ECCOMAS Congress 2016
VII European Congress on Computational Methods in Applied Sciences and Engineering
M. Papadrakakis, V. Papadopoulos, G. Stefanou, V. Plevris (eds.)
Crete Island, Greece, 5–10 June 2016

BUILDING A MORE EFFICIENT LAGRANGE-REMAP SCHEME THANKS TO PERFORMANCE MODELING

Thibault GASC[1,2,3], Florian De VUYST[2], Mathieu PEYBERNES[4], Raphaël PONCET[5] and Renaud MOTTE[3]

[1]Maison de la Simulation USR 3441
CEA Saclay, F-91191 Gif-sur-Yvette
e-mail: thibault.gasc@cea.fr

[2] CMLA, ENS Cachan, CNRS, Université Paris-Saclay,
94235 Cachan, France
e-mail: devuyst@cmla.ens-cachan.fr

[3]CEA DAM DIF
F-91297 Arpajon

[4] CEA, DEN, DM2S, STMF
CEA Saclay, F-91191 Gif-sur-Yvette
e-mail: mathieu.peybernes@cea.fr

[5] CGG
27 Avenue Carnot, F-91300 Massy
e-mail: raphael.poncet@cgg.com

Keywords: Performance Modeling, Co-design, Lagrange-Remap, Roofline model, Execution Cache Memory Model

Abstract. *This paper is a practical example of co-design between numerical analysis and high performance computing, applied to compressible fluid mechanics. We consider a legacy numerical method, based on a Lagrange-Remap solver for the compressible Euler equations, use tools from analytical performance modeling to quantitatively understand its behavior on recent multicore CPUs, and extract its computational bottlenecks. This analysis inspires us to propose a new numerical method, called Lagrange-Flux. Experimental results show that this new method yields an algorithm that is more computationally efficient, and at the same time retains the good numerical properties of the original Lagrange-Remap solver.*

1 Introduction

This paper is motivated by the adaptation and modification of a legacy numerical method arising in compressible fluid mechanics — the Lagrange-Remap solver —, in order to devise an alternative algorithm that is numerically equivalent, but more computationally efficient on current typical computing hardware: multicore CPUs. This kind of solver is used to approximate the compressible Euler equations, and is of particular interest for industrial applications involving complex compressible flows. It is for instance used in numerical simulation of plasma, high-speed flows of elastoplastic materials, and multi-material flows [1].

Computational efficiency of a given piece of code on a given hardware can be readily evaluated by measuring the runtime of this code. It is then straightforward to evaluate relative performance of two algorithms solving the same problem by comparing their results on the same benchmark, for both numerical quality and runtime. However, the validity of these results become tied to the particular hardware used for the tests. It is a problem, since in general, computing hardware evolves much more quickly other time than numerical methods (especially in the case of algorithms used in legacy industrial codes).

Hence, we propose to use a more future-proof, quantitative and general methodology to assess algorithm computational efficiency: we use a theoretical performance model to predict algorithm performance. This model depends on the algorithm itself, and on a simplified hardware description using a few parameters (typically less than 10). Hence, similarly to the scientific method applied to physics or engineering, if our performance model is rich enough to adequately capture the behaviour of our algorithm, it can be used not only for quantitative prediction, but also for precise qualitative understanding. In a previous work [2], we established that the so-called ECM model [3] has the capability to accurately predict Lagrange-Remap solver performance on multicore CPUs.

The main contribution of this paper is to show how performance modeling using the ECM model leads us to extract the computational bottlenecks of the Lagrange-Remap algorithm, link them to properties of the numerical method, and naturally propose the Lagrange-Flux algorithm (originally introduced in [4] from a numerical analysis point view) as an alternative. Superior computational performance of this new algorithm — from both a scalability and absolute performance point of view — is then inferred from theoretical analysis, and verified by numerical experiments. Up to our knowledge, this is the first published work in computational fluid dynamics using analytical performance models as a quantitative methodology for carrying out co-design between numerical analysis and high performance computing.

This paper is organized as follows: in section 2, we detail the analytical performance models used in this study for predicting and understanding algorithm runtimes. In section 3, we briefly recall the formulation of the legacy Lagrange-Remap solver, sum up the results of its performance analysis carried out in our previous work [2], and deduce its performance bottlenecks. In section 4 we briefly introduce the Lagrange-Flux solver, conduct its performance analysis, theoretically show its computational advantages over the Lagrange-Remap solver, and verify the better performance and scalability of this new algorithm. Finally we discuss the interest of our methodology and some perspectives brought by this work in section 5.

2 Performance modeling

2.1 Introduction

Performance modeling aims at quantitatively predicting and understanding code performance — in terms of runtime — using simple analytical models for both the algorithm itself, and the hardware on which the algorithm is executed. Because of the complexity of recent computing hardware, it is not a simple task.

Indeed, let us consider the simplest performance model one could think of for predicting a given algorithm runtime on a given machine: count the number of arithmetic operations (such as additions, multiplications or divisions) the algorithm performs, and compare it with the maximum — or *peak* — number of operations the hardware can execute per unit of time (which can be readily obtained using information from hardware vendor specification sheets, e.g. frequency and peak number of arithmetic instructions per hardware cycle). In most of the case, it will yield a very inaccurate number, for several reasons:

- the algorithm runtime is determined not only by the arithmetic operations it performs, but also the rate at which it can fetch data to/from memory. The bottlenecks of most algorithms on recent hardware correspond to the latter case (we talk of *memory bound* algorithms, compared to *compute bound* algorithms whose performance is bounded by the rate at which the hardware can process arithmetic instructions).

