Implementing High-Resolution Fluid Dynamics Solver in a Performance Portable Way
Applications to astrophysical compressible fluid dynamics

Pierre Kestener

CEA Saclay, DRF, Maison de la Simulation, FRANCE

GPU Technology Conference (GTC) 2017, San Jose, May. 8, 2017
Motivations: computational sciences and software engineering
Kokkos: library for performance portability
RamsesGPU: CFD applications for astrophysics
  Refactoring Hydrodynamics and MHD kernels
  Same performance between old CUDA kernels and new Kokkos Kernels?
Implementing high-order numerical schemes with Kokkos
Performance measurements on IBM Power8 + Nvidia Pascal P100
  OpenMP scaling on Power8 (device Kokkos::OpenMP)
  GPU performance on Pascal P100 (device Kokkos::Cuda)
Perspectives / Future applications and developments
Motivations of this work - 1

- **RAMSES-GPU** is developed in CUDA/C++ for **astrophysics applications** on regular grid
- ~ 70k lines of code (out of which ~ 16k in CUDA)
- Development started in **2009**!
- A lot of optimization techniques accumulated over the years are not so critically important anymore on today’s GPU; **both GPU hardware/sofware have tremendously evolved** (in orders of magnitude in memory bandwidth, number of registers per SM, c++11, ...)
- Collaborations with domain scientists are hard when required software skills include CUDA.

- **2016-2017** is the right time to refactor code, **sparkle new ways to develop scientific software at a higher abstraction level**

- **Science cases applications:**
  - **MRI in accretion disk** \((Pm = 4)\): (256 GPU) at 800 × 1600 × 800
  - **MHD Driven turbulence:** (Mach ~ 10): resolution 2016\(^3\) (486 GPUs)
Motivations of this work - 2

- **Computational science ground** - Computational Fluid Dynamics
  - **High-order numerical schemes** for **compressible hydrodynamics**
  - CFD - Euler system of partial differential equations
  - **How fast the numerical solution converges to the reference solution when increase space resolution?**

![Grids](image)

- For high-order numerical methods, one expects the error to decrease as $|f - f_r| \leq h^{-N}$
- **MOOD numerical schemes**, introduced in 2011 by Diot, Chain, Loubère; **very compute intensive** (ref: Diot PhD thesis)

<table>
<thead>
<tr>
<th>MOOD with</th>
<th>Machine 1</th>
<th>Machine 2</th>
<th>Machine 3</th>
<th>Memory storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>[PAD → DMP → t/2]</td>
<td>Intel Xeon E5335 @ 2.0GHz</td>
<td>Intel Core2Duo P7550 @ 2.26GHz</td>
<td>Intel Core i5 2500 @ 3.30GHz</td>
<td></td>
</tr>
<tr>
<td>MOOD-P_2</td>
<td>66µs/it./cell</td>
<td>57µs/it./cell</td>
<td>30µs/it./cell</td>
<td>0.4 GB</td>
</tr>
<tr>
<td>MOOD-P_3</td>
<td>183µs/it./cell</td>
<td>136µs/it./cell</td>
<td>69µs/it./cell</td>
<td>0.8 GB</td>
</tr>
<tr>
<td>MOOD-P_5</td>
<td>439µs/it./cell</td>
<td>385µs/it./cell</td>
<td>185µs/it./cell</td>
<td>3.0 GB</td>
</tr>
</tbody>
</table>

- Reference number to keep in mind ~ 1µs/it./cell: time to update a cell in a mesh (serial, CPU, low-order scheme).
Motivations of this work - 3

