KTH ROYAL INSTITUTE
OF TECHNOLOGY

Porting Nek5000 on GPUs Using OpenACC and CUDA

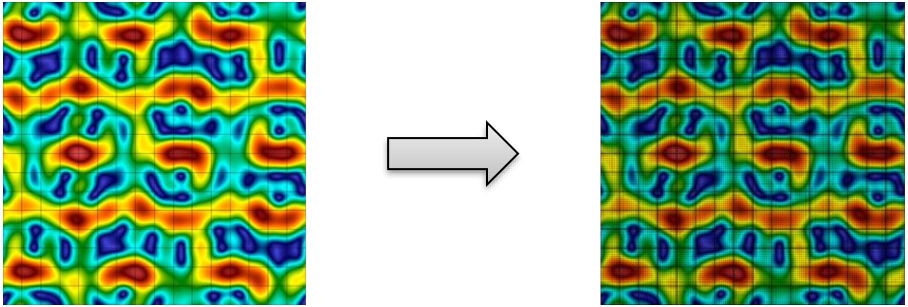
Niclas Jansson
PDC Center for High Performance Computing

Acknowledgments

- Jing Gong, KTH
- Adam Peplinski, KTH
- Jonathan Vincent, KTH
- Martin Karp, KTH
- Andreas Jocksch, CSCS (EuroHack’19 mentor)
- Alan Gray, Nvidia (EuroHack’19 mentor)
- Nek5000 developers (ANL and UIUC)

Introduction

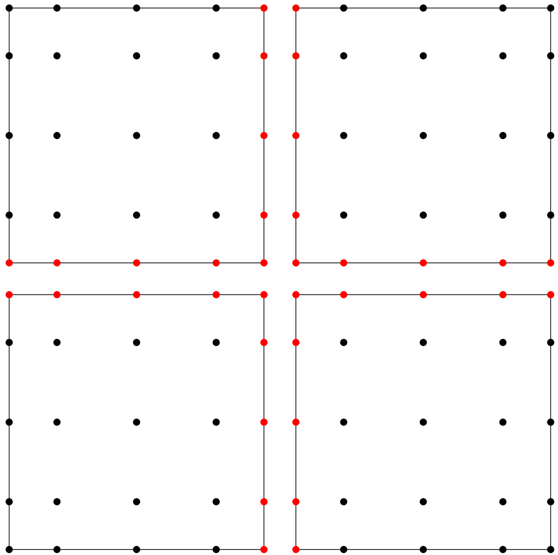
- Open-source spectral element code (ANL), with its roots in the 80s (MIT) and a commercial version NEKTON (part of Fluent).
- Solves the incompressible Navier-Stokes with a number of additional physics (heat transfer, magneto-hydrodynamics, low Mach number, electrostatics)



- General hexahedral spectral element, with special focus on single-core efficiency (matrix-free, tensor products)
- Mainly Fortran 77, with communication and I/O parts in C

Introduction

- High order basis functions
  - Dense local stiffness matrices A^E
  - Too expensive to assemble global (and local matrices)
  - Continuity across elements
- Matrix-free operator evaluation
  - A^E can be computed in parallel
  - $A = Q^T A_L Q$
  - Boolean matrix Q
  - QQ^T gather-scatter kernel

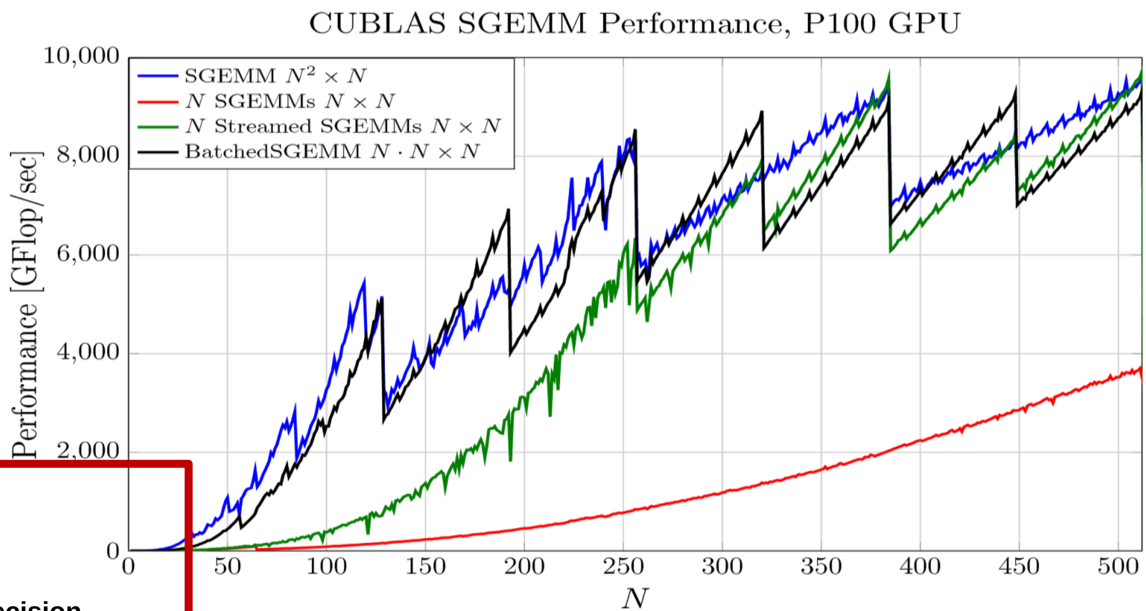


Nek5000 on GPUs

- Small tensor products (`mxm`) of various sizes e.g. $N^3, N^2 \cdot N, N \cdot N^2$ with $N \approx 8 - 16$,or more, related to polynomial order
- Code written in a rather GPU unfriendly way
  - Many small function calls

```
do e=1,nel
  call mxm(…)
  do j=1, n
    call mxm(…)
  enddo
  call mxm(…)
enddo
```