- the memory subsystem of any recent architecture is complex: it consists in several hierarchical layers of memory, such as caches for CPUs, with various capacities and bandwidths.

- all arithmetic instructions are not equal performance wise: for instance, divisions and square roots typically have a throughput at least one order of magnitude lower than additions or multiplications.

- last, but not least, any recent piece of hardware has several layers of parallelism. For instance, common multicore CPUs such as the one considered in this paper have three levels of parallelism: they consist in several cores, each core having single instruction multiple data — SIMD — vectorized units (such as AVX for recent Intel CPUs). Finally, each core has several pipelines executing instructions in parallel (this is called instruction level parallelism — ILP). Moreover, this parallelism is tied to the memory subsystem: for instance, some memory layers (usually the L1 and L2 caches) are private to CPU cores, but some (usually the L3 cache and the DRAM) are shared between all cores.

This underlying hardware complexity implies that precisely predicting and understanding the performance of even deceptively simple algorithms such as the Schönauer Vector Triad $A = B + C * D$ (where A, B, C and D are arrays) on a typical multicore CPU is not trivial [3].

However, some successful analytical performance models have been recently popularized in order to assess algorithmic performance on recent hardware. The most known model is the so-called Roofline model ([5]), which has been initially proposed as a conceptual qualitative model characterizing the behavior of an algorithm as compute-bound (CB) or memory-bound (MB).

2.2 The Roofline model

In the following, we consider elementary algorithms called *kernels*, acting on floating point data. Hence, we use the *billion floating point operations per second* (GFlop/s) metric for reporting Roofline performance, as is commonly done. The advantage of this metric is that it is easy to directly compare algorithm performance with peak CPU arithmetic throughput which is also given in GFlop/s, and thus get an insight on kernel efficiency on a given machine. The Roofline model approximates the performance of an algorithm with the following single formula:

$$P = \min(\mathrm{P}_{\mathrm{max}}(\mathrm{P}_{\mathrm{peak}}), \mathrm{AI} * \mathrm{B}_{\mathrm{max}}), \tag{1}$$

where $\mathrm{P}_{\mathrm{peak}}$ is the machine peak arithmetic throughput (in GFlop/s), $\mathrm{P}_{\mathrm{max}}$ is the kernel maximum arithmetic throughput (in GFlop/s), AI is the kernel arithmetic intensity (in Flop/Byte), and $\mathrm{B}_{\mathrm{max}}$ is the maximum machine bandwidth (in GByte/s). Parameter AI only depends on the algorithm (and can be readily obtained by counting the ratio between the *number* of arithmetic instructions and transferred data), whereas parameter $\mathrm{P}_{\mathrm{peak}}$ only depends on the machine. Several choices are possible for parameter $\mathrm{B}_{\mathrm{max}}$ and for the function $\mathrm{P}_{\mathrm{max}}(\mathrm{P}_{\mathrm{peak}})$. The basic version of Roofline takes for $\mathrm{B}_{\mathrm{max}}$ the peak CPU bandwidth provided by the vendor, and determine $\mathrm{P}_{\mathrm{max}}(\mathrm{P}_{\mathrm{peak}})$ by manually counting algorithm arithmetic instructions. However, this yields an overly optimistic model which is rarely quantitatively accurate. Hence, we use a refined version of the Roofline model (sometimes called *Roofline with ceilings*), where $\mathrm{B}_{\mathrm{max}}$ is chosen to be the effective bandwidth of the full CPU system measured using a STREAM benchmark [6]. Moreover, $\mathrm{P}_{\mathrm{max}}(\mathrm{P}_{\mathrm{peak}})$ is determined using static analysis, using the `Intel IACA` tool [7] in throughput mode (thus ignoring any latency effects). In particular, the obtained maximum arithmetic throughput takes into account the heterogeneous throughput of arithmetic instructions, and their concurrent scheduling on the core pipelines. In this refined version of the model, the function $\mathrm{P}_{\mathrm{max}}(\mathrm{P}_{\mathrm{peak}})$ is part of the model. We must emphasize that it does not come from execution of the kernel, but from *static inspection* of the binary by `Intel IACA`.

This Roofline has a simple interpretation: it is an optimistic model that simply expresses the fact that, for a kernel to be executed, data has to be fetched to/from memory, which corresponds to the $\mathrm{AI} * \mathrm{B}_{\mathrm{max}}$ term, and kernel computations have to be performed, which corresponds to the $\mathrm{P}_{\mathrm{max}}(\mathrm{P}_{\mathrm{peak}})$ term. Hence, kernel performance is necessary less than the minimum of these two terms. Roofline assumes that kernel performance can be approximated by this asymptotic value.

This model has enjoyed wide success [5], nevertheless it has some limitations. First, it does not accurately model caches. The consequence is that single core performance is not quantitatively accurate for memory-bound kernels (it is too optimistic). Hence, in particular, multicore scalability can not be inferred from this model [3, 8]. Moreover, the GFlop/s metric which is generally used with Roofline models has some drawbacks from a practical methodological point of view. First, it is not directly correlated with a meaningful performance metric for algorithm developers (who are generally interested in the number of loop iterations done per unit of time). Moreover, in some cases, the Flop/s hardware counters on recent multi-core CPUs — such as the Sandy Bridge processor — are known to be unreliable. The ECM — Execution Cache Memory — model has been recently introduced to overcome these drawbacks.