**Software engineering**
- Refactoring existing C++/CUDA code
- As much as possible **performance portable code**: write the code once, and let the user run it on the available target platform with performance as good as possible.
- Prefer a **high-level approach** among:
  - **Directive-based**: OpenACC, OpenMP
    - ease of use, incremental approach, for large legacy code bases, ...
  - **External smart library** implementing **parallel programming patterns** (for, reduce, scan, ....): [Kokkos](https://github.com/kokkos/kokkos), RAJA, agency, arrayFire libraries are such possibilities
    - parallel programming patterns as 1st class concepts, architecture adapted data containers, c++ integration / engineering, ...
  - **Other high-level approaches (more experimental)**: SYCL (Khronos Group *standard*), hpx (heavy use of new c++ standards (11,14,17): `std::future`, `std::launch::async`, distributed parallelism, ...)
C++ Kokkos library summary


- Framework for efficient **node-level parallelism**

- Provides some **high-level (abstract) concepts** as template C++ classes:
  - A **kokkos device**: Kokkos::Cuda, Kokkos::OpenMP, Kokkos::Pthreads, Kokkos::Serial, ...
  - concepts controlled by C++ template meta-programing: **execution space, memory space, memory layout, ...**
  - **Computationnal parallel patterns** (for, reduce, scan, ...) controlled with a **execution policy** (i.e. how many iterations, teams, ...)

- **Kokkos::View**: A **multi-dimensionnal data container with hardware adapted memory layout**
  - Kokkos::View<double **> data("data",NX,NY); // 2D array with sizes known at runtime
  - **How do I access data?** data(i,j)!

- Mostly a header library (C++ metaprograming)
C++ Kokkos library summary

- Most commonly in a C/C++, **multi-dimensionnal array access** is done through **index linearization** (row or column-major in 2D):

  \[ index = i + nx \cdot j \]

- In Kokkos, one should/must avoid this index linearization, let Kokkos::View do its job (decided at compile-time, hardware adapted):
  - 1D Kokkos::View with index linearization + 1D Iteration range
  - 2D Kokkos::View + 1D Iteration range (**used in this work**)
  - 2D Kokkos::View + 2D (Kokkos::MDRange Kernel policy) : still an experimental feature

- Kokkos::MDRange is functional, but was generating kernels with some performance loss, will surely be solved shortly by Kokkos core developers.

- See also new development on hierarchical task-data parallelism, session S7253 (Monday 8th, room 211B).
Compressible hydrodynamics: Euler system of equations

- **Euler equations** as conservative law system \( \partial_t \mathbf{U} + \nabla \cdot \mathbf{F}(\mathbf{U}) = 0 \)

\[
\begin{align*}
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) &= 0 \\
\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} \otimes \mathbf{v} + P \mathbf{I}) &= 0 \\
\frac{\partial \rho E}{\partial t} + \nabla \cdot [\mathbf{v}(\rho E + P)] &= 0
\end{align*}
\]

- (+ dissipative terms (viscous, resistive) + MHD with shearing box setup)
- Formal 1st order discretization:

\[
\mathbf{U}_{i}^{n+1} = \mathbf{U}_{i}^{n} + \frac{\Delta t}{|\mathbf{V}_{i}|} \sum_{j} |e_{i,j}| \mathbf{F}(\mathbf{\tilde{U}}_{i}, \mathbf{\tilde{U}}_{j})
\]

- In high-order scheme, use Runge-Kutta time integration + quadrature rules for computing the numerical fluxes \( \mathbf{F} \)
A Finite volume solver - MUSCL-Hancock

2nd order MUSCL-Hancock

- A priori limiting (to avoid spurious oscillations)
- Slope computations: linear reconstruction inside each cell
  \[ \delta U_i = \text{MINMOD}(U_i - U_{i-1}, U_{i+1} - U_i) \]
- Reconstruct states \( U_{left} \) and \( U_{right} \) on both sides of a given edge using limited slopes
- This numerical scheme is already available in C++/CUDA in RAMSES-GPU
- Refactored with Kokkos
High-order MOOD (Multi-Dim Optimal Order Detection)