Nek5000 on GPUs



Nek5000

N = 4-16
Double precision

Nekbone on GPUs

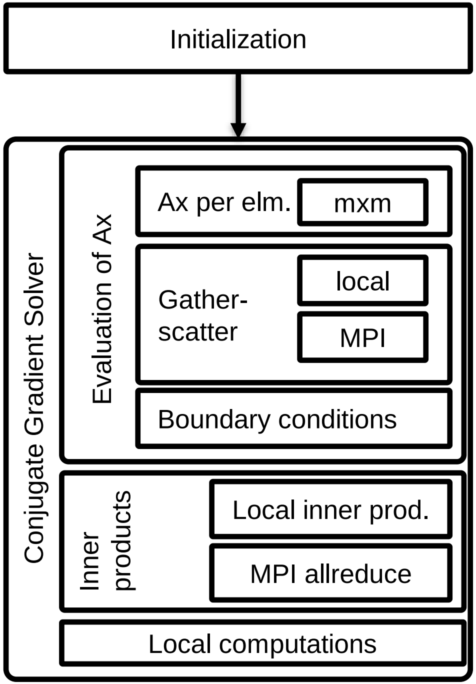
- Focus on Nekbone, Nek5000's mini-app
  - Solves a linear system using PCG
    - > *Trivial preconditioner M*

Algorithm 2 Matrix-free conjugate gradient solver.

for $k = 1, 2, \ldots$ **do**
  $z_L \leftarrow M^{-1} r_L$
  $\rho_2 \leftarrow \gamma_1$
  $\rho_1 \leftarrow r_L^T C_L z_L$
  $\beta \leftarrow \frac{\rho_1}{\rho_2}$
  if $k = 1$ **then**
    $\beta \leftarrow 0$
  end if
  $p_L \leftarrow \beta p_L + z_L$
  $w_L \leftarrow M_L Q Q^T A_L p_L$ — Ax and gs routines
  $\gamma \leftarrow w_L^T C_L p_L$
  $\alpha = \frac{\rho_1}{\gamma}$
  $x_L \leftarrow x_L + \alpha p_L$
  $r_L \leftarrow r_L - \alpha w_L$
  $\epsilon \leftarrow \sqrt{r_L^T C_L r_L}$
end for

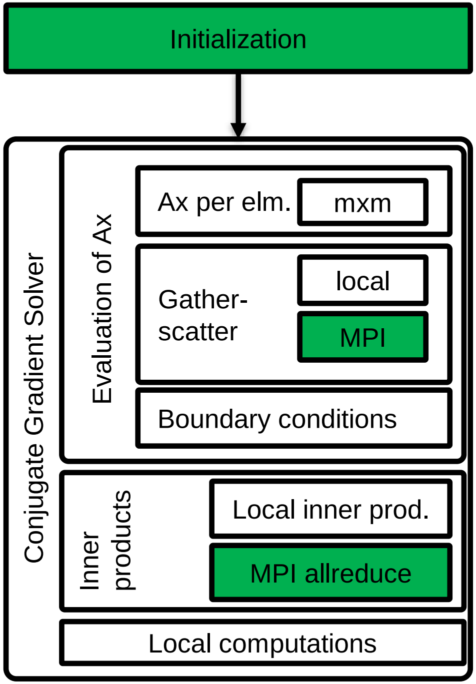
Nekbone on GPUs

- Focus on Nekbone, Nek5000's mini-app
  - Solves a linear system using PCG
    - > *Trivial preconditioner M*
- OpenACC implementation
  - Use directives to move data between host and device
    - > *Keep entire problem on device except for communication steps*



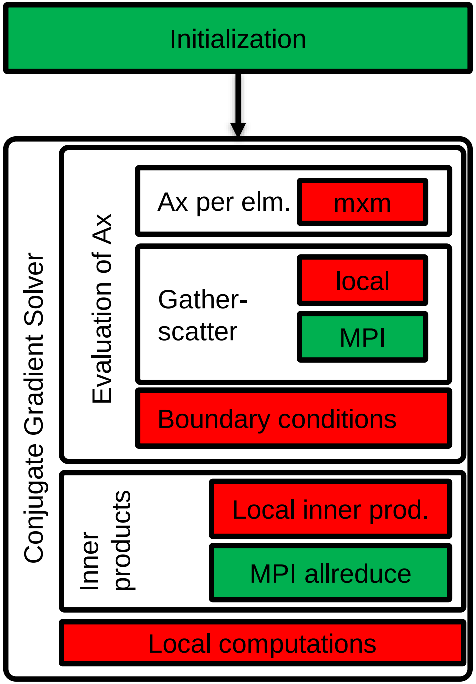
Nekbone on GPUs

- Focus on Nekbone, Nek5000's mini-app
  - Solves a linear system using PCG
    - > *Trivial preconditioner M*
- OpenACC implementation
  - Use directives to move data between host and device
    - > *Keep entire problem on device except for communication steps*



