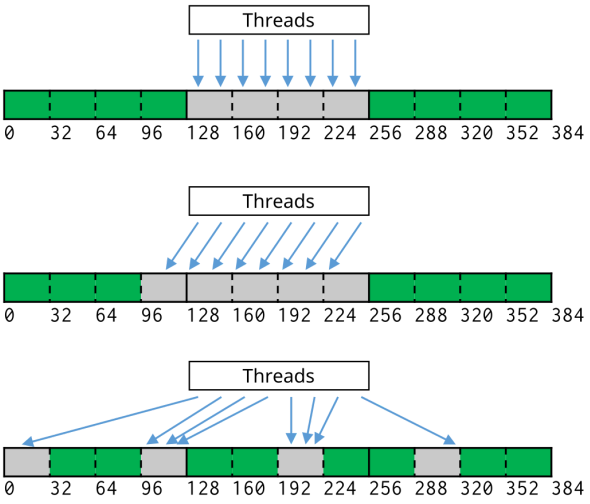
- Depends on the accelerator you are using
- You can try out different combinations, but deterministic optimizations require good knowledge on the accelerator hardware
  - In the case of NVIDIA GPUs you should start with the NVVP results and refer to CUDA documentation
  - One hard-coded value: for NVIDIA GPUs the vector length should always be 32, which is the (current) warp size

Branches in device code

- 32 threads running the same instruction at the same time
- Avoid branches based on thread id unless evenly dividable by 32
  - If (i%2) NO!
  - if (i%32) ok
- When unavoidable keep branches short

Coalesced memory access

- Coalesced memory access
  - 32 threads accessing memory at the same time
  - 32 Byte access granularity
- Overly simplified
  - Some cases 128 bytes access granularity
  - 128 byte coalesced accesses can improve performance



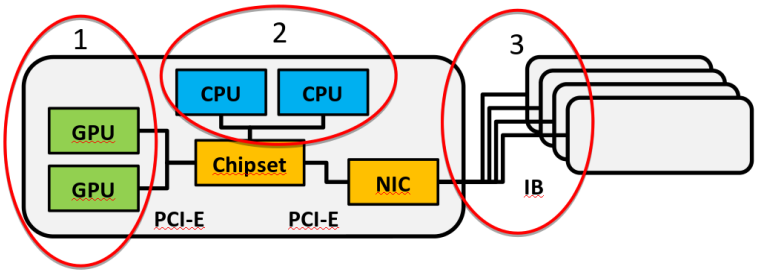
Summary

- Profiling is essential for optimization
  - NVPROF and NVVP for NVIDIA platform
- Loop optimizations
- Branches
- Memory access patterns

OpenACC: multi-GPU programming

Multi-GPU programming with OpenACC

- Three levels of hardware parallelism in a supercomputer:
  1. GPU - different levels of threads
  2. Node - multiple GPUs and CPUs
  3. System - multiple nodes connected with interconnect
- Three parallelization methods:
  1. OpenACC
  2. OpenMP or MPI
  3. MPI between nodes



Multi-GPU communication cases

- Single node multi-GPU programming
  - All GPUs of a node are accessible from a single process and its OpenMP threads
  - Data copies either directly or through CPU memory
- Multi-node multi-GPU programming
  - Communication between nodes requires message passing (MPI)
- In this lecture we will discuss in detail only parallelization with MPI
  - It enables direct scalability from single to multi-node

Multiple GPUs

- OpenACC permits using multiple GPUs within one node by using the `acc_get_num_devices` and `acc_set_device_num` functions
- Asynchronous OpenACC calls, OpenMP threads or MPI processes must be used in order to actually run kernels in parallel
- Issue when using MPI:
  - If a node has more than one GPU, all processes in the node can access all GPUs of the node
  - MPI processes do not have any a priori information about the other ranks in the same node
  - Which GPU the MPI process should select?

Selecting a device with MPI

- Model is to use **one** MPI task per GPU
- Launching job
  - Launch you application so that there are as many MPI tasks per node as there are GPUs
  - Make sure the affinity is correct - processes equally split between the two sockets (that nodes typically have)
  - Read the user guide of the system for details how to do this!
- In the code a portable and robust solution is to use MPI3 shared memory communicators to split the GPUs between processes
- Note that you can also use OpenMP to utilize all cores in the node for computations on CPU side

Selecting a device with MPI

```
MPI_Comm shared;
int local_rank, local_size, num_gpus;

MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0,
                    MPI_INFO_NULL, &shared);
MPI_Comm_size(shared, &local_size); // number of ranks in this node
MPI_Comm_rank(shared, &local_rank); // my local rank
num_gpus = acc_get_num_device(acc_device_nvidia); // num of gpus in node
if (num_gpus == local_size) {
    acc_set_device_num(local_rank);
} // otherwise error
```

Data transfers

- Idea: use MPI to transfer data between GPUs, use OpenACC-kernels for computations
- Additional complexity: GPU memory is separate from CPU memory
- GPU-aware MPI-library
  - Can use the device pointer in MPI calls - no need for additional buffers
  - No need for extra buffers and device-host-device copies
  - If enabled on system, data will be transferred via transparent RDMA
- Without GPU-aware MPI-library
  - Data must be transferred from the device memory to the host memory and vice versa before performing MPI-calls

Using device addresses with host_data

- For accessing device addresses of data on the host OpenACC includes `host_data` construct with the `use_device` clause
- No additional data transfers needed between the host and the device, data automatically accessed from the device memory via **R**emote **D**irect **M**emory **A**ccess
- Requires *library* and *device* support to function!

MPI communication with GPU-aware MPI

- MPI send
  - Send the data from the buffer on the **device** with MPI
- MPI receive
  - Receive the data to a buffer on the **device** with MPI
- No additional buffers or data transfers needed to perform communication

MPI communication with GPU-aware MPI

```
/* MPI_Send with GPU-aware MPI */
#pragma acc host_data use_device(data)
{
    MPI_Send(data, N, MPI_DOUBLE, to, MPI_ANY_TAG, MPI_COMM_WORLD);
}

/* MPI_Recv with GPU-aware MPI */
#pragma acc host_data use_device(data)
{
    MPI_Recv(data, N, MPI_DOUBLE, from, MPI_ANY_TAG, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}
```

Summary

- Typical HPC cluster node has several GPUs in each node
  - Selecting the GPUs with correct affinity
- Data transfers using MPI
  - GPU-aware MPI avoids extra memory copies