Durham University
Department of Computer Science

Flavours of GPU kernels in ExaHyPE

SYCL's impact on algorithms, data structures and implementations

D.E. Charrier[*], T. Weinzierl[+], M. Wille[⊤], H. Zhang[⊥]

[*] Advanced Micro Devices (AMD), Germany
[+] Computer Science, Durham University
[⊤] Computer Science, Technische Universität München
[⊥] Institute for Computational Cosmology (ICC), Durham University

February 27, 2023

Outline

Durham University
Department of Computer Science

ExaHyPE

High-performance computing strategy

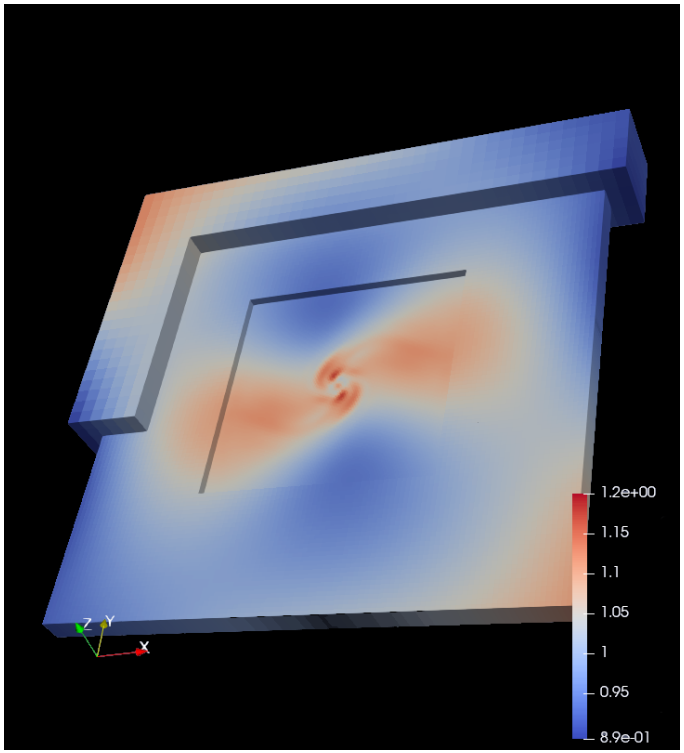
Finite Volumes

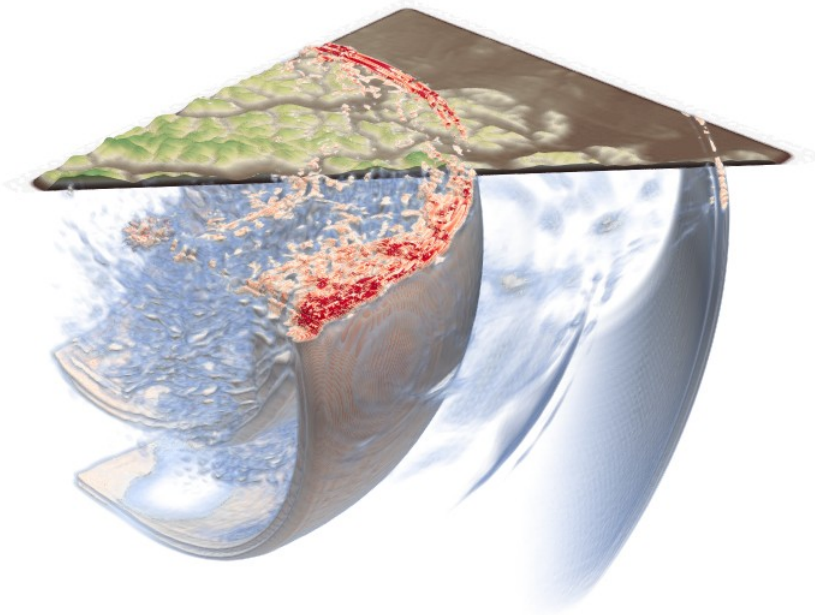
Migrating from OpenMP to SYCL

Conclusion

Two application challenges

Durham University
Department of Computer Science





Work by groups of B. Li et al (Institute for Computational Cosmology) and M. Bader et al (TUM)

Chris Johnson: “Before every great invention was the discovery of a new tool”[+].

[+] From an interview in the video “The Golden Age of Computing”.