Nekbone on GPUs

- Focus on Nekbone, Nek5000's mini-app
  - Solves a linear system using PCG
    - > *Trivial preconditioner M*
- OpenACC implementation
  - Use directives to move data between host and device
    - > *Keep entire problem on device except for communication steps*
  - Add directives to move computation from host to device in remaining parts
    - > *Matrix-matrix products*
    - > *Inner products*
    - > *Local gather-scatter operations*
    - > *Various vector ops. e.g. scale, add.*



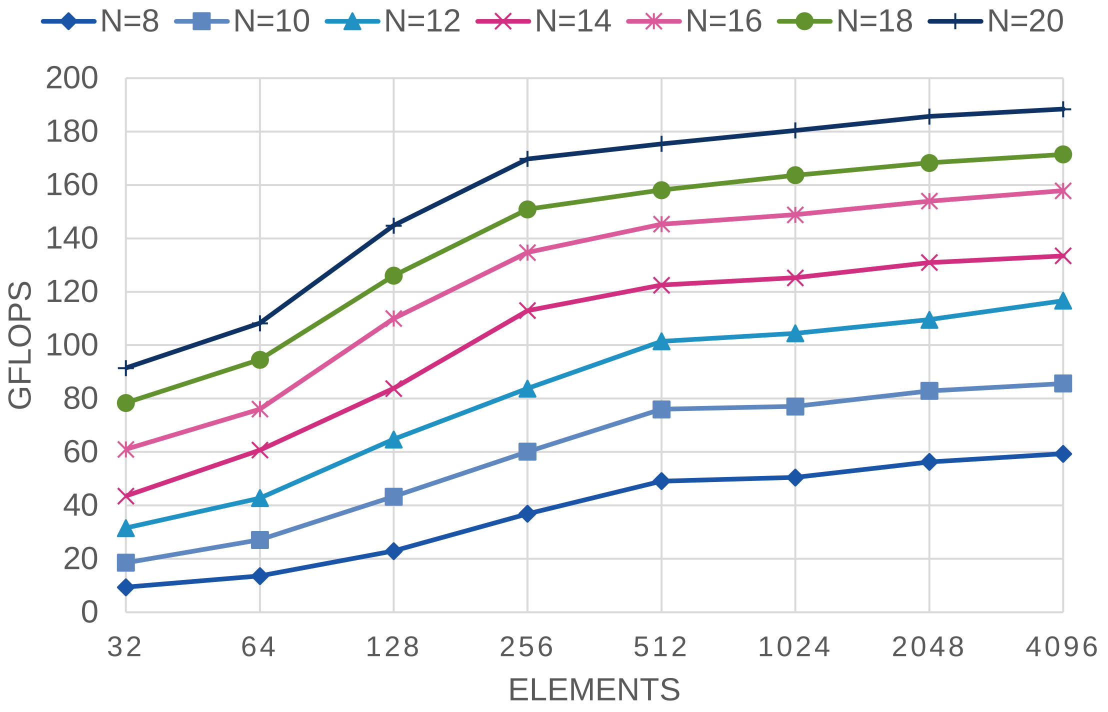
Nekbone on GPUs

- Refactor code to be more GPU friendly
  - Merge small function calls into larger loop nests

