GPU and APU computations of Finite Time Lyapunov Exponent fields

Christian Conti, Diego Rossinelli *, Petros Koumoutsakos

Computational Science, ETH Zurich, Zurich CH-8092, Switzerland

Abstract

We present GPU and APU accelerated computations of Finite-Time Lyapunov Exponent (FTLE) fields. The calculation of FTLEs is a computationally intensive process, as in order to obtain the sharp ridges associated with the Lagrangian Coherent Structures an extensive resampling of the flow field is required. The computational performance of this resampling is limited by the memory bandwidth of the underlying computer architecture. The present technique harnesses data-parallel execution of many-core architectures and relies on fast and accurate evaluations of moment conserving functions for the mesh to particle interpolations. We demonstrate how the computation of FTLEs can be efficiently performed on a GPU and on an APU through OpenCL and we report over one order of magnitude improvements over multi-threaded executions in FTLE computations of bluff body flows.

1. Introduction

The Finite-Time Lyapunov Exponent (FTLE) field characterizes the rate of separation of neighboring trajectories over a finite time interval (see [29] and works therein). FTLEs are a diagnostic tool which often indicates the presence of Lagrangian Coherent Structures (LCS) and they have received significant attention in several areas of fluid dynamics. FTLE fields have been used to reveal flow structures in the context of locomotion of jellyfish and flying animals [42,30], fluid transport in stirred fluids and laminar vortex rings [37,38] as well as in the context of geophysical flows [12] and magnetic fields [31].

The FTLE field is usually computed by seeding a velocity field with massless particles, integrating their positions forward or backward in time and computing the FTLE values using the maximum eigenvalues of the Cauchy–Green deformation tensors of the flow map. The computation of FTLE fields is computationally intensive as it requires a large number of integration steps for particle trajectories in order to compute a flow map. A number of algorithmic improvements have been proposed to accelerate the computation of FTLEs: in [10] the flow map is approximated with bidirectional and unidirectional intermediate flow maps while Adaptive Mesh Refinement techniques have been used for accelerating the flow map computation [14] and the identification of the ridges [36].

We propose the use of many-core computer architectures to accelerate the computation of FTLEs, as in recent years it has been shown that GPUs provide substantial performance improvements in the field of computational fluid dynamics [26]. Examples include flow simulations on GPUs using finite difference/volume schemes [13,20,39,8], Lattice-Boltzmann [5,25,40] and remeshed vortex methods [33].

In this work we perform also simulations on an Accelerated Processing Unit (APU) [2], a new many-core architecture that has a SIMD engine (also called Fusion GPU) and a multicore CPU physically integrated on the same die. The big advantage of this configuration is the loss of the PCIe bus for communication between CPUs and GPUs that often represent the bottleneck for GPU computing. A second fundamental difference is that the CPUs and the GPUs share the DRAM memory, although the
memory space is divided in a region dedicated to the CPU managed by the OS and one for the SIMD engine managed by software running on the SIMD engine [9]. These new devices are designed for a tight collaboration between CPUs and GPU components thus enabling true heterogeneous computing. This model simplifies the work of the programmer who is not forced anymore to move as much data as possible to the GPU to minimize communication through the PCIe bus.

In this work we investigate how the computation of FTLE fields for bluff body flows can be accelerated using GPUs and APUs along with the method of intermediate flow maps proposed by Brunton and Rowley [10]. The implementation of FTLEs for GPUs has been explored also by Garth et al. [15] without however going into the details of the numerics and the optimization, since their work focused on issues related to visualization.

The FTLE computations involve a large amount of integration steps and an extensive sampling of the flow field at the locations of the trajectories. This mesh–particle interpolation is performed using moment-conserving kernels [23]. The efficiency of such mesh–particle samplings hinge on the memory bandwidth of the underlying computer architecture. Many-core architectures are well-suited to accelerate this process due to their high bandwidth [34]. We consider different techniques to effectively map the necessary computational steps to these devices and we quantify the attainable performance. To effectively coordinate the CPU–GPU/APU memory traffic we rely on OpenCL, which provides a suitable programming model as well as efficient execution and memory models.

We present quantitative performance results for the GPU Fermi Tesla C2070 and the APU A8-3850, a mid-end platform providing a performance comparable to high-end GPUs. With respect to the direct FTLE computation on the CPU, the developed techniques lead the time-to-solution to an acceleration of two orders of magnitude: one order of magnitude is provided by harnessing the data parallelism of many-core architectures, a further order of magnitude comes from the algorithmic improvement provided by the indirect FTLE computation with the intermediate flow maps.