Hyperbolic equation systems

Durham University
Department of Computer Science

ExaHyPE's vision: The engine ExaHyPE allows groups with decent computational background to write an exascale solver for

$$\frac{\partial}{\partial t} Q + \nabla \cdot \mathbf{F}(Q) + \sum_i \mathcal{B}_i \frac{\partial Q}{\partial x_i} = \mathbf{S}(Q).$$

User:

- Pick numerical scheme (FV, RKFD, RKDG, ADER-DG)
- Supplement BCs, initial conditions
- Implement PDE terms
- Indicate where to refine
- ...
- Do **not** care about all the code environment, meshing, numerics

Engine:

- Constrain to a small, fixed set of ingredients
- Generate actual simulation code (glue code, numerical kernels, ...)
- Determine *where*, *when*, *in which order* and *how* to call compute kernels

Disclaimer: Concepts inherited from underlying AMR framework Peano, prototyped in ExaHyPE, all rewritten from scratch in ExaHyPE 2 (cmp. ACM TOMS).

ExaHyPE 2's Python scripts

Durham University
Department of Computer Science

```
import peano4
import exahype2

my_solver    = exahype2.solvers.rkdg.rusanov.GlobalAdaptiveTimeStepWithEnclaveTasking(
 name               = "CCZ4DG8RK4",
 rk_order           = 4,
 polynomials        = exahype2.solvers.GaussLegendreBasis(8),
 unknowns           = 59,
 [...]

my_solver.set_implementation(
 boundary_conditions  = exahype2.solvers.rkdg.PDETerms.Empty_Implementation,
 flux                 = exahype2.solvers.rkdg.PDETerms.None,
 ncp                  = """
  // my_code_snippets
 """
 [...]

project = exahype2.Project( ... )
project.add_solver( my_solver )
project.set_global_simulation_parameters(
 dimensions = 3,
 offset = [0.0,0.0,0.0],
 size   = [1.0,1.0,1.0],
 [...]

project.set_load_balancing( ... )
project.set_Peano4_installation("../../..", peano4.output.CompileMode.Release)
peano4_project = project.generate_Peano4_project()
peano4_project.output.makefile.parse_configure_script_outcome( "../../.." )
peano4_project.generate()
```

Callbacks in ExaHyPE 2

Durham University
Department of Computer Science

```
double examples::exahype2::euler::Euler::maxEigenvalue( ... ) {
  constexpr double gamma = 1.4;
  const double irho = 1./Q[0];
  #if Dimensions==3
  const double p = (gamma-1) * (Q[4] - 0.5*irho*(Q[1]*Q[1]+Q[2]*Q[2]+Q[3]*Q[3]));
  #else
  const double p = (gamma-1) * (Q[4] - 0.5*irho*(Q[1]*Q[1]+Q[2]*Q[2]));
  #endif
  const double u_n = Q[normal + 1] * irho;
  const double c   = std::sqrt(gamma * p * irho);
  return std::max( std::abs(u_n-c), std::abs(u_n+c) );
}
```

(Example above: Euler equations, i.e. nothing fancy but textbook by R. LeVeque)

- Python API yields plain Makefile/CMake
- Glue code all generated
  (meshing, load balancing, algorithm phases, plotting, ...)
- Empty stubs for physics/application domain-specific ingredients
  (Python may inject C code snippets directly or use SymPy

Outline

Durham University
Department of Computer Science

ExaHyPE

High-performance computing strategy

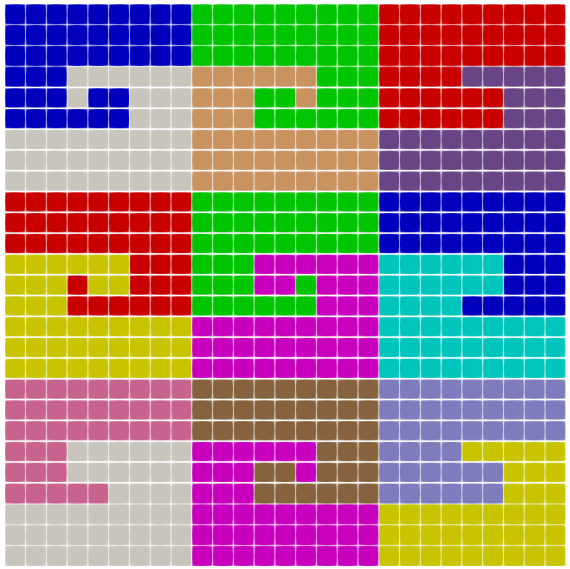
Finite Volumes

Migrating from OpenMP to SYCL

Conclusion

Classic domain decomposition: MPI+X

Durham University
Department of Computer Science



SFC-based domain partitioning of regular grid for 18 ranks/cores.

