Deeper dive into GPUs, debugging and profiling with NVIDIA tools

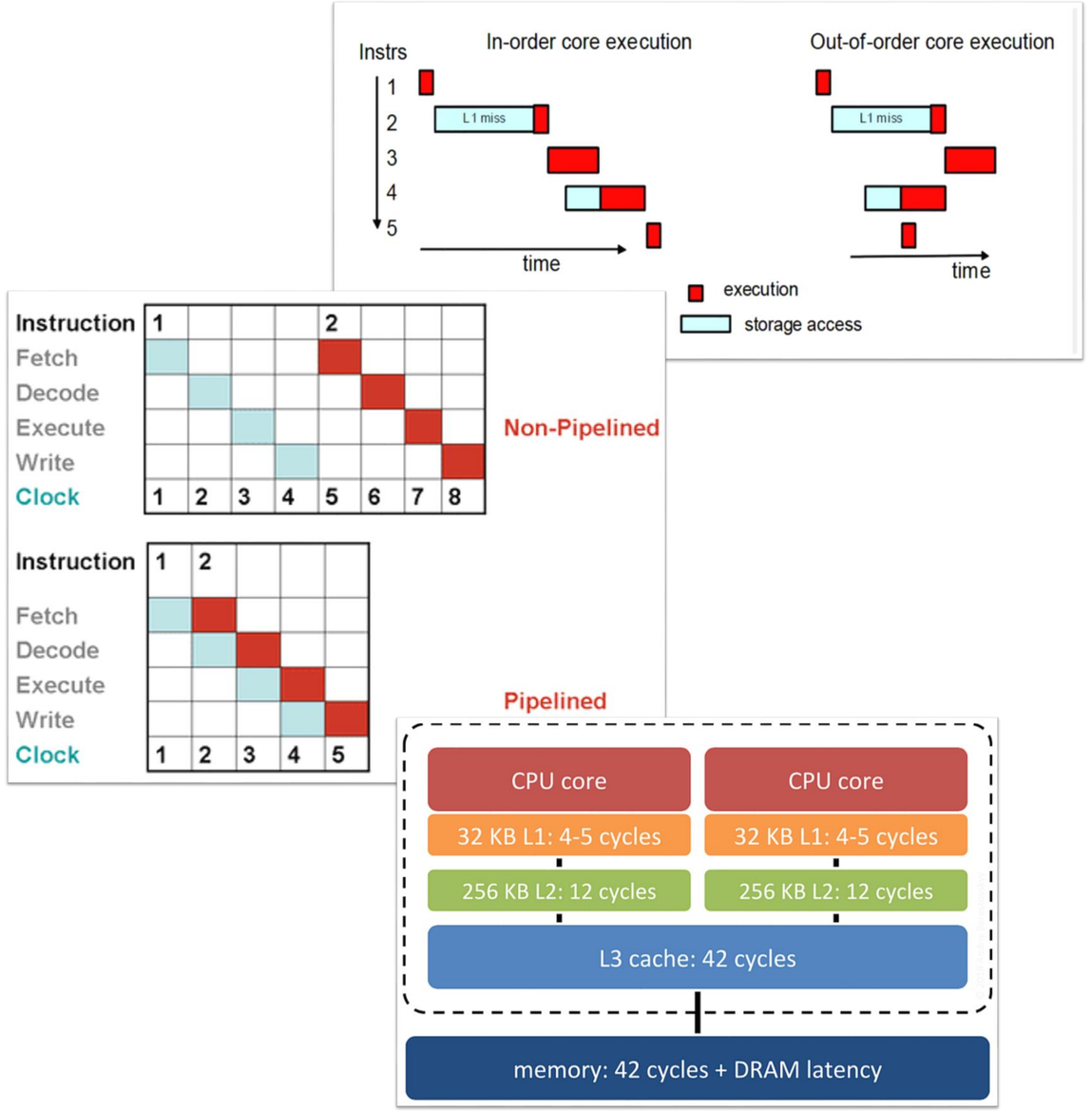
Dr. Dmitry Alexeev, NVIDIA

07/2024

Why the GPUs are the way they are?