2.3 The ECM model

The ECM model ([3], [8]) is a refinement of the Roofline model for multicore CPUs that still neglects any latency effects, but takes into account the cache hierarchy. It uses the *cycles per cacheline worth of data* (cy/CL) performance metric. A *cacheline worth of data* corresponds, in a loop-based algorithm, to the number of loop elements that fit in a cacheline (64 Bytes in most modern CPU architectures, such as the one we used in this paper). For instance, for algorithms using double precision — corresponding to 8 byte storage —, a cacheline worth of data is 8 elements for scalar non-vectorized code, but 2 elements for AVX vectorized code (because AVX registers are 4-Byte wide). This metric is interesting because, on the one hand, it is directly correlated with the number of loop iterations per unit of time, and on the other hand, it is a reliable metric that can be measured on any CPU (because it can be derived directly from CPU runtime), and does not depend on hardware counters.

As in [3], [8], [9], we assume that STORE and floating point arithmetic instructions are overlapped with the data transfers between different levels of the memory system, whereas LOAD operations do not overlap. The data transfers between the L1 and L2 caches, and between the L2 and L3 caches, depends on the architecture and are given in table 1. The data transfer transfer rate $B_{\text{L3}\to\text{mem}}$ (in cy/CL) between the L3 cache and memory subsystem depends on the CPU frequency f (in GCycles/s) and the sustainable bandwidth $B_{\max}$ (in GBytes/sec), and is given by the following formula:

$$B_{\text{L3}\to\text{mem}} = \frac{64f}{B_{\max}}. \tag{2}$$

In practice, ECM model delivers a prediction of the number of CPU cycles required to execute a certain number of iterations of a given loop on a single core. Then, the prediction time T_{ECM} is given by $T_{ECM} = \max(T_{OL}, T_{nOL} + T_{data})$ ([3], [8]), where T_{data} is the transfer time through the memory hierarchy (L1 upwards), T_{nOL} the time taken by the load instructions (we assume that these instructions can not overlap with any memory transfer) and T_{OL} the time for all other instructions (store and arithmetic instructions). As for the refined Roofline model, the times T_{OL} and T_{nOL} are determined using `Intel IACA`. For data in memory, the time T_{data} is given by $T_{data} = T_{L1L2} + T_{L2L3} + T_{L3Mem}$, where T_{L1L2}, T_{L2L3} and T_{L3Mem} are respectively the transfer times between L1 and L2, L2 and L3 and L3 and memory. Transfer times for streaming loads and stores can be computed readily using micro architecture knowledge (see table 1 for the parameters used in this study). We refer to [9], [2] for the estimation of performance for stencil-like access patterns.

3 Identifying the legacy Lagrange-Remap solver computational bottlenecks using performance modeling

In this section, we first recall the main findings of the performance analysis we carried out for the Lagrange-Remap algorithm in [2]. We then use these results to identify the bottlenecks of this algorithm.

3.1 Description of the legacy Lagrange-Remap solver

We consider a cartesian Lagrange-Remap solver with the following features:

- discrete thermodynamic variables (mass, internal energy, pressure) are cell-centered variables;
- discrete velocity are node-base variables (this helps for physics coupling, and mesh distortion management);
- the lagrangian step is performed using a 2^{nd} order in time integration scheme of leap-frog type;
- the remap step is performed using an alternating direction strategy;
- second order in space accuracy is obtained thanks to a MUSCL-type reconstruction during the remap step;

- for shock capturing and entropy consistency, pseudo-viscosity is used.

This solver has been described in [1, 10, 11]. Its implementation is organized as follows: at the start of each time step, the mass (or density), the energy (or pressure) and the velocity fields are known. The lagrangian step is performed to obtain intermediate updated lagrangian fields of these variables. Then to go back to the fixed cartesian grid, a remap step is performed in each direction. Each step consists of several kernels. Other secondary variables are also introduced. Figure 1 is a dataflow diagram for the idealized Lagrange-Remap solver studied in [2] that shows dependencies between kernels and data for the lagrangian step and the remap step in one direction. Part of the complexity of these graphs are due to the use of staggered variables, as we explain in section 3.2.

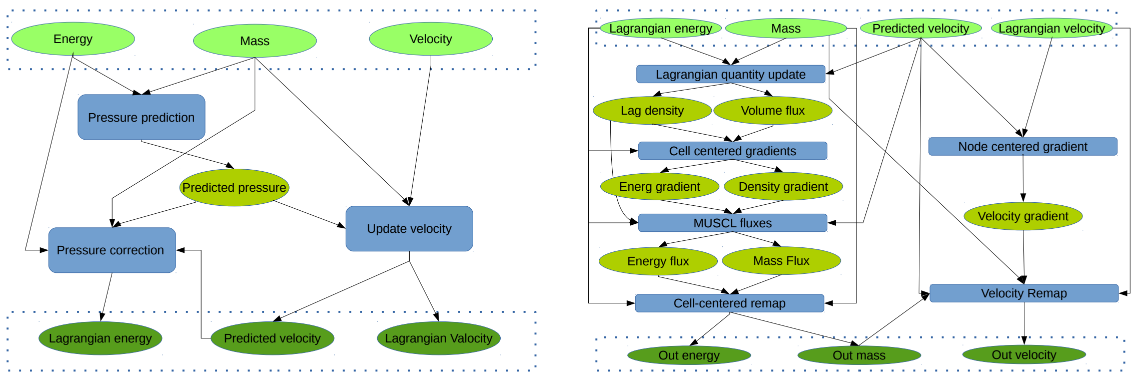

Figure 1: Dataflow diagrams of the lagrange (left) and remap (right parts of the algorithm. Kernels are represented in blue, input data in light green, output data in dark green, and temporary data in khaki green. Courtesy of [2]