- **A posteriori limiting**
- Introduced in 2011 by Clain, Diot and Loubère
- **Reconstructing multivariate polynomials of degree \( d \)**
  - Define a **stencil** large enough to perform a **least square estimation** of the \( n \)-dimensionnal **multivariate polynomial** interpolating cell-average values of \( U_j \) in **stencil**
  - if \( N \) is the number of cells in stencil, the linear system to solve (one per cell), using QR decomposition

\[
\begin{align*}
L_{i1} & = u_x \\
L_{i2} & = u_y \\
L_{i3} & = u_{xx} \\
& \quad \vdots \\
L_{iN} & = u_{yy} \\
\end{align*}
\]

\[
\begin{bmatrix}
L_{i1} & u_x \\
L_{i2} & u_y \\
L_{i3} & u_{xx} \\
& \quad \vdots \\
L_{iN} & u_{yy}
\end{bmatrix} =
\begin{bmatrix}
w_{i1}(u_1 - \bar{u}_i) \\
w_{i2}(u_2 - \bar{u}_i) \\
w_{i3}(u_3 - \bar{u}_i) \\
& \quad \vdots \\
w_{iN}(u_N - \bar{u}_i)
\end{bmatrix}
\]
A Finite volume solver - MOOD

High-order MOOD (Multi-Dim Optimal Order Detection)

- **Reconstructing multivariate polynomials of degree $d$**
- Least-square solve

\[
\begin{bmatrix}
L_{i1} & u_x \\
L_{i2} & u_y \\
L_{i3} & u_{xx} \\
\vdots & \vdots \\
L_{iN} & u_{yy}
\end{bmatrix}
\begin{bmatrix}
w_{i1} (\overline{u_1} - \overline{u_i}) \\
w_{i2} (\overline{u_2} - \overline{u_i}) \\
w_{i3} (\overline{u_3} - \overline{u_i}) \\
\vdots \\
w_{iN} (\overline{u_N} - \overline{u_i})
\end{bmatrix}
\]

- Matrix $L$ is **purely geometric** (independant of $U$) with $L_{ij} = (w_{ij}\hat{x}_{ij}, w_{ij}\hat{y}_{ij}...)$
- These geometrical terms are computed once for all in init

\[
\overline{x^n y^m}_{i,j} = \frac{1}{V_j} \int_{y_j} (x-x_i)^n(y-y_i)^m \, dr - \frac{1}{V_i} \int_{x_i} (x-x_i)^n(y-y_i)^m \, dr
\]
High-order numerical scheme comparison - WENO

- Comparison of scheme WENO3 / WENO5 for the Euler system, using TVDRK3 time integration.
- Small scale vorticity structures can be seen on the WENO5 solver.
- Time to solution for resolution $3200^2$ (CPU, serial)
  - **WENO3-S-VL:** 112 hours
  - **WENO5-S-VL:** 176 hours
High-order numerical scheme comparison - MOOD

MOOD $P_2$ - device Kokkos::Cuda

MOOD $P_4$ - device Kokkos::Cuda

Performed on system ouessant at IDRIS/GENCI, France.

- Comparison of schemes MOOD $P_2$ / MOOD $P_4$ for the Euler system, using TVDRK3 time integration, **made with our Kokkos implementation**
- Same resolution as before: $800^2$, $1600^2$, $3200^2$
- Small vorticity structures already start to form with MOOD $P_2$ at high resolution
- Time to solution for resolution $3200^2$ (1 GPU, Pascal P100)
  - MOOD $P_2$: 1.04 hours
  - MOOD $P_4$: 7.80 hours
Kokkos implementation

- While the overall code organization is the same, each computational kernel is tuned into a **Functor** (just a C++ class, with method `operator()` called by each device thread)
- Debug functionnality with `Kokkos::OpenMP` device, then activated `Kokkos::Cuda`
- **Used C++11 a few times**, when it seems to be a good idea, e.g. **generic code for 2D/3D**:

```cpp
#include <enable_if.h>
#include <conditional>

template<int dim, int degree, STENCIL_ID stencilId>
class ComputeFluxesFunctor : public MoodBaseFunctor<dim,degree> {
    public:
    // typedef DataArray here
    // operator() for 2d - put here the 'equivalent' cuda kernel code
    // operator() for 3d - put here the 'equivalent' cuda kernel code
    // ...
};
```
Power8 system has 2 sockets, 10-core CPU
- max num of hardware threads is 160