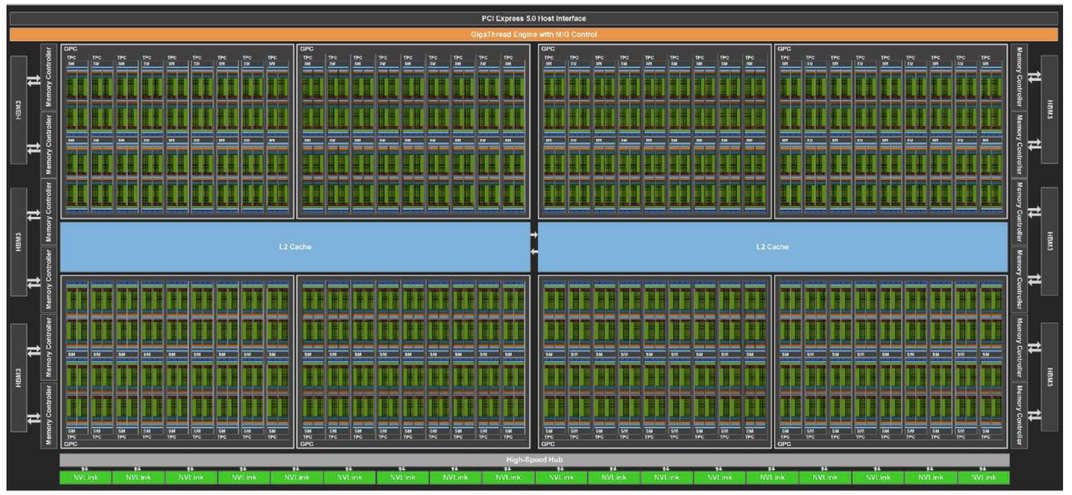
First the CPUs

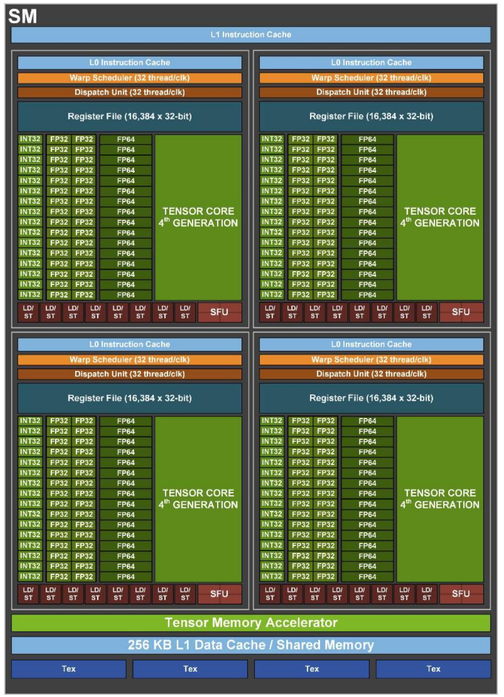
- A CPU (core) wants to execute a **stream** of instructions as fast as possible (think IPC)
- Every instruction takes a few cycles, 5-50-500 or more
- How can we reduce the latency?
  - Pipelining!
- But what if pipelining is not possible?
  - Instructions with too much latency (memory accesses) => cache
  - Data dependency => out-of-order execution
  - Branching => speculative execution and branch predictions
  - etc. etc. etc
- Now the CPU core has a very large **frontend**, and can execute instructions very very fast
- What else do we need?



Now the GPUs

- Can we somehow save transistors on the **frontend** and use them instead for the **backend**?
- Yes! But only for certain tasks
- The GPU **assumes** that the running task has a lot of data parallelism, i.e., very many items that can be processed *almost* independently
- But if we just multiply a CPU core and make each one process its own elements, we'll still need a wide frontend to reduce latency
- Instead, we'll use **oversubscription**