- Adaptive Cartesian mesh based upon spacetrees
- SFC-based non-overlapping domain decomposition
- Boundary data exchange hidden
- Rebalancing triggered by callbacks

Realisation:

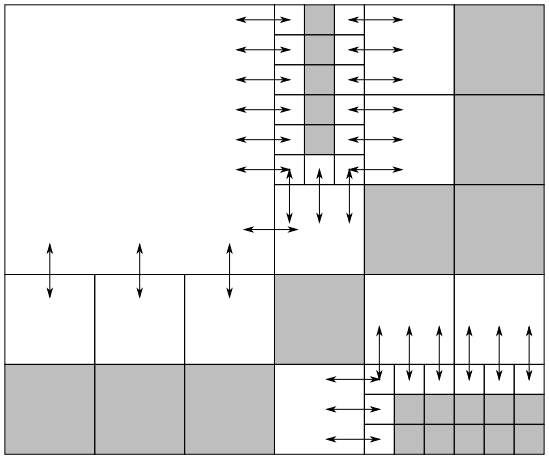
- No difference between MPI and shared memory parallelisation
- Both MPI+shared memory: SPMD+BSP

$\Rightarrow$ Inappropriate for many cores (boundary exchange overhead)

$\Rightarrow$ Struggles with imbalances (due to AMR or non-linear compute kernels, e.g.)

Enclave tasking

Durham University
Department of Computer Science



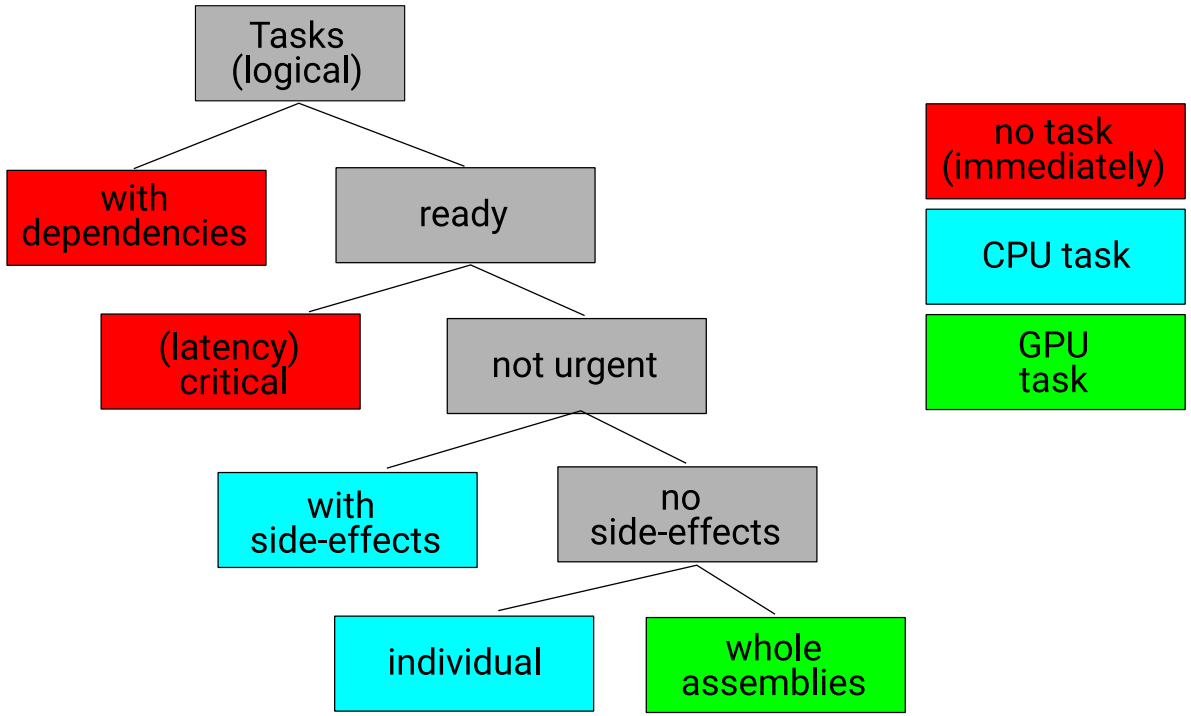
- Mark all cells along MPI boundary and resolution transitions
  (those are involved in MPI and might refine/coarsen)
  - reordering of these cells challenging
  - these cells are along critical path in task graph (latency sensitive)