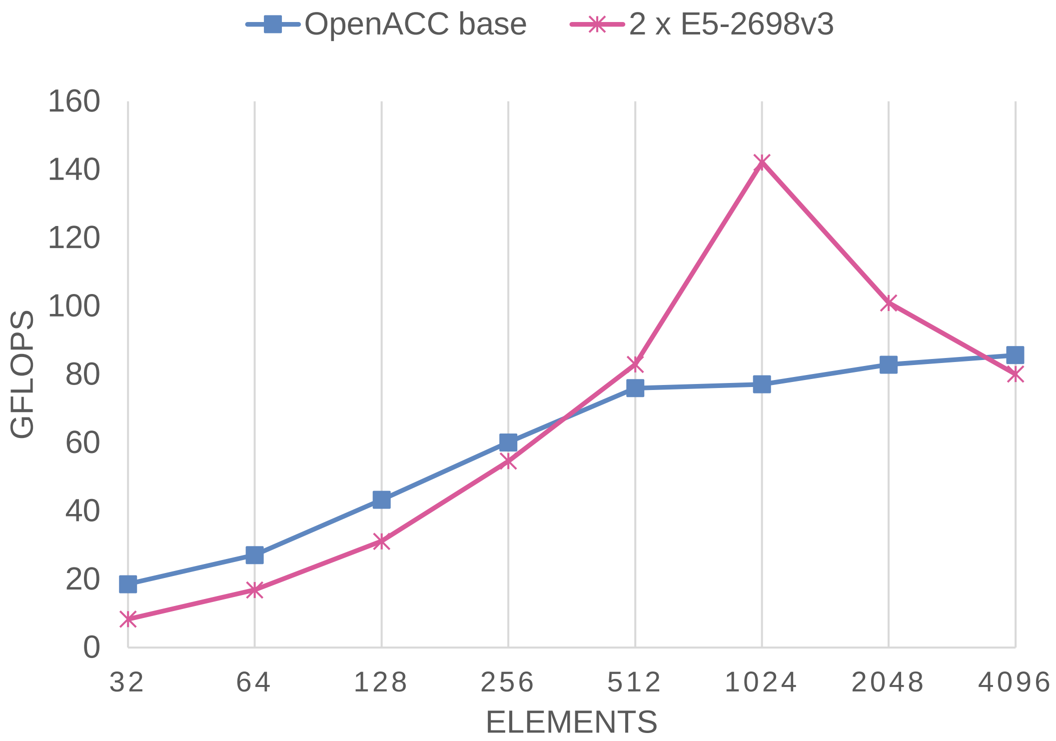
```
do e=1,nel
  call mxm(…)
  do j=1, n
    call mxm(…)
  enddo
  call mxm(…)
enddo
```

```
!$ACC  DATA PRESENT(u,w)
!$ACC& PRESENT(D,g)
!$ACC  PARALLEL LOOP COLLAPSE(4)
!$ACC& GANG WORKER VECTOR PRIVATE(temp)
!$ACC& VECTOR_LENGTH(128)
do e=1, nel
  do k=1,n
  do j=1,n
  do i=1,n
    temp = 0
    !$ACC SEQ
    do l=1,n
      temp = temp +D(i,l)*u(l,j,k,e)
    enddo
    w(i,j,k,e) = g(i,j,k,e)*temp
  enddo
  enddo
  enddo
enddo
```

Nekbone on a single GPU (P100) OpenACC



Nekbone on a single P100 vs Haswell node



EuroHack'19

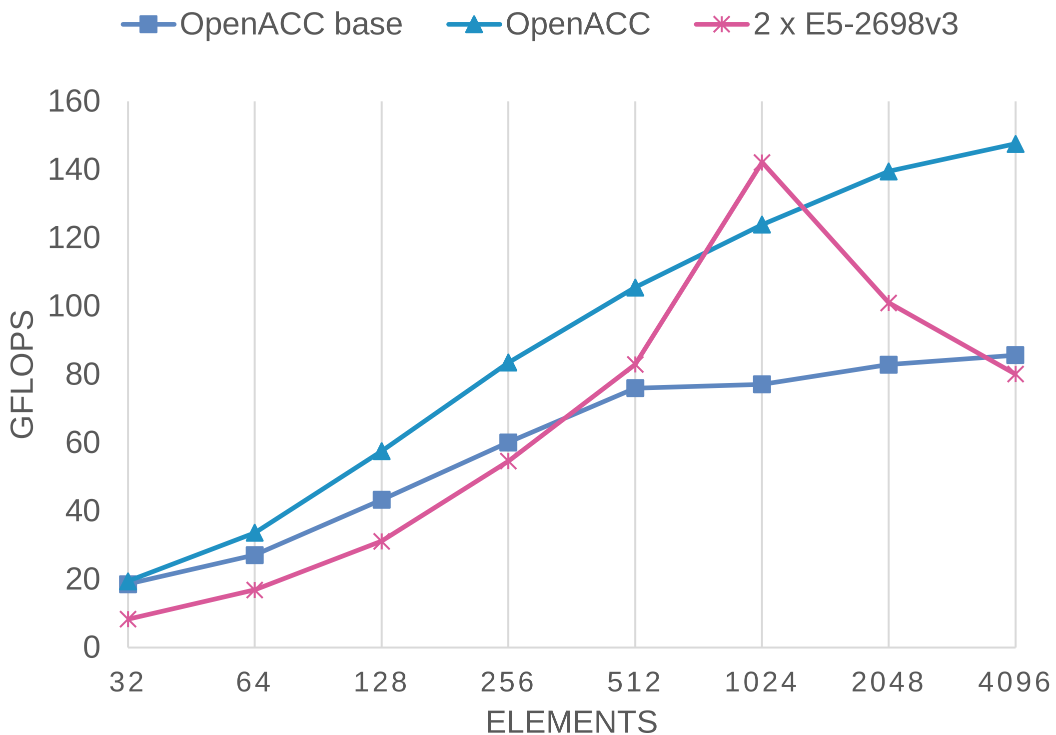
- Focus on tensor products performance using Nekbone
  - Tested different approaches with CUBLAS routines dgemm:
    - > *Single matrix, banded, strided*
    - > *Better performance for* $N > 32$ *, but no improvement for lower polynomial orders due to minimal kernel sizes*
  - Tune OpenACC directives

```
!$ACC  DATA PRESENT(u,w)
!$ACC& PRESENT(D,g)
!$ACC  PARALLEL LOOP COLLAPSE(4)
!$ACC& GANG WORKER VECTOR PRIVATE(temp)
!$ACC& VECTOR_LENGTH(128)
do e=1, nel
  do k=1,n
  do j=1,n
  do i=1,n
    temp = 0
    !$ACC SEQ
    do l=1,n
      temp = temp +D(i,l)*u(l,j,k,e)
    enddo
    w(i,j,k,e) = g(i,j,k,e)*temp
  enddo
  enddo
  enddo
enddo
```