The paper is structured as follows: in Section 2 we describe how the FTLE fields can be computed, we introduce the equations used for the simulation of bluff body flows and present the moment-conserving mesh–particle interpolations. In Section 3 we briefly introduce OpenCL for many-core architectures along with techniques for computing FTLE fields in the context of steady and unsteady flows. Among the techniques for unsteady flows, we present our many-core implementation of the method proposed in [10] that is based on the computation of intermediate flow map to reduce the amount of computation required in the trajectory integration. In this section we also discuss two techniques for performing mesh–particle interpolation efficiently on many-core architectures. In Section 4 we report the achieved performance improvements for steady and unsteady flows and we discuss the FTLE fields obtained with the present technique for the analysis of fish motions.

2. Method

2.1. Finite-Time Lyapunov Exponent (FTLE)

The FTLE $\sigma^T_{t_0}(x)$ [24,21] is a scalar field which characterizes the amount of stretching experienced by a volume element associated with the trajectory of a massless particle in a flow field, from the location $x_0$ to the location $x$ over the time interval $[t_0, T = t_0 + T_{\epsilon}]$. Considering a flow described by the velocity field $u(x,t)$ in a domain $\Omega$, the trajectories $X$ are given by the solution of the following equation:

$$\frac{dX(t; t_0, x_0)}{dt} = u[X(t; t_0, x_0), t],$$

$$X(t_0; t_0, x_0) = x_0, \quad x_0 \in \Omega.$$

The flow map $\phi^t_{t_0}$ advects particles to their position at time $t$ for a fixed initial time $t_0$:

$$\phi^t_{t_0}(x_0) = x_0 + \int_{t_0}^{t} u[X(t; t_0, x_0), \tau]d\tau.$$

The derivative of the flow map $\phi^t_{t_0}$,

$$\frac{d\phi^t_{t_0}}{dx}(x_0) = I + \int_{t_0}^{t} \frac{d\phi^\tau_{t_0}}{dx}(x_0) \cdot \nabla u[X(t; t_0, x_0), \tau]d\tau,$$

is used to compute the Cauchy–Green deformation tensor $\Delta(x_0)$ at a given point $x_0$, which quantifies the stretching of an infinitesimal material line over the interval $[t_0, t_0 + T_{\epsilon}]$:

$$\Delta(x_0) = \left[ \frac{d}{dx} \phi^{t_0 + T_{\epsilon}}_{t_0}(x_0) \right]^T \left[ \frac{d}{dx} \phi^{t_0 + T_{\epsilon}}_{t_0}(x_0) \right].$$

The eigenvalues of this matrix are associated with the Finite-Time Lyapunov Exponents and the largest eigenvalue $\lambda_{\text{max}}$ (corresponding to the maximum FTLE), represents the largest infinitesimal stretching rate occurring along the trajectory initialized at $x_0$. 

The FTLE field at the location $x_0$ is then defined as:

$$\sigma_{t_0}^{t_0+TLE}(x_0) = \frac{1}{|TLE|} \ln \sqrt{\lambda_{\text{max}}(\Delta(x_0))}. \quad (6)$$

FTLEs measure the maximum local rate of separation of the particles in a system and can be used to find locally attracting and repelling material surfaces in time-dependent systems, the strongest of which are called LCS. However, as shown by Haller [18], LCSs and FTLE ridges do not always identify each other. As FTLE describes the stretching of trajectories, FTLE fields will show ridges of high values over separatrices that represent material lines that have no flux across them.

The forward-time FTLE ($TLE > 0$) often reveals repelling LCS, which are stable manifolds in a steady flow. The backward-time FTLE ($TLE < 0$) on the other hand, often reveals attracting LCS, that are unstable manifolds in a steady flow [17,42].

Let us consider the following velocity field:

$$u = -\frac{\partial \psi}{\partial y}; \quad \psi = \frac{\partial w}{\partial x} \quad \text{(7)}$$

$$v = \frac{\partial \psi}{\partial x} \quad \text{(8)}$$

in the region $[0,1]^2$ where $\psi = \sin(\pi(x+0.5))\sin(\pi(y+0.5))$. The velocity field is shown in Fig. 1 with the trajectories of four particles seeded on the middle-left side of the domain.

Two trajectories which start close to each other but on opposite sides of the stable manifold will diverge as $t \rightarrow +\infty$. Similarly, trajectories initially on opposite sides of an unstable manifold diverge as $t \rightarrow -\infty$. Fig. 2 shows how these manifolds are revealed as ridges of the FTLE field. Computing the forward-time FTLE reveals a ridge of high FTLE values over the unstable manifold which repels the particles. The backward-time FTLE reveals a ridge of high FTLE values over the stable manifold which attracts the particles.

2.2. Remeshed vortex methods for simulations of bluff