  $\Rightarrow$ skeleton grid
- Remaining cells define real tasks
  - overlap with MPI and AMR
  - compensate for BSP imbalances

  $\Rightarrow$ enclave cells

Task classification and GPU offloading

Durham University
Department of Computer Science



- Hold GPU-fitting tasks back
- Deploy en bloc to device
  (classic offloading model)

Outline

Durham University
Department of Computer Science

ExaHyPE

High-performance computing strategy

Finite Volumes

Migrating from OpenMP to SYCL

Conclusion

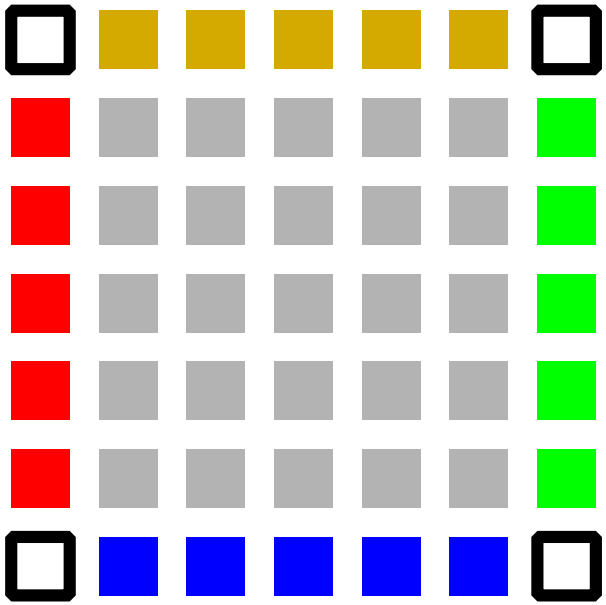
Disclaimer

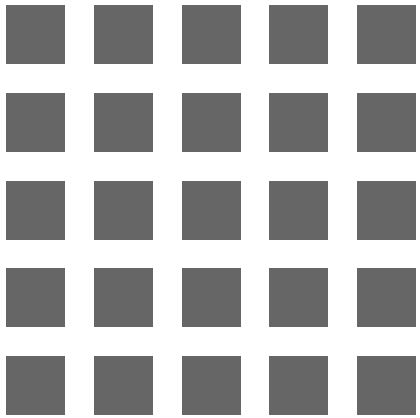
Durham University
Department of Computer Science

- Finite Volumes
  (same principles hold for all numerical schemes)
- Rusanov
  (everything else too complicated with CCZ4)
- Generic, application-agnostic kernels
  (see outlook)
- OpenMP
  (production back-end)
- Study only these enclave tasks
  (omit meshing/AMR, MPI, . . .)