→

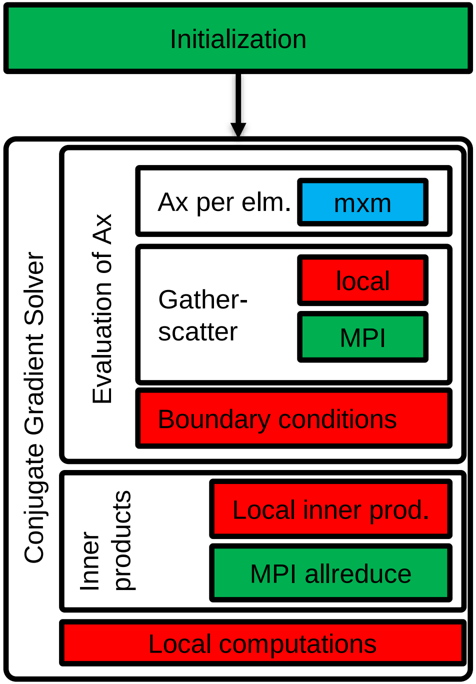
```
!$ACC  DATA PRESENT(u,w)
!$ACC& PRESENT(D,g)
!$ACC  PARALLEL PRIVATE(temp)
!$ACC& LOOP GANG
do e=1, nel
!$ACC SEQ
  do k=1,n
!$ACC LOOP COLLAPSE(2)
  do j=1,n
  do i=1,n
    temp = 0
    do l=1,n
      temp = temp +D(i,l)*u(l,j,k,e)
    enddo
    w(i,j,k,e) = g(i,j,k,e)*temp
  enddo
  enddo
  enddo
enddo
```

Nekbone on a single P100 (EuroHack'19)

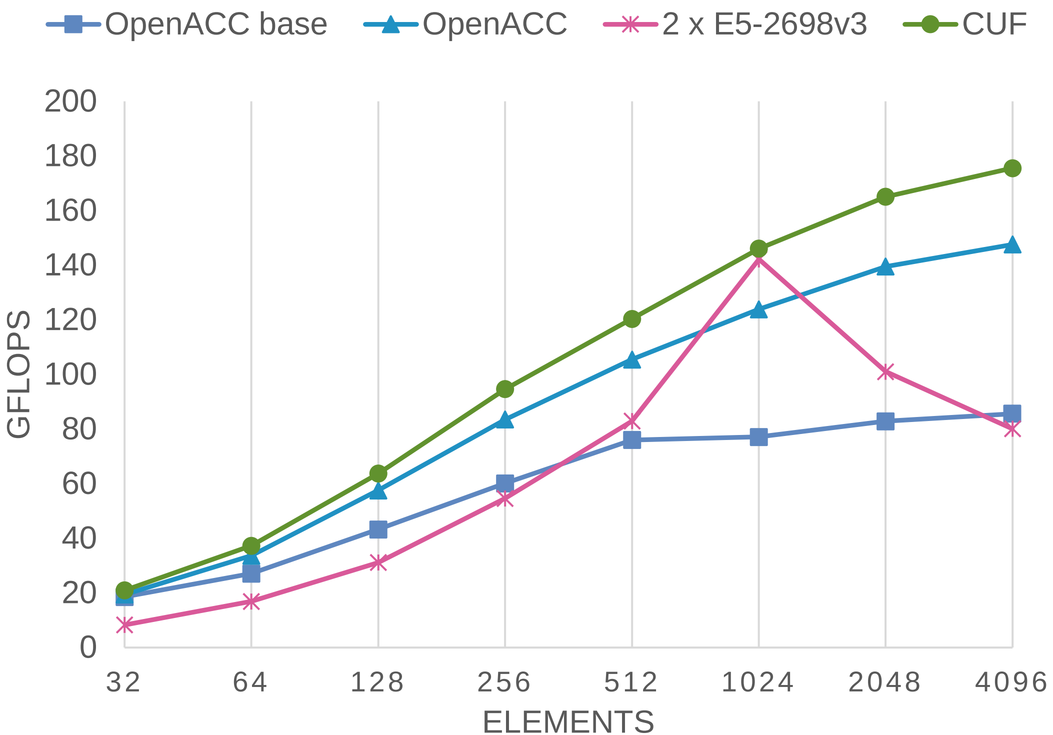


EuroHack’19

- How about using CUDA?
  - Generic CUDA Fortran kernels for Ax

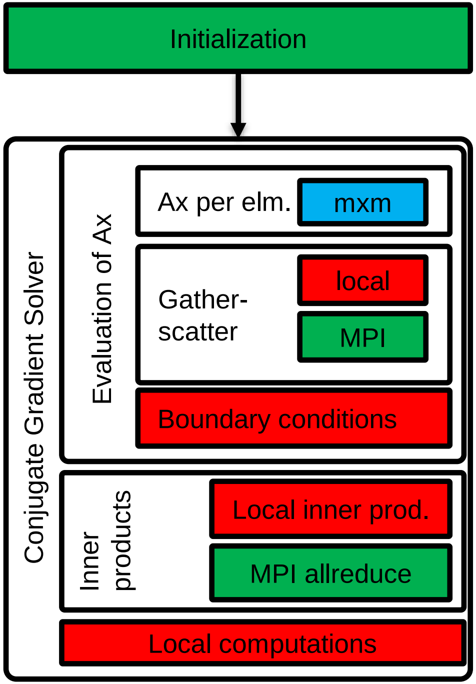


Nekbone on a single P100 (EuroHack'19)