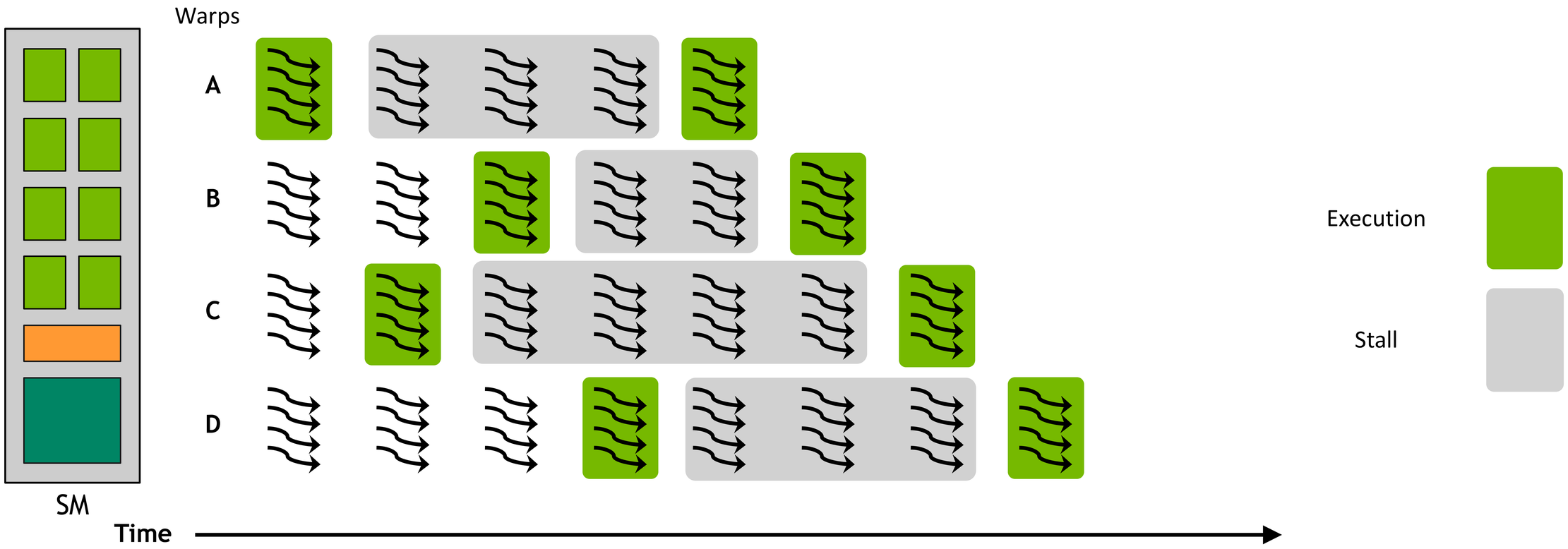




Now the GPUs

Oversubscription

- The idea is that we **additionally assume** that there are much more independent items than the hardware execution units
- Therefore, the execution units can switch between the different items in case of stalls
- Fast switching means that HW needs to store the state of each item it works on in fast memory: registers and cache
- These resources are limited, hence **occupancy**



Now the GPUs

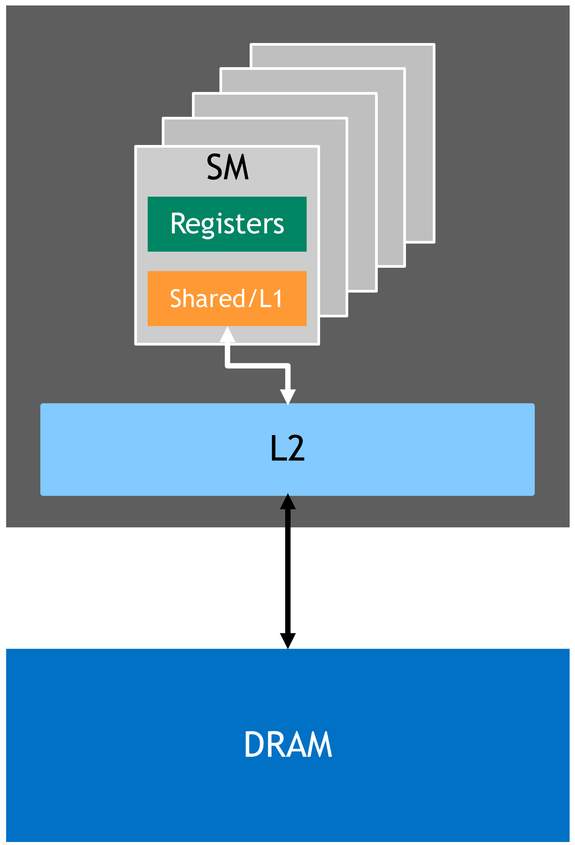
SMs

- Streaming multiprocessor is like a “core” of the GPU, and it implements the idea of oversubscription
- No OOO execution or branch prediction, frontend is very simple and predictable
- To save more frontend space: 1 instruction per 32 data elements, or **threads**. This is what’s called a **warp** and the concept is similar to CPU SIMD instructions, however, more flexible
- We need quite a few registers and cache, or shared memory, per each backend execution unit to enable fast context switching
- Added benefit: warps that reside on the same SM are physically close to each other and can synchronize/communicate quickly. This is exposed in CUDA as **thread blocks**
- Modern GPUs have more complexity with SM clusters, tensor cores connected to multiple SMs, separate instruction pointer per thread, etc. I only touched the basics!