3.2 Performance modeling of the Lagrange-Remap solver

The main result of paper [2] is that the performance of all individual kernels of our solver can be theoretically predicted with single digit accuracy on multicore CPUs, if the ECM model is used. The reported mean and median errors between model prediction and measurement lie between 3% and 8%. Tests have been conducted using Haswell and SandyBridge micro-architecture, for input data in L3 cache and main memory, and for scalar, vectorized, and multithreaded & vectorized versions of each kernel.

This validates the relevance of the ECM model for finely assessing performance of the Lagrange-Remap solver algorithm (and by extension, of most explicit solvers for fluid mechanics on cartesian meshes). Moreover, a closer look at the results allows us to identify three performance bottlenecks, and, more importantly, to link them with numerical properties of the numerical method:

1. using **staggered variables**;
2. using an **alternate direction** strategy for the remap;
3. taking into account **geometry** of the lagrangian mesh.

Staggered variables Most kernels are stencil type kernels. This means output data at a given grid location is obtained from input data at this location and its nearest neighbours. In the Lagrange+Remap solver, staggered variables are used (see figure 2). Hence, most kernels alternate reading and writing data between the primal cell-centered grid, and the dual node-centered grid. For this reason, successive kernels can not be easily merged. Thus, we have multiple small kernels in which arithmetical intensity is low, and data reuse is limited. As a consequence, most kernels are memory-bound, which yields suboptimal performance and low scalability.

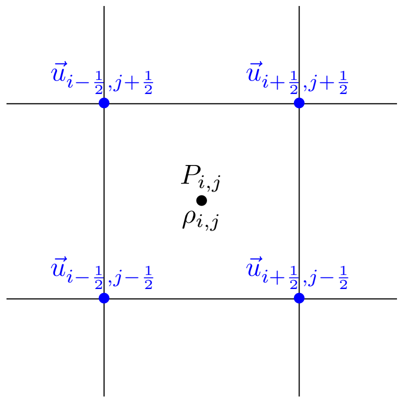
$\vec{u}_{i-\frac{1}{2},j+\frac{1}{2}}$
$\vec{u}_{i+\frac{1}{2},j+\frac{1}{2}}$
$P_{i,j}$
$\rho_{i,j}$
$\vec{u}_{i-\frac{1}{2},j-\frac{1}{2}}$
$\vec{u}_{i+\frac{1}{2},j-\frac{1}{2}}$

Figure 2: Variable positions for the legacy staggered Lagrange-Remap solver. Pressure and density are stored at cell centers, velocity is stored at node centers.

Alternate directions Using an alternate direction strategy for the remap step, e.g. a succession of 1D remap steps, introduces an extra intermediaty states, which adds stress on memory transfers compared to a true multidimensional remap step, which is already the bottleneck of most of our kernels [2]. Since memory bandwidth is shared among multiple cores, it also lowers multi-core scalability.

Lagrangian geometry Since the mesh is moving, lengths of the edges of the mesh are changing. Thus it is not possible to pre-compute useful geometrical data and this can lead to costly computations. The most significant example in the numerical scheme is the computation of gradients during the remap step. Indeed computing $\frac{q_i - q_{i-1}}{x_i - x_{i-1}}$ is more expensive if we cannot precompute once and for all $\frac{1}{x_i - x_{i-1}}$ because it is not constant. If it was, we could save the result and perform a multiplication instead of a division, which is typically an order of magnitude faster.

4 The Lagrange-Flux solver

4.1 Lagrange-Flux numerical scheme

The detailed construction of the Lagrange-Flux algorithm is given in [4]. Hereafter, we give key ideas of its construction and provide a schematic description. From the numerical properties of the original Lagrange-Remap solver and the conclusions of its performance analysis presented in section 3, we extract the following desirable properties constraining the design of the Lagrange-Flux solver:

1. A lagrangian solver is used (for multi-material flows or physics coupling);

2. A cell centered description is used rather than a staggered one, this will allow kernel fusion optimization and reduce data communication;

3. A direct multidimensional remap strategy is used rather than alternating direction strategy, this should also reduce data communications;
4. The method can be extended to second order accuracy in space and time, matching the accuracy of the legacy Lagrange-Remap solver;
5. Complex geometry related computations should be limited;
6. 3D extension of the solver should remain simple.

Let us now briefly describe the proposed scheme. We consider the compressible Euler equations for two-dimensional problems. Denoting ρ, $\boldsymbol{u} = (u_i)_i$, $i \in \{1,2\}$, p and E the density, velocity, pressure and specific total energy respectively, the mass, momentum and energy conservation equations can be written in the compact following form:

$$\partial_t U_\ell + \nabla \cdot (\boldsymbol{u} U_\ell) + \nabla \cdot \boldsymbol{\pi}_\ell = 0, \quad \ell = 1, \ldots, 4, \tag{3}$$

where $U = (\rho, (\rho u_i)_i, \rho E)$, $\boldsymbol{\pi}_1 = \vec{0}$, $\boldsymbol{\pi}_2 = (p, 0)^T$, $\boldsymbol{\pi}_3 = (0, p)^T$ and $\boldsymbol{\pi}_4 = p\boldsymbol{u}$. For the sake of simplicity, we will close this system using a perfect gas equation of state $p = (\gamma - 1)\rho(E - \frac{1}{2}|\boldsymbol{u}|^2)$, $\gamma \in (1, 3]$.