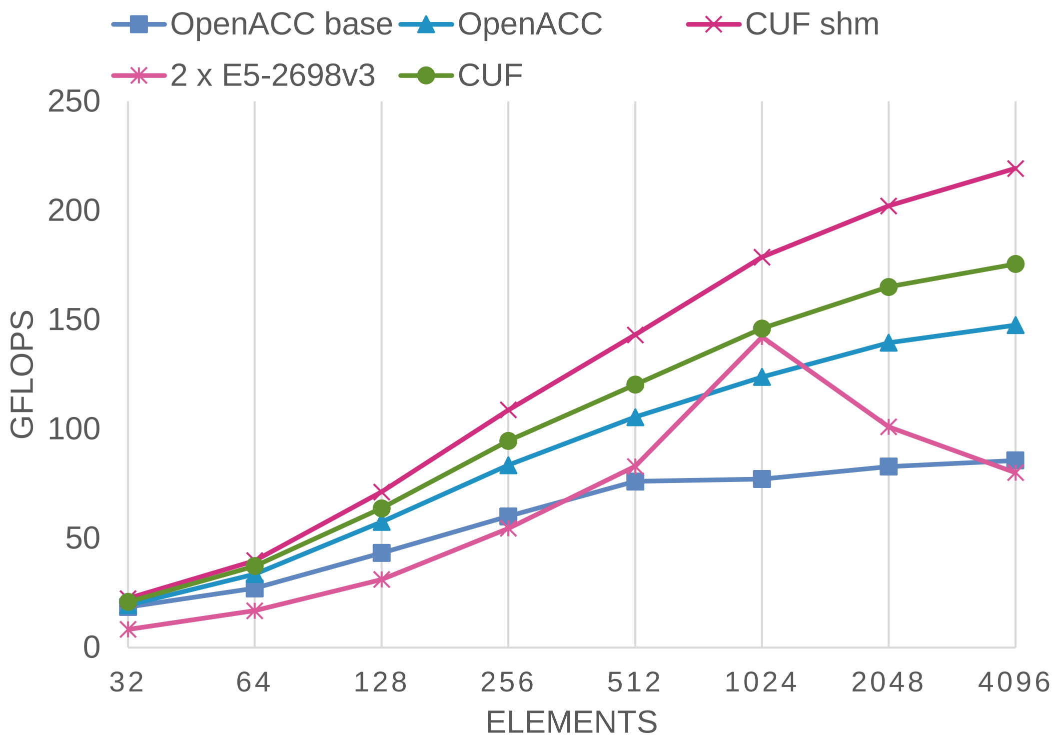


EuroHack’19

- How about using CUDA?
  - Generic CUDA Fortran kernels for Ax
- Exploit the GPU’s shared memory
  - Implement specific kernels depending on polynomial order
  - 3D block thread structure, $N = 10$ largest size that could fit on a P100

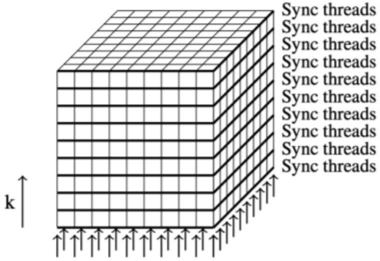


Nekbone on a single P100 (EuroHack'19)