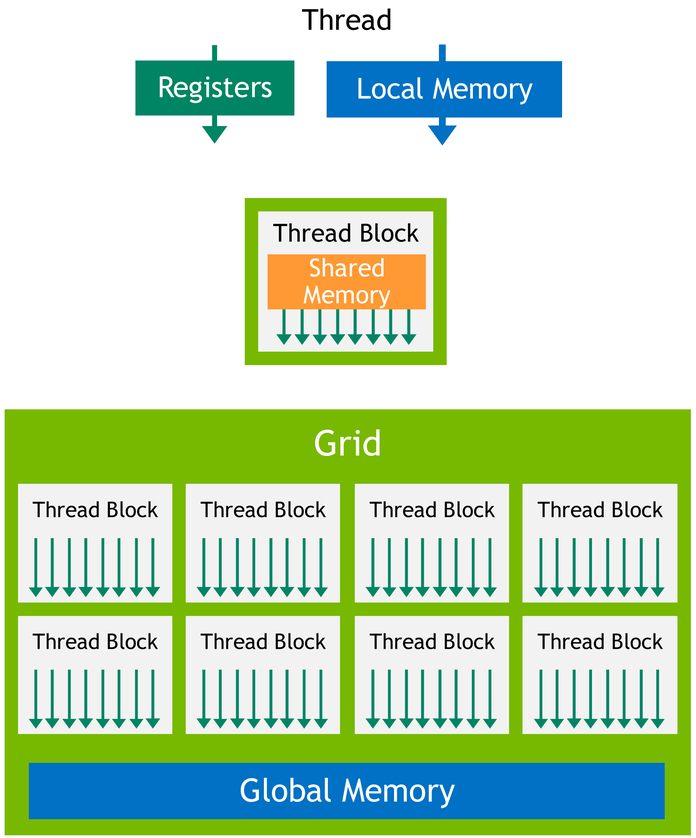
Now the GPUs

Memory hierarchy, because DRAM latency and bandwidth are still not great

Hardware



CUDA/Software



- Per-thread **registers**.
  - Lowest possible latency.
- Per-thread **local** memory.
  - Private storage.
  - Backed up by **global** memory (hence the color).
- Per-block **shared** memory.
  - Visible by all threads in a block.
  - Can be used to exchange data between threads in a thread block.
  - Very fast access.
  - Can serve as **L1** cache.
- **Global** memory.
  - Visible by all threads in a grid.
  - Slowest access.
  - Augmented by **L2** shared across all SMs and per-SM **L1**

How to program the GPUs with OpenACC?

OpenACC tips and tricks

- There is **no silver bullet**! All codes, algorithms and patterns are different
- In many cases GPU optimizations may help CPU performance as well
- Data layout is the king, data access is the queen (or vise versa, both are equally important and need to match)
- Vector lanes (*threads* in CUDA) should read contiguous chunks of data. This is called **coalesced access**

```
!$acc parallel loop gang vector
do iCell=cellSolveStart,cellSolveEnd
   !$acc loop seq
   do k=2,nVertLevels
      rw_p(k,iCell) = (rw_p(k,iCell)-a_tri(k,iCell)* &
                  rw_p(k-1,iCell))*alpha_tri(k,iCell)
   end do


   !$acc loop seq
   do k=nVertLevels,1,-1
      rw_p(k,iCell) = rw_p(k,iCell) - gamma_tri(k,iCell)*rw_p(k+1,iCell)
   end do
end do
!$acc end parallel
```

10x faster

```
!$acc parallel loop gang vector
do iCell=cellSolveStart,cellSolveEnd
   !$acc loop seq
   do k=2,nVertLevels
      rw_p(iCell,k) = (rw_p(iCell,k)-a_tri(iCell,k)* &
                  rw_p(iCell,k-1))*alpha_tri(iCell,k)
   end do


   !$acc loop seq
   do k=nVertLevels,1,-1
      rw_p(iCell,k) = rw_p(iCell,k) - gamma_tri(iCell,k)*rw_p(iCell,k+1)
   end do
end do
!$acc end parallel
```