Rusanov

Durham University
Department of Computer Science



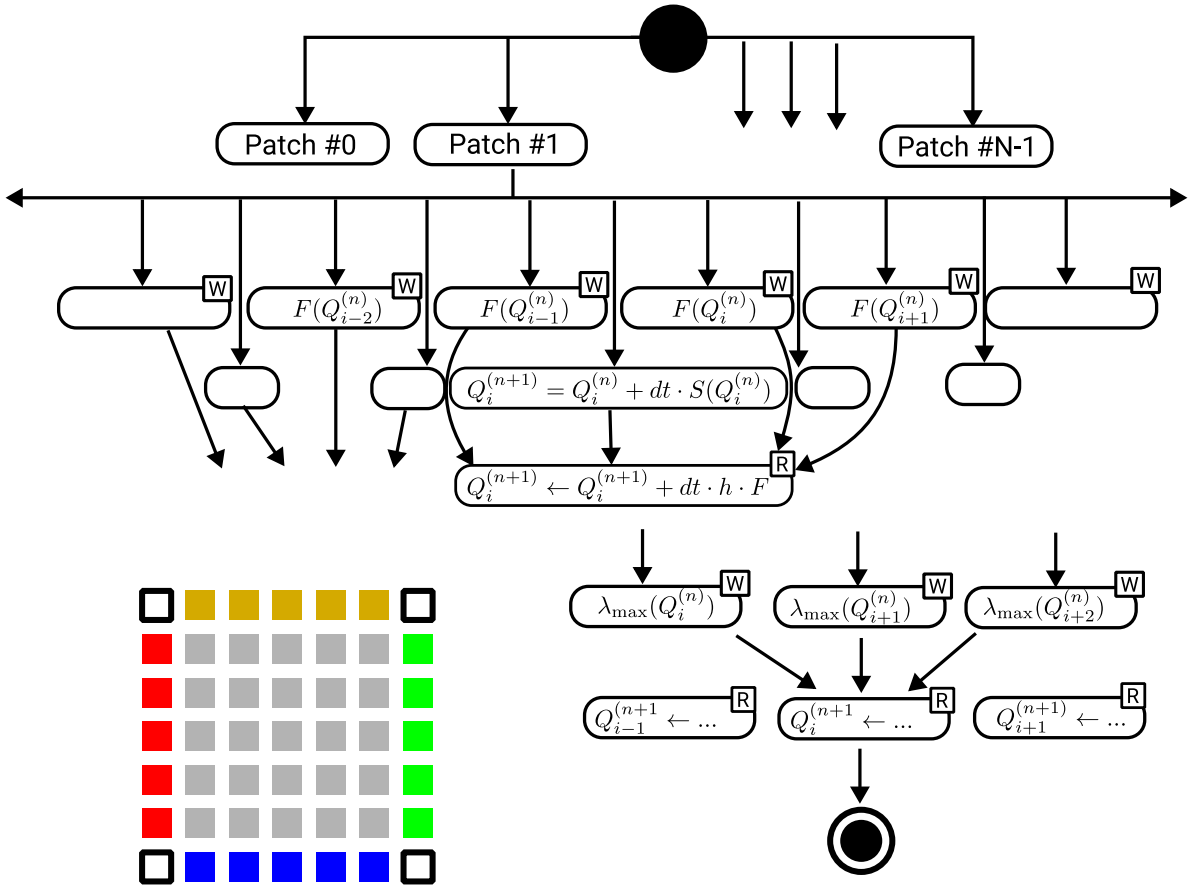


- Halo layer valid
- Array of structures (AoS)
- Numerics blueprint

$$Q_i^{(n+1)} = Q_i^{(n)} + dt \cdot S(Q_i^{(n)}) + dt \cdot h \cdot F_{i-\frac{1}{2}}(Q^{(n)}) + dt \cdot h \cdot F_{i+\frac{1}{2}}(Q^{(n)})$$

$$F_{i+\frac{1}{2}}(Q^{(n)}) = \frac{1}{2}F(Q_i^{(n)}) + \frac{1}{2}F(Q_{i+1}^{(n)}) - \frac{\max\left(\lambda_{\max}(Q_{i+1}^{(n)}), \lambda_{\max}(Q_i^{(n)})\right)}{2}\left(Q_{i+1}^{(n)} - Q_i^{(n)}\right)$$

Rusanov graph



- Outer loop over patches $n \in \{0, \ldots, N-1\}$
- Per PDE term: Compute data (write to temp array; concurrent)—update $Q^{(n+1)}$ (read; concurrent fan in/gather)
- Computations per PDE term concurrent (cmp. $\lambda_{\max}(Q^{(n)})$ vs. $F(Q^{(n)})$)
- Flexibility to order temporary data

Patch-wise Finite Volume Rusanov kernels

Durham University
Department of Computer Science

```
// commit to certain order
exahype2::enumerator::AoS0ALexicographicEnumerator fluxHostEnumerator(...);

// allocate temp buffers -> do on GPU
double* tempFluxX = new double[fluxHostEnumerator.size()];

#pragma omp target teams distribute device(targetDevice) \
 ...
for (int patchIndex=0; patchIndex<numberOfCells; patchIndex++) {
 if (Dimensions==2 and evaluateSource) {
  #pragma omp parallel for simd collapse(2)
  for (int y = 0; y < numberOfVolumesPerAxisInPatch; y++)
  for (int x = 0; x < numberOfVolumesPerAxisInPatch; x++) {
   internal::copySolutionAndAddSourceTerm_LoopBody<Solver>(
    ...

 if (Dimensions==2 and evaluateFlux) {
  #pragma omp parallel for simd collapse(2)
  for (int y = 0; y < numberOfVolumesPerAxisInPatch; y++)
  for (int x = -1; x < numberOfVolumesPerAxisInPatch+1; x++) {
   internal::evaluateFlux<Solver>(
    ...

  #pragma omp parallel for simd collapse(2)
  for (int y = 0; y < numberOfVolumesPerAxisInPatch; y++)
  for (int x = 0; x < numberOfVolumesPerAxisInPatch; x++) {
   internal::updateWithFlux<Solver>(
    ...
}
```