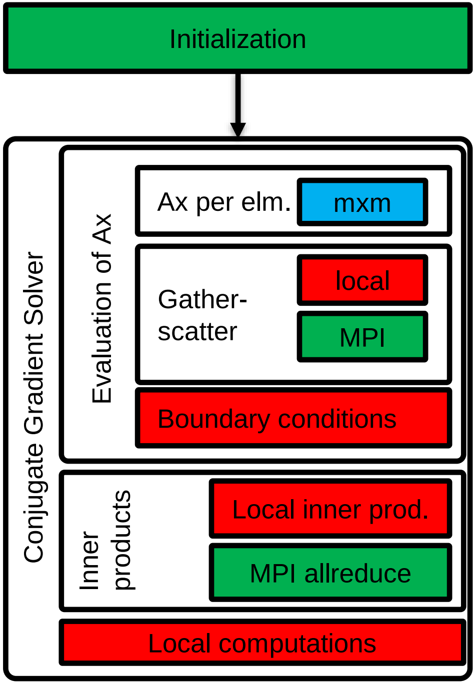


Post EuroHack'19 work

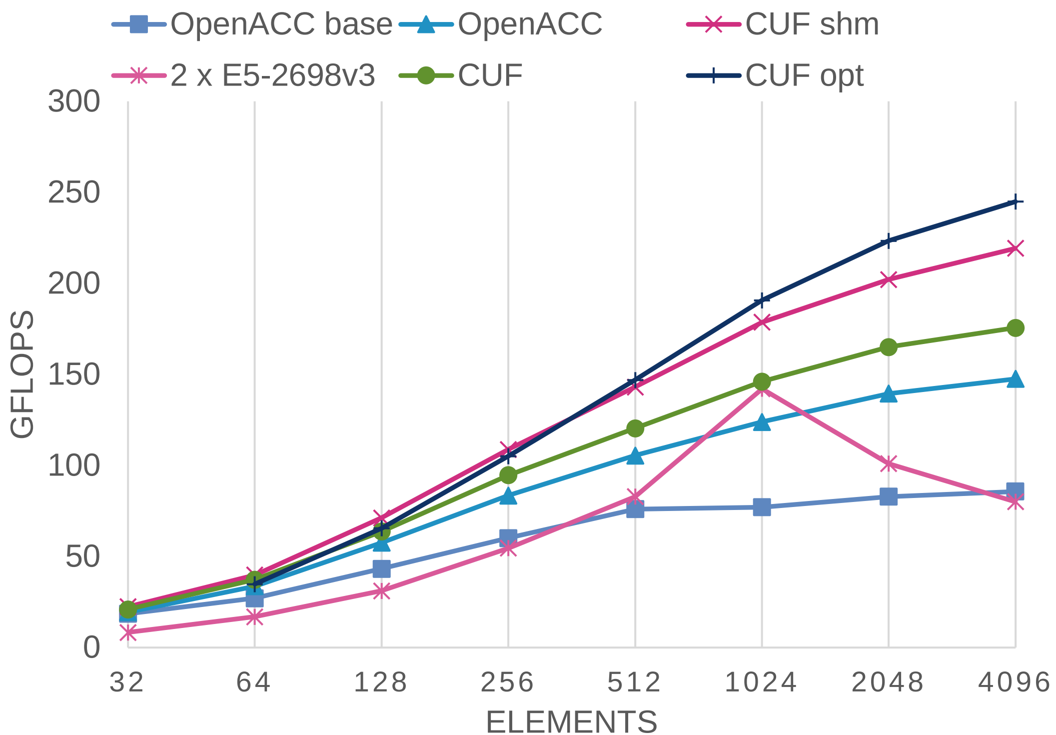
- How about using CUDA?
  - Generic CUDA Fortran kernels for Ax
- Exploit the GPU's shared memory
  - Implement specific kernels depending on polynomial order
  - 3D block thread structure, $N = 10$ largest size that could fit on a P100
- Change thread structure
  - Use 2D slices instead of blocks



  - Less shared memory usage, can handle larger N



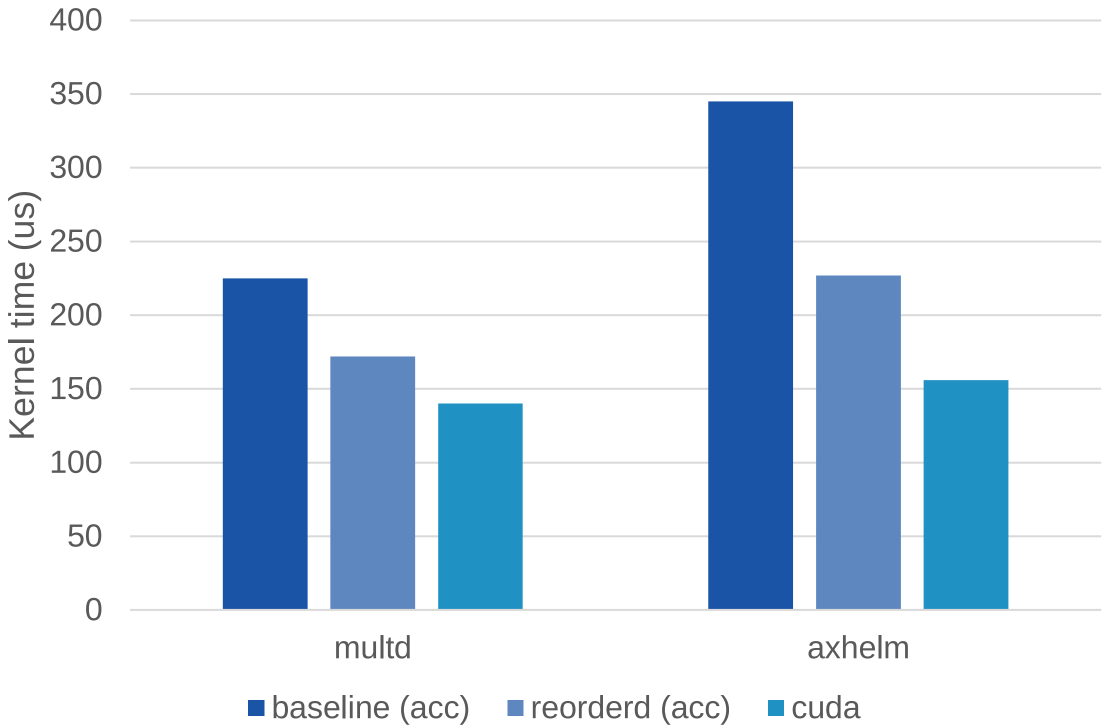
Nekbone on a single P100 (EuroHack'19)