OpenACC tips and tricks

- Remember oversubscription: we want to expose as much parallelism as possible. Total number of loop iterations, parallelized with *gang* and vector clauses, should ideally be higher than 100’000 or so.
- Data movements to and from the GPU could be costly and it’s better to keep all the data and all the computations on the GPU whenever possible
- However, partially ported codes *always* have some copies, and typically you should only remove them once the whole code path of interest is on the GPU
- Nvidia HPC compiler has an info flag: -Minfo=accel or -Minfo=all. This output can be helpful: tells you which loops are parallelized, and which are not, if private variables end up in registers or shared memory, etc.
- You can call functions and subroutines from ACC parallel regions. If such functions are in the same source file, performance should be good. If they are in another source file, consider using inlining and inline libraries: https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html#fn-inline-use
- However, local arrays in device functions are (currently) very inefficient!

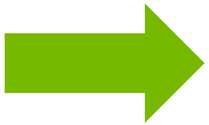
```
do i=1, N
   result(i) = process(input(i))
end do
```

```
function process(input)
   real :: input
   real :: process
   real :: temporary(global_size)

   ...
end function
```

OpenACC tips and tricks

Loop fusion

```
!$ACC PARALLEL DEFAULT(NONE) ASYNC(1)
!$ACC LOOP GANG VECTOR
DO jk = 1, kbdim
  zwindspeed10m_lnd(jk)     = 0.8_wp * zwindspeed_lnd(jk)
END DO
!$ACC END PARALLEL


!$ACC PARALLEL DEFAULT(NONE) ASYNC(1)
!$ACC LOOP GANG VECTOR
DO jl = jcs,kproma
  IF (rpds(jl) > 0._wp) THEN
    fract_par_diffuse(jl) = rpds_dif(jl) / rpds(jl)
  ELSE
    fract_par_diffuse(jl) = 0._wp
  END IF
END DO
!$ACC END PARALLEL
```



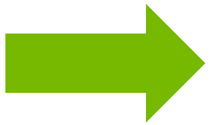
```
!$ACC PARALLEL DEFAULT(NONE) ASYNC(1)
!$ACC LOOP GANG(static:1) VECTOR
DO jk = 1, kbdim
  zwindspeed10m_lnd(jk)     = 0.8_wp * zwindspeed_lnd(jk)
END DO

!$ACC LOOP GANG(static:1) VECTOR
DO jl = jcs,kproma
  IF (rpds(jl) > 0._wp) THEN
    fract_par_diffuse(jl) = rpds_dif(jl) / rpds(jl)
  ELSE
    fract_par_diffuse(jl) = 0._wp
  END IF
END DO
!$ACC END PARALLEL
```

- Avoid GPU kernel launch latency

OpenACC tips and tricks

Loop fusion

```
!$ACC PARALLEL DEFAULT(NONE) ASYNC(1)
!$ACC LOOP GANG VECTOR COLLAPSE(2)
DO jk = 1, nlev
  DO je = i_startidx, i_endidx
    nabv_tang = ...
    nabv_norm = ...
    z_nabla4_e2(je,jk) = 4._wp * (                          &
      (nabv_norm - 2._wp*z_nabla2_e(je,jk,jb))              &
      *p_patch%edges%inv_vert_vert_length(je,jb)**2 +       &
      (nabv_tang - 2._wp*z_nabla2_e(je,jk,jb))              &
      *p_patch%edges%inv_primal_edge_length(je,jb)**2 )
  ENDDO
ENDDO
!$ACC END PARALLEL
!$ACC PARALLEL DEFAULT(NONE) ASYNC(1)
!$ACC LOOP GANG VECTOR COLLAPSE(2)
DO jk = 1, nlev
  DO je = i_startidx, i_endidx
    p_nh_prog%vn(je,jk,jb) = p_nh_prog%vn(je,jk,jb)  +                                  &
      p_patch%edges%area_edge(je,jb) *                                                  &
      (MAX(nudgezone_diff*p_int%nudgecoeff_e(je,jb),REAL(kh_smag_e(je,jk,jb),wp))* &
      z_nabla2_e(je,jk,jb) - diff_multfac_vn(jk) * z_nabla4_e2(je,jk)   *           &
      p_patch%edges%area_edge(je,jb))
  ENDDO
END
!$ACC END PARALLEL
```



```
!$ACC PARALLEL DEFAULT(NONE) ASYNC(1) IF( i_am_accel_node .AND. acc_on )
!$ACC LOOP GANG(static:1) VECTOR COLLAPSE(2)
DO jk = 1, nlev
  DO je = i_startidx, i_endidx
    nabv_tang = ...
    nabv_norm = ...
    z_nabla4_e2(je,jk) = 4._wp * (                          &
      (nabv_norm - 2._wp*z_nabla2_e(je,jk,jb))              &
      *p_patch%edges%inv_vert_vert_length(je,jb)**2 +       &
      (nabv_tang - 2._wp*z_nabla2_e(je,jk,jb))              &
      *p_patch%edges%inv_primal_edge_length(je,jb)**2 )
  ENDDO
ENDDO

!$ACC LOOP GANG(static:1) VECTOR COLLAPSE(2)
DO jk = 1, nlev
  DO je = i_startidx, i_endidx
    p_nh_prog%vn(je,jk,jb) = p_nh_prog%vn(je,jk,jb)  +                                  &
      p_patch%edges%area_edge(je,jb) *                                                  &
      (MAX(nudgezone_diff*p_int%nudgecoeff_e(je,jb),REAL(kh_smag_e(je,jk,jb),wp))* &
      z_nabla2_e(je,jk,jb) - diff_multfac_vn(jk) * z_nabla4_e2(je,jk)   *           &
      p_patch%edges%area_edge(je,jb))
  ENDDO
ENDDO
!$ACC END PARALLEL
```