We first split the usually considered 2 steps Lagrange-Remap scheme in 3 steps. The lagrange step is taken identical, the remap step is split in a backward lagrangian motion followed by a forward advection step described in the eulerian frame.

This leads to the following scheme:

$$\begin{aligned} (U_\ell)_K^{n+1} = (U_\ell)_K^n \quad &- \quad \frac{\Delta t^n}{|K|} \sum_{A \subset \partial K} |A| \left(\frac{|A^{n+\frac{1}{2},L}|}{|A|} (\boldsymbol{\pi}_\ell)_A^{n+\frac{1}{2},L} \cdot \nu_A^{n+\frac{1}{2},L} \right) \\ &- \quad \frac{\Delta t^n}{|K|} \sum_{A \subset \partial K} |A| \left((U_\ell)_A^{n+\frac{1}{2},\star} (\boldsymbol{v}_A^{n+\frac{1}{2}} \cdot \nu_A) \right). \end{aligned} \tag{4}$$

where K is the cell area, A an edge of the cells, ν_A the unit normal vector of the edge A pointing outwards, $\cdot^L$ superscript denotes elements of the lagrangian mesh, $\cdot^{n+\frac{1}{2}}$ superscript denotes elements at time $t^{n+\frac{1}{2}}$ used for targeted second order in time accuracy, and $\boldsymbol{v}_A^{n+\frac{1}{2}}$ denotes the velocity of the grid used in the lagrangian step.

From this intermediate numerical scheme scheme we would like to suppress the geometrical computation of lagrangian geometry. By making Δt go to zero, ($t > 0$), we have $A^{n+\Delta t/2,L} \to A$, $(\boldsymbol{\pi}_\ell)^{n+\Delta t/2,L} \to \boldsymbol{\pi}_\ell$, $\boldsymbol{v}^{n+\Delta t/2} \to \boldsymbol{u}$, $(U_\ell)^\star \to U_\ell$; we obtain a partial discretization in space of the conservation laws which can be seen as a method of lines:

$$\frac{d(U_\ell)_K}{dt} = -\frac{1}{|K|} \sum_{A \subset \partial K} |A| \left((\boldsymbol{\pi}_\ell)_A \cdot \nu_A \right) - \frac{1}{|K|} \sum_{A \subset \partial K} |A| \, (U_\ell)_A \, (\boldsymbol{u}_A \cdot \nu_A). \tag{5}$$

This can be also be written as a finite volume method:

$$\frac{dU_K}{dt} = -\frac{1}{|K|} \sum_{A \subset \partial K} |A| \, \Phi_A,$$

with a numerical flux Φ_A whose components are

$$(\Phi_\ell)_A = (U_\ell)_A \, (\boldsymbol{u}_A \cdot \nu_A) + (\boldsymbol{\pi}_\ell)_A \cdot \nu_A. \tag{6}$$

In (5), pressure fluxes $(p_\ell)_A$ and interface velocities $\boldsymbol{u}_A$ are computed from an approximate Riemann solver in lagrangian coordinates (for example the lagrangian HLL solver, see [12]). Then, the interface states $(U_\ell)_A$ are computed from a upwind process according to the sign of the normal velocity $(\boldsymbol{u}_A \cdot \nu_A)$. Higher-order accuracy in space can be achieved using a standard MUSCL reconstruction with a slope limiting process. In this semi-discrete formalism, there is no time discretization, thus all kernel act on the eulerian mesh and fluxes are defined at the edges the eulerian cells. To achieve second-order accuracy in time, one can apply any second-order time advance scheme. The second-order Heun scheme for example leads to the following algorithm:

1. Compute the time step Δt^n subject to the CFL stability condition;

2. Predictor step:

   (a) Reconstruct higher order interpolation from the discrete values U_K^n (MUSCL + slope limitation): compute a discrete gradient for each cell K ;

   (b) Compute interface velocities $\boldsymbol{u}_A^n$ and pressure fluxes $\boldsymbol{\pi}_A^n$ using a lagrangian approximate Riemann solver;

   (c) Select the upwind edge values $(U_\ell)_A^n$ according to the sign of $(\boldsymbol{u}_A^n \cdot \nu_A)$;

   (d) Compute the numerical flux Φ_A^n as defined in (6);

   (e) Compute the first order predicted states $U_K^{\star,n+1}$:

$$U_K^{\star,n+1} = U_K^n - \frac{\Delta t^n}{|K|} \sum_{A \subset \partial K} |A| \; \Phi_A^n;$$

3. Corrector step:

   (a) Repeat steps (2a-2d) to compute the numerical flux $\Phi_A^{\star,n+1}$ from the predicted state $U_K^{\star,n+1}$;

   (e) Compute the second-order accurate states U_K^{n+1} at time t^{n+1}:

$$U_K^{n+1} = U_K^n - \frac{\Delta t^n}{|K|} \sum_{A \subset \partial K} |A| \; \frac{\Phi_A^n + \Phi_A^{\star,n+1}}{2}.$$

This numerical scheme fullfills the design specifications presented at the beginning of this section. It has been validated using several test cases. We present the results of such a test in figure 3: numerical quality of the method is assessed by comparing it to the original Lagrange-Remap solver on a triple point type computation.