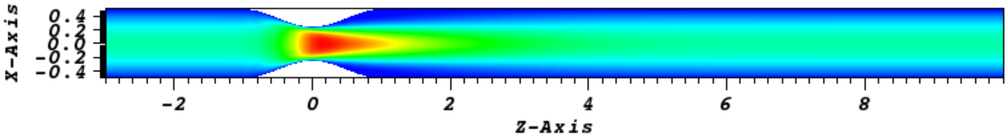
From Nekbone to Nek5000

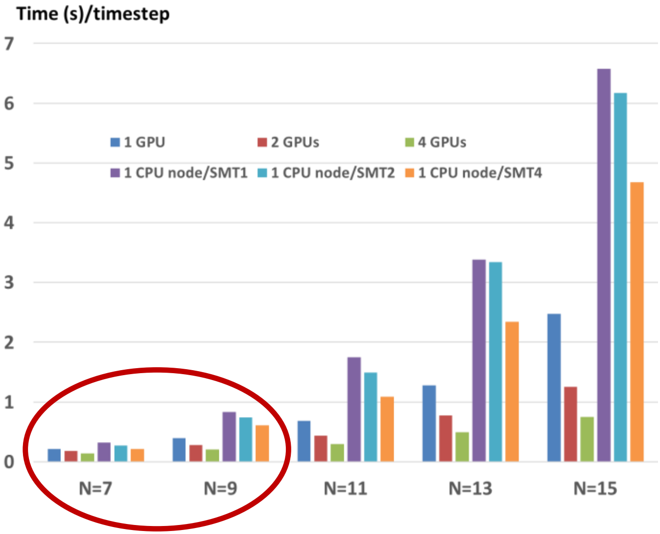
- Follow the same approach as for Nekbone
  - Incrementally adding OpenACC directives throughout the code
  - Tune existing OpenACC implementation, reorder directives
    - > *proper use of* `loop seq` *and* `loop vector collapse`
- Gather-Scatter kernels are the same
  - Use OpenACC version from Nekbone
- Nek5000's Helmholtz solver very similar to Nekbone
  - Prototyped kernels more or less a drop in replacement
  - Six matrix-matrix kernels
- Pressure solver is a different story…
  - Most "issues" related to coarse grid solver

Nek5000 – Optimised kernels



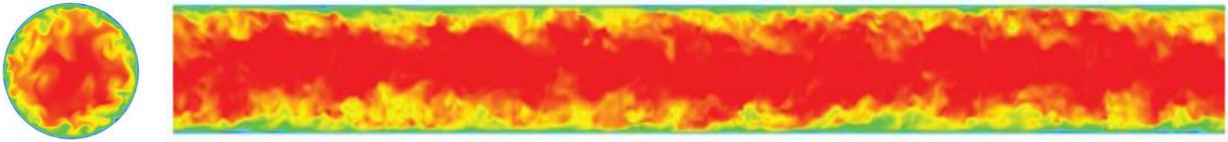
Stenosis Simulation on CTE-Power System



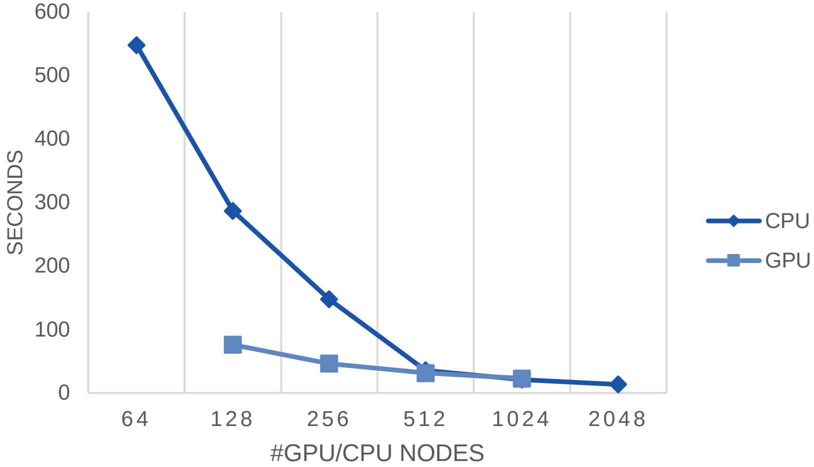


Comparison of the computing time per time-step for the stenosis case on a single CTE-Power node, between 40 CPU cores with SMT 1/2/4 and 1/2/4 V100 GPUs, for different polynomial orders N and a fixed number of elements E=2000.

Pipe Simulations on Piz Daint



The turbulent flow in a straight pipe. The flow was run at friction Reynolds numbers Re_tau = 360. Run for 50 steps and only calculate the execution time for the last 20 steps



Summary and future work

- OpenACC was a perfect tool for incrementally porting the entire code base from CPU to GPU
- Still necessary to spend some time tuning kernels (directives)
  - Important to have extracted representative kernels or mini-apps (Nekbone)
    - > *proper use of* ***loop seq*** *and* ***loop vector collapse***
    - > ***40%*** *improvement* ***for mxm call***
  - Use tuned CUDA for key kernels and OpenACC for the bulk of the application
- Scalability is an issue
  - Good performance requires either high polynomial orders or large problems
    - > *Higher order puts unrealistic constraints on CFL condition*
    - > *Larger problem sizes per GPU causes communication/memory bottlenecks*
  - Necessary to put more effort on the coarse grid solver (EuroHack'20?)