- Promote data reuse between the loops

OpenACC tips and tricks

Advanced options

- Many possibilities to improve your performance even more
- Using TILE instead of collapse to group the iterations in a different way and improve caching
- Using STATIC:n for the GANG to make it process multiple loop iterations and increase ILP and caching
- CACHE directive may be useful, especially in conjunction with gang-private arrays and -gpu=safecache flag to make sure data ends up in fast shared memory
- -gpu=maxregcount:n would limit the number of registers of the GPU kernels and would improve occupancy
- CUDA graphs could be applied for some parts of the code to reduce latency between the kernels
- Atomic operations are quite efficient, but floating-point atomics break reproducibility (you may or may not really need it though)

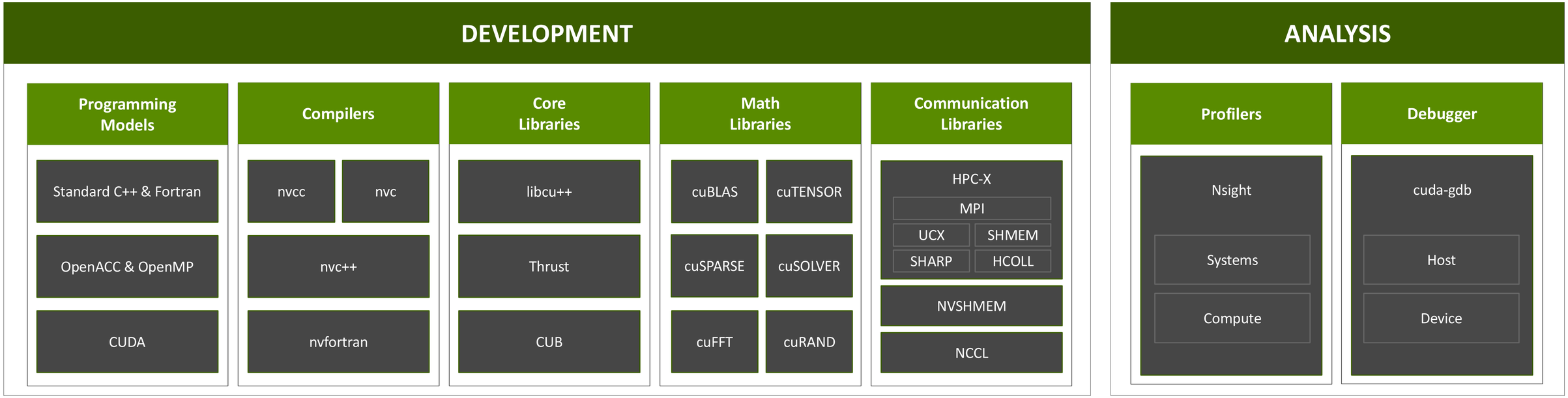
Compiling the GPU code

Useful compiler flags

- Try different optimization levels like -O2, -O3, -O4. Some kernels may be faster or slower with higher optimization
- -Mstack_arrays would place all the automatic arrays on stack. You have to increase stack size limit before running, e.g.: ulimit –s unlimited
- -nvmalloc would use jemalloc library. Very helpful on Arm CPUs, might be helpful on x86 for the codes doing many allocation and deallocations

NVIDIA HPC SDK

Available at developer.nvidia.com/hpc-sdk, on NGC, via Spack, preinstalled at CSCS



Develop for the NVIDIA Platform: GPU, CPU and Interconnect

Libraries | Accelerated C++ and Fortran | Directives | CUDA

x86_64 | Arm

6 Releases Per Year | Freely Available

How to debug and profile the GPUs?