HT=1, hyperthreading is not activated, up to N=20 OpenMP threads
when HT2 activated, $N \leq 40$
when HT4 activated, $N \leq 80$
when HT8 activated, $N \leq 160$

From N=20 OpenMP threads to $20 \times 8$ threads, performance is multiplied by 2.1

Domain is $8192 \times 8192$

Performed on system ouessant at IDRIS/GENCI, France.
2D Muscl-Hancock in Kokkos - OpenMP scaling on Power8

- Power8 system has 2 sockets, 10-core CPU
- **intra-node performance comparison**
- old code (Ramses on CPU, MPI only)
- new code (refactored with Kokkos)
- **gray line** is intra-node MPI scaling of RamsesGPU running on CPU-only
- When Hyper-threading is activated, and MPI ranks increased to 160, we recover the same performance between old (RamsesGPU) and new (Kokkos) code on Power8.
- Domain is $8192 \times 8192$
- Performed on system ouessant at IDRIS/GENCI, France.
3D Muscl-Hancock in Kokkos - OpenMP scaling on Power8

- **Power8 system has 2 sockets, 10-core CPU**
- **HT=1**, hyperthreading is not activated, up to $N=20$ OpenMP threads
  - when **HT2** activated, $N \leq 40$
  - when **HT4** activated, $N \leq 80$
  - when **HT8** activated, $N \leq 160$
- From $N=20$ OpenMP threads to $20 \times 8$ threads, performance is multiplied by 1.7
- Domain is $256^3$
- Performed on system **ouessant** at **IDRIS/GENCI**, France.
2D MOOD schemes in Kokkos - perf on IBM Power8 + Nvidia GPU P100

- On average **Pascal P100** is $\times 2.8$ to $\times 4.0$ faster than **Kepler K80** (single GPU), no special optimization, just rebuild with architecture flags.
- **Pascal P100** is $\sim \times 10$ faster than **Power8 - HT8**
- Mood $P_3$ on all architectures below sm_60 generates a weird runtime error (to be analyzed)
- 2nd-order MUSCL (2D / 3D) performance in Kokkos are 2 to 5% slower compared to hand-written CUDA kernels in RamsesGPU
- Performed on system ouessant at IDRIS/GENCI, France.
Conclusion

- **Refactored an existing C++/CUDA application using Kokkos**
  - much better global software design
  - high-level concept (no CUDA), focus on parallel computing pattern (for, reduce, scan, ...)
  - should be easier to convince domain scientist to have a PhD student get started
  - C++11 + template: a key to generic cleaner code

- **Learned a lot about C++11** (generic code, type traits, ...)

- **Sparkled a new code**, temporary name is ppkMHD (performance portable kernels for MHD), with most RamsesGPU kernels + new high-order numerical schemes (~ 18k SLOC).

- **Futur**:
  - Implement MOOD schemes for MHD
  - Explore variants of implementation for 3D MOOD, flux operator implemented in Kokkos
  - Integrate high-order scheme in a block-structured adaptive mesh refinement (AMR) code
A few minor things not supported by nvcc: constexpr support in nvcc is a bit behind g++; e.g.

- Can't make a constexpr array (would have been useful, short to store parameters know at compile time for quadrature weights, ...)
- Can't used std::array or Kokkos::Array as constexpr for use in device code with CUDA backend

```cpp
// in header1
constexpr int STENCIL_SIZE[STENCIL_TOTAL_NUMBER] = { ... };

// in header2 (Kokkos functor definition)
template<int id>
class Functor {
public:
  static constexpr int stencil_size = STENCIL_SIZE[id];
  ...
};

error at line 8: expression must have a constant value

// not supported by nvcc
constexpr double SQRT_2 = sqrt(2.0);
```