- Hide temp enumeration
- Distribute loop over patches among SMs/warps
- Collapse internal d-dimensional/$d+1$-dimensional `parallel for` loops

Batched Finite Volume Rusanov kernels

Durham University
Department of Computer Science

```
if (Dimensions==2 and evaluateSource) {
  #pragma omp target teams distribute device(targetDevice) \
              parallel for simd collapse(3)
  for (int patchIndex=0; patchIndex<numberOfCells; patchIndex++)
  for (int y = 0; y < numberOfVolumesPerAxisInPatch; y++)
  for (int x = 0; x < numberOfVolumesPerAxisInPatch; x++) {
   internal::copySolutionAndAddSourceTerm_LoopBody<Solver>(
    ...

f (Dimensions==2 and evaluateFlux) {
  #pragma omp target teams distribute device(targetDevice) \
              parallel for simd collapse(3)
  for (int patchIndex=0; patchIndex<numberOfCells; patchIndex++)
  for (int y = 0; y < numberOfVolumesPerAxisInPatch; y++)
  for (int x = -1; x < numberOfVolumesPerAxisInPatch+1; x++) {
   internal::evaluateFlux<Solver>(
    ...

  #pragma omp parallel for simd collapse(2)
  for (int patchIndex=0; patchIndex<numberOfCells; patchIndex++)
  for (int y = 0; y < numberOfVolumesPerAxisInPatch; y++)
  for (int x = 0; x < numberOfVolumesPerAxisInPatch; x++) {
   internal::updateWithFlux<Solver>(
    ...
}
```

- Hide temp enumeration
- Distribute loop over patches among SMs/warps
- Collapse internal d-dimensional/$d+1$-dimensional loops
- Use `parallel for` for collapsed loops

Outline

Durham University
Department of Computer Science

ExaHyPE

High-performance computing strategy

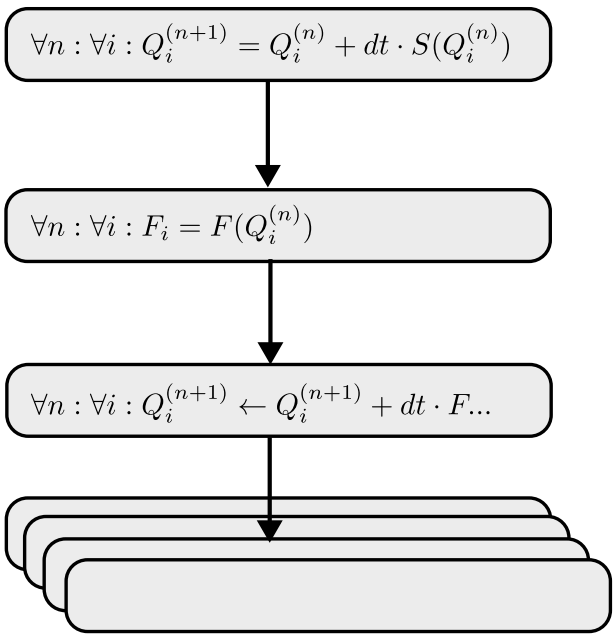
Finite Volumes

Migrating from OpenMP to SYCL

Conclusion

Batched kernels in SYCL

Durham University
Department of Computer Science



- Sequence of steps
  - OpenMP: Sequence of nested loops
  - SYCL: Sequence of kernels

  (ignore concurrency between steps)
- Parallel, collapsed loop
  - OpenMP: Collapse and `parallel for`
  - SYCL: `range`
- Tuning
  - OpenMP: annotations
  - SYCL: `nd_range` and partial serialisation
    (required anyway due to lack of 4d/5d ranges)
- Degenerated DAG

Patch-wise realisation in SYCL

Durham University
Department of Computer Science

```
::sycl::queue& queue = tarch::accelerator::getSYCLQueue(targetDevice);
[...]
double* tempFluxX ::sycl::malloc_device<double>(fluxEnumerator.size(), queue);
[...]
queue.submit(
 [&](::sycl::handler &handle) {
  handle.parallel_for(   // or parallel_for_work_groups
   ::sycl::range<1>{numberOfPatches},
   [=] (::sycl::item<1> i) {
    int patchIndex = i[0];

    if (Dimensions==2 and evaluateSource) {
     handle.parallel_for(   // or parallel_for_work_items
      ::sycl::range<2>{numberOfVolumesPerAxisInPatch,numberOfVolumesPerAxisInPatch},
      [&] (::sycl::item<2> i) {
       int x          = i[0];
       int y          = i[1];
       internal::copySolutionAndAddSourceTerm_LoopBody<Solver>(
       [...]
    if (Dimensions==2) {
     handle.parallel_for(
      ::sycl::range<2>{numberOfVolumesPerAxisInPatch,numberOfVolumesPerAxisInPatch+2*solversHaloSize},
      [=] (::sycl::item<2> i) {
       int y          = i[0];
       int x          = i[1]-solversHaloSize;
       internal::computeMaxEigenvalue_LoopBody<Solver>(
       [...]
```

- Multiple nested `parallel_for` regions within kernel
- Different iteration ranges

$\Rightarrow$ 1:1 translation not possible

A proper compute-DAG in SYCL

Durham University
Department of Computer Science

```
std::vector< ::sycl::event > copySolutionEvent(numberOfPatches);
if (Dimensions==2 and evaluateSource) {
 for (int patchIndex = 0; patchIndex<numberOfPatches; patchIndex++) {
  copySolutionEvent[patchIndex] = queue.submit(
   [&](::sycl::handler &handle) {
   handle.parallel_for(
    ::sycl::range<2>{numberOfVolumesPerAxisInPatch,numberOfVolumesPerAxisInPatch},
    [=] (::sycl::item<2> i) {
     int x          = i[0];
     int y          = i[1];
     internal::copySolutionAndAddSourceTerm_LoopBody<Solver>(
     [...]


  updateWithEigenvalueEvent[patchIndex] = queue.submit(
   [&](::sycl::handler &handle) {
    handle.depends_on(copySolutionEvent[patchIndex]);
    [...]
```

- Smallish compute kernels
- Individual dependencies

  (no address magic required)

Outline

Durham University
Department of Computer Science

ExaHyPE

High-performance computing strategy

Finite Volumes

Migrating from OpenMP to SYCL

Conclusion

Before we continue

Durham University
Department of Computer Science

- Paper on batching vs. patch-wise on CPUs:
  B. Li, H. Schulz, T. Weinzierl, H. Zhang: Dynamic task fusion for a block-structured finite volume solver over a dynamically adaptive mesh with local time stepping, ISC High Performance 2022, LNCS 13289, 2022, pp. 153–173
- Paper on GPU offloading:
  M. Wille, T. Weinzierl, G. Brito Gadeschi, M. Bader: Efficient GPU Offloading with OpenMP for a Hyperbolic Finite Volume Solver on Dynamically Adaptive Meshes, ISC High Performance 2023, LNCS (accepted)
- Benchmark suite: www.peano-framework.org
  - ExaHyPE 2 shipped extension of AMR framework Peano
  - Guidebook (pdf) part of repository
  - Benchmarks read-to-use
- Waiting for Intel GPUs
  (Codeplay release of NVIDIA bindings appreciated)

Conclusion and future work

Durham University
Department of Computer Science

- Features in SYCL missing (?)
  - Multiple parallel fors per kernel with different ranges
    (compute max range and add masking)
  - Ranges for $d \geq 4$
    (any implicit linearisation is **not** performance-portable)
- All realisations possible with all GPUisation approaches

$\Rightarrow$ Realisation language favors algorithm flavour

- Batched and patch-wise kernels only two out of many realisations

$\Rightarrow$ Graph focus (in theory) very powerful

- Performance
  - Graph-based approach outperform loop parallelism
  - Performance portable (spread over multiple cores, e.g.)

Academic Centers of Excellence
DiRAC High Performance Computing Facility
ExCALIBUR