Debugging the GPU code

General considerations

- You can use **print** in the device code. This is all you need 😈
- Honestly, print is really valuable when used properly:
  - Each thread can do a print. To avoid overwhelming output, only print the data from a few threads (use if conditions)
  - Keep in mind that the order of printed lines is not specified, especially when the line come from different thread blocks. Always add an ID that can correlate work with its output
  - print is slow and it may affect the execution flow. This may mask or exacerbate synchronization bugs
- **compute-sanitizer** is ideal to track illegal and uninitialized accesses and find race conditions. Similar to valgrind for the CPU. Compiling the code without OpenACC and with –Mbounds flag can catch the same errors, but not always
- **cuda-gdb** is helpful, especially when you know that a particular kernel is faulty and when you can run your application with a single rank (i.e., no MPI parallelization)

Debugging the GPU code

Environment variables

- NVCOMPILER_ACC_SYNCHRONOUS=1: make all the GPU work synchronous with the host despite ASYNC clauses. Useful when synchronization bug and race conditions are suspected
- NVCOMPILER_ACC_NOTIFY=N where N is binary combination of flags (use 31 for all the info):
  - 1: prints GPU kernel launches (parallel or kernels clauses)
  - 2: prints data transfers
  - 4: prints exit/entry of a data region
  - 8: prints info about wait clauses and synchronization
  - 16: prints allocations and deallocations
- NVCOMPILER_ACC_DEBUG=1: similar to NOTIFY=31, but more information. Useful to track memory issues, like when CPU memory is freed without freeing GPU memory
- NV_TERM=trace: if compiled with –traceback -gopt flags, will print backtrace on segmentation violations

Debugging the GPU code

PCAST

- Nvidia compiler can compare the results, computed with OpenACC on the GPU with the CPU while running
- This is called PCAST and can be used in a manual or automatic mode
- Automatic: add –gpu=autocompare flag and run
- For more details, see https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html#pcast
- Automatic approach works well for small-ish and partially ported codes, may not be that great for large fully ported apps like the full ICON
- To bring CPU and GPU computations closer together, you can turn off FMA in both: -Mnofma and add –gpu=math_uniform flag to the GPU compilation

Profiling the GPUs

- There are two tools available: **Nsight Systems** and **Nsight Compute**
- Nsight Systems shows you an overview of the program as whole. It has a timeline with kernels running, API calls, CPU samples, etc. etc. Always start with Nsight Systems
- Nsight Compute is designed to help you optimize individual kernels. Can be a bit overwhelming, so go through On-Demand tutorials and demos
  - What I find very important is to check kernel memory traffic and FLOPs against expectations, probably roofline analysis
  - Then, warp stall reasons and occupancy calculator
  - And moreover, check the most sampled instructions in the Source tab
- Both tools are available in HPC SDK are require no special compilation. Reports can be viewed as text on the target machine or saved to a file. The file can be nicely visualized on your own laptop or desktop
- Nsight Systems example command: nsys profile -t openacc -o my_report ./app
- Nsight Compute example command: ncu --set full --import-source -o my_report ./app
  - Plenty of kernel filtering options, check the docs!

Profiling the GPUs

NVTX

- You can manually annotate some parts of the code with simple markers which will show up in the Nsight Systems report (will see it later)
- This is called NVTX ranges: https://docs.nvidia.com/nvtx/index.html
- Easy basic usage in Fortran:

```
! main.f90

use nvtx

...

call nvtxStartRange(name)
call do_stuff(...)
call nvtxEndRange()
```

```
# Makefile

# Compilation doesn't change
%.o: %.f90
    $(FC) -o $@ -c $(FCFLAGS) $<

# Linking requires an extra flag
app: $(object_files)
    $(LD) -o $@ -c $(LDFLAGS) -lnvhpcwrapnvtx $^
```

The Tools

GPU tools are good, use them!

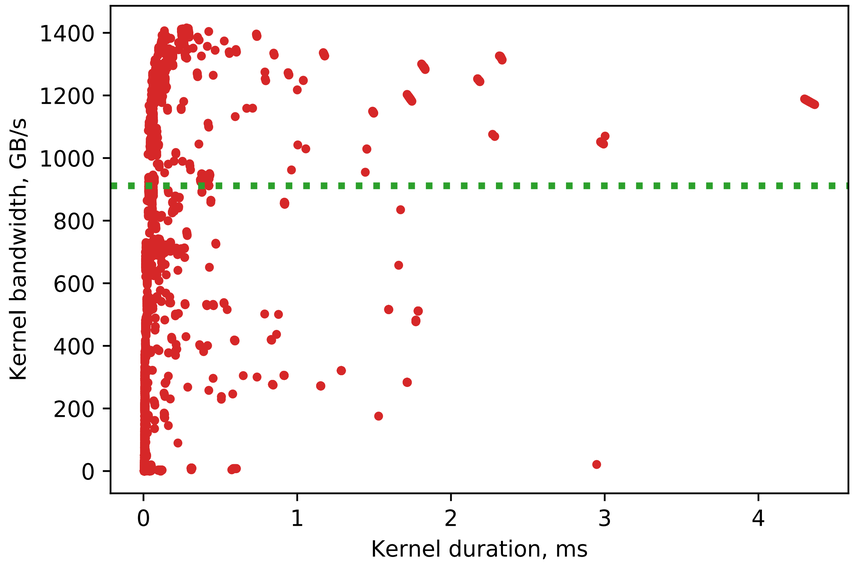
- My main point: the tools are so easy to use, give them a try! Do not be afraid of complexity, the workflow is actually not complex at all
- Some of the tool reports could be hard to interpret, but it’s still much better that no reports at all

Nsight Systems example

- Let’s have a quick look at some profiles

ICON automated performance analysis

- Nsight Compute can be used to extract bandwidth achieved by each kernel
- Required to perform detailed analysis of kernels bottlenecks
- Can identify kernels that yield low memory throughput or low overall GPU load
- Analysis can be easily automated with simple Python scripts



This kernel needs to improve

Kernel	Duration, ms	Bandwidth, GB/s
adding_1017_gpu	4.319616	1069.032674
lw_solver_noscat_136_gpu	3.002656	1028.267723
o3_pl2ml_219_gpu	2.947744	21.056265
inc_2stream_by_2stream_bybnd_576_gpu	2.325728	1041.890441
sw_two_stream_838_gpu	2.178112	1247.553819
lw_transport_noscat_576_gpu	1.826496	1328.899786
gas_optical_depths_major_339_gpu	1.788000	501.779672
inc_1scalar_by_1scalar_bybnd_456_gpu	1.771296	436.706152
compute_planck_source_578_gpu	1.716992	274.720473
sw_source_2str_942_gpu	1.714656	1253.359748

Extra resources

- OpenACC standard, actually very accessible: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.3-final.pdf
- Nvidia on-demand: https://www.nvidia.com/en-us/on-demand/
- GTC recordings: https://www.nvidia.com/gtc/session-catalog/?regcode=no-ncid&ncid=no-ncid&search=#/
  - https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31816/
  - https://resources.nvidia.com/en-us-summer-of-learning-for-students/gtcspring22-s41496?ncid=no-ncid
  - https://www.nvidia.com/en-us/on-demand/session/gtc24-s62191/?playlistId=playList-d59c3dc3-9e5a-404d-8725-4b567f4dfe77
  - https://www.nvidia.com/en-us/on-demand/session/gtcspring23-s51120/
  - https://www.nvidia.com/en-us/on-demand/session/gtcspring23-s51772/
  - https://www.nvidia.com/en-us/on-demand/session/gtc24-dlit61667/
- Be careful when looking at old stuff like GPU Gems or 10 yr old stackoverflow answers. A lot has changed in CUDA platform and a lot more is possible now
- The best way to learn is to try, so get your hands dirty!

Hands on session

Hands on session

- The code is supposed to solve a system with conjugate gradient method, but it has a few bugs and a few optimization opportunities.
- You'll need to debug it, then profile with Nsight System and try to speed it up
- Try to use some tools/environment variables rather than just staring at the code
- Get Nsight Systems GUI onto your laptop: https://developer.nvidia.com/nsight-systems/get-started