4.2 Implementation details of the Lagrange-Flux solver

The implementation of the Lagrange-Flux solver is quite simple compared to the legacy Lagrange-Remap solver. Excluding the computation of the time step common to both solvers, we first derive a naive implementation by simply following the algorithm description given at the end of the previous section. At this point, we start optimizing the code. Since the Lagrange-Flux solver uses collocated cell-centered variables, we are able to merge multiple small kernels into bigger ones. In the most simple case, instead of doing computations for kernel K_1, writing data to memory, reloading it immediately after, and doing the computations for kernel K_2, we

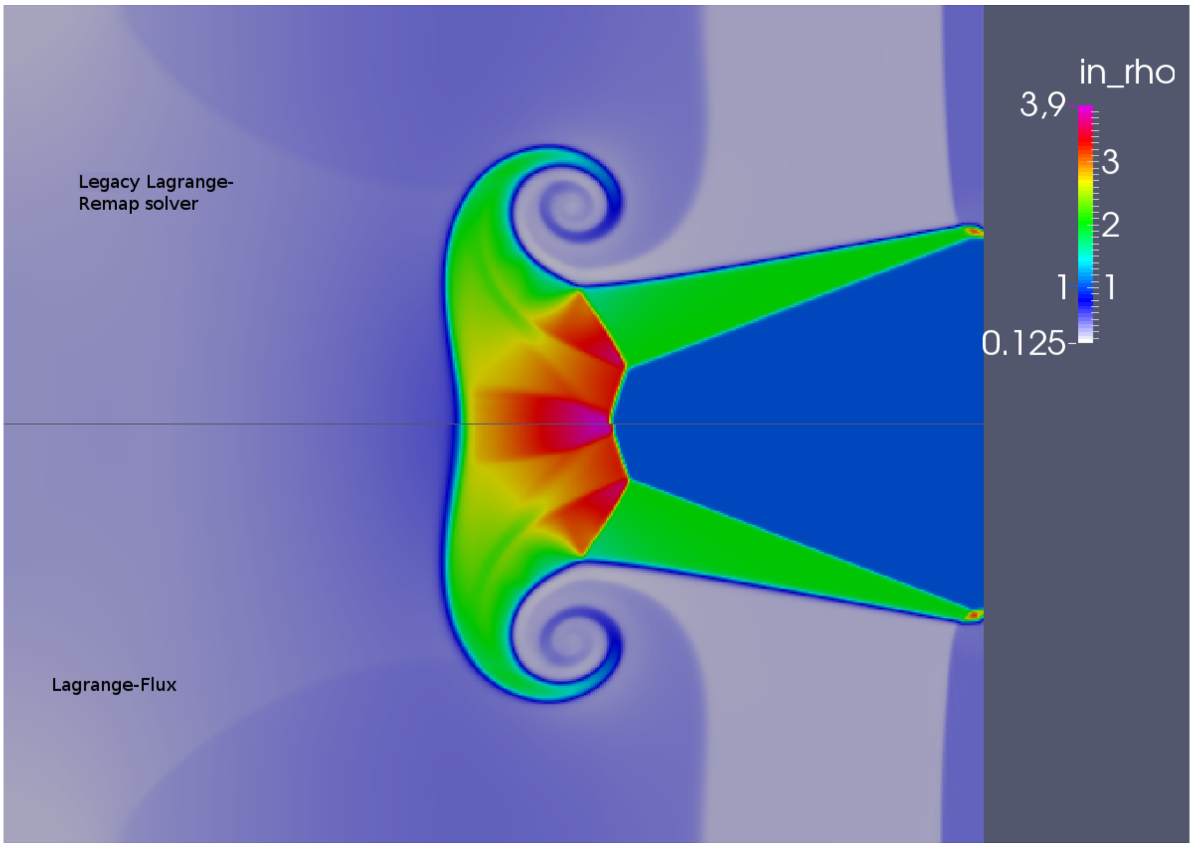

Figure 3: Comparison of the Lagrange-Remap solver and the Lagrange-Flux solver on a triple point test case. In this two-dimensional Riemann problem, we check that the shock wave and the vortex are correctly captured. Density maps at time $t = 3.5s$ are presented for the legacy Lagrange-Remap solver (upper part) and the Lagrange-Flux solver (lower part) on a 560×240 mesh. One can notice that both solvers capture the same physical phenomenons.

load the data once and do the computations for both kernels K_1 and K_2. This mechanically increases arithmetic intensity. In more complex cases, there is a compromise to be made: if the result of a given kernel K is reused by N other kernels, merging can be done, at the cost of doing kernel K computations N times. In this case, merging is beneficial only when the time needed for extra computation is shorter that the time saved from increased memory reuse. For the Lagrange Flux algorithm, it turns out that merging aggressively is the best option.

At the end of the optimization process, we are left with only two large kernels which are almost identical: one for the prediction step, and one for the correction step (see figure 5). In these kernels, density, velocity and energy are updated at the same time using the same stencil (see figure 4). Comparing figures 5 and 1, we can expect that the Lagrange-Flux solver is less memory-bound than the original Lagrange-Remap solver. The following section confirms that this is indeed the case.

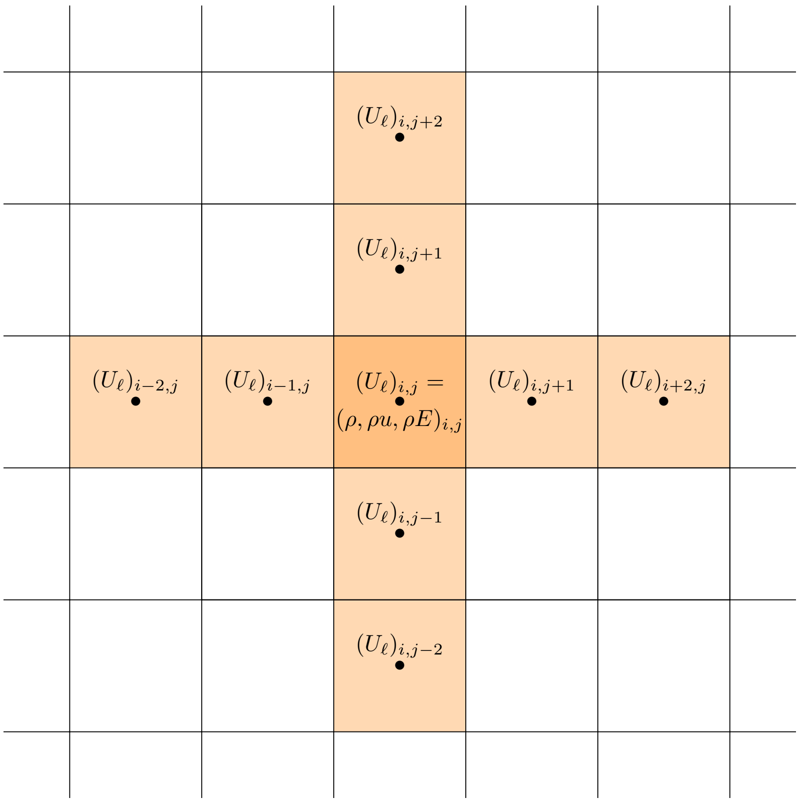

Figure 4: Stencil of the Lagrange-Flux kernels. The target updated cell is in dark orange, the full neighborhood is in lighter orange. All variables (density, energy and velocities) are loaded from the same stencil. Computations for their update are performed in the same time and many common intermediary results can be reused without being stored in the main memory.

4.3 Performance analysis and measurements

In this section, we prove that the ECM model has predictive capabilities for both the full Lagrange-Remap solver, and its Lagrange-Flux alternative. The machine used for the following tests is a dual socket octo-core SandyBridge processor (see 1).

First, in table 2, we present a comparison between model prediction and measurements on a single core, without vectorization. We observe quantitative agreement for both solvers. A closer look at the Lagrange-Flux performance model output shows that the divide port is the bottle-

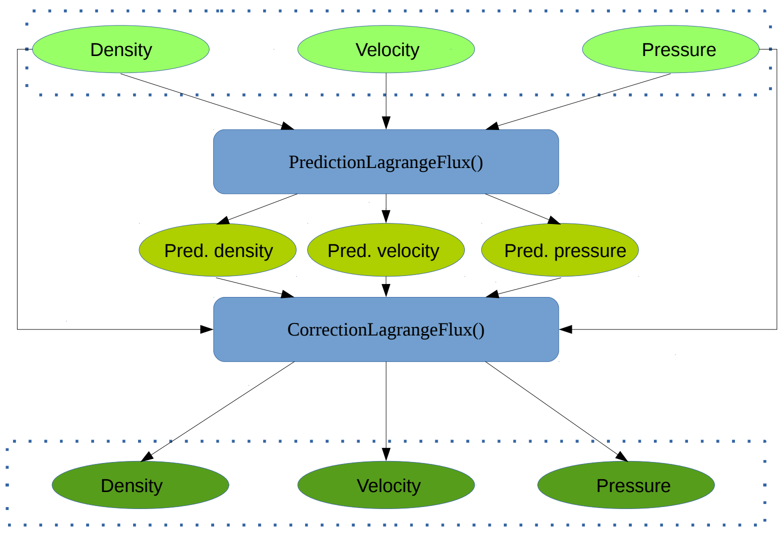

Figure 5: Dataflow diagrams of the Lagrange-Flux solver. Kernels are represented in blue, input data in light green, output data in dark green, and temporary data in khaki green

Table 1: Characteristics of the CPU hardware used.

Micro Architecture	**Intel SandyBridge**
Model	E5-2670
Number of cores	2x8
Clock (GHz)	2.6
Peak Flops DP (GFlop/s) (full socket)	168
Cache sizes: L1	8x32kB
L2	8x256kB
L3	20MB
Theoretical Bandwidth (GBytes/s)	51.2
Measured triad Bandwidth $B_{\max}$ (GBytes/s)	38.4
L1 $\leftrightarrow$ L2 bandwidth (cy/CL)	2
L2 $\leftrightarrow$ L3 bandwidth (cy/CL)	2

Table 2: Comparison between single core non-vectorized ECM prediction and measurements for the intel Sandy-Bridge architecture. Performance is expressed in cycle per cache line worth of data (cy/CL), and kernel type can be CB — Compute-Bound — or MB — Memory-Bound. The analytical ECM model gives an accurate performance prediction.

Kernel name	**type**	**prediction**	**measure**	**error**
Full Lagrange-Remap	MB	10697 cy/CL	10092 cy/CL	6%
Prediction Lagrange-Flux	CB	5712 cy/CL	5548 cy/CL	2.8%
Correction Lagrange-Flux	CB	5712 cy/CL	5690 cy/CL	0.3%

Table 3: Speed up on single core Intel SandyBridge with AVX for both Lagrange-Flux kernels. Even if the kernels are compute bound, speedup is not perfect, because the bottleneck is the DIV port throughput.

Kernel name	**type**	**predicted**	**measured**
Prediction Lagrange-Flux	CB	2.0	1.9
Correction Lagrange-Flux	CB	2.0	1.9

neck for both kernels, making them compute-bound. Note that in this case, the Roofline and ECM models are equivalent, and give the same prediction. With AVX SIMD vectorization, the bottleneck remains the divide port. The consequence is that the ECM model predict a speedup of 2 with respect to the baseline scalar version: on the one hand, elements are processed 4 by 4 thanks to AVX, but on the other hand, the divide port throughput is only doubled. Since the Lagrange-Flux is compute-bound, our performance model also predicts perfect scalability (e.g. x16 speedup) for multithreaded execution. It is close to the observed speedup (x13.3).

Finally, in order to compare absolute performance and scalability of Lagrange-Flux and Lagrange-Remap solvers, we report in 4.3 a measured runtime comparison for the same test case. The results confirm what our performance model is able to predict: the Lagrange-Flux solver has superior absolute performance. It is interesting to note that this is achieved by trading single core unvectorized performance (which is worse than the corresponding Lagrange-Remap solver) for much better scalability.

5 Conclusions

In this paper, we have proposed a practical and quantitative way of modifying a fluid mechanics solver, the Lagrange-Remap algorithm, for better performance on multicore CPUs. The resulting algorithm, the Lagrange-Flux solver, has comparable numerical quality, but better runtime, and is more compact (two kernels instead of a dozen). The main tools used in our

Scheme	**1 core**	**1 core AVX**	**16 cores AVX**	**scalability**
Lagrange-Flux	1.9	3.9	52.0	27.1
Lagrange-Remap	2.4	3.7	36.5	14.6

Table 4: Performance comparison between the reference Lagrange-Remap solver and the Lagrange-Flux solver in millions of cell updates per second (MCUPs) on SandyBridge. Different machine configurations are presented. Scalability (last column) is computed as the speed up of the multi-threaded vectorized version compared to the baseline purely sequential version. Tests are performed for fine meshes, such that kernel data lies in DRAM memory. The Lagrange-Flux solver exhibits superior scalability, because it has — by design — better arithmetic intensity.

methodology are analytical performance models, allowing us to quantitatively assess and qualitatively precisely understand algorithmic performance. They explain the main reason behind the superior performance of our new solver: it is a compute-bound algorithm, whereas the original is memory-bound.

We believe that our methodology is quite general, and can be extended to other computational physics algorithms, for fluid mechanics but also for other domains of application. An other interesting perspective is to extend the use of performance models to many-core architectures such as graphical processing units (GPUs), or the Intel Many Integrated Core (MIC).

References

[1] David J Benson. "Computational methods in Lagrangian and Eulerian hydrocodes". *Computer methods in Applied mechanics and Engineering* 99.2 (1992), pp. 235–394.

[2] Raphael Poncet, Mathieu Peybernes, Thibault Gasc, and Florian De Vuyst. *Performance modeling of a compressible hydrodynamics solver on multicore CPUs*. accepted at ParCo2015 conference, Edinburgh, Scotland, UK 1-4 sept. 2015. 2015.

[3] Jan Treibig and Georg Hager. "Introducing a Performance Model for Bandwidth-Limited Loop Kernels". *Parallel Processing and Applied Mathematics*. Ed. by Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski. Vol. 6067. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 615–624.

[4] Florian De Vuyst, Thibault Gasc, Renaud Motte, Mathieu Peybernes, and Raphael Poncet. *Lagrange-flux schemes: reformulating second-order accurate Lagrange-remap schemes for better node-based HPC performance*. Proccedings of the SimRace2015 conference in OGST. 2016.

[5] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76.

[6] JD McCalpin. "STREAM: Sustainable memory bandwidth in high performance computers". *Technical Report, University of Virginia* (continuously updated).

[7] *Intel architecture code analyzer*. Available: http://software.intel.com/en-us/articles/intel-architecture-code-analyzer/.

[8] Georg Hager, Jan Treibig, Johannes Habich, and Gerhard Wellein. "Exploring performance and power properties of modern multi-core chips via simple machine models". *Concurrency and Computation: Practice and Experience* 28.2 (2016), pp. 189–210.

[9] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. "Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model". *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS '15. Newport Beach, California, USA: ACM, 2015, pp. 207–216.

[10] Mark L Wilkins. "Use of artificial viscosity in multidimensional fluid dynamic calculations". *Journal of computational physics* 36.3 (1980), pp. 281–303.

[11] Paul Woodward and Phillip Colella. "The numerical simulation of two-dimensional fluid flow with strong shocks". *Journal of computational physics* 54.1 (1984), pp. 115–173.

[12] Eleuterio F Toro. *Riemann solvers and numerical methods for fluid dynamics: a practical introduction. Third edition*. Springer-Verlag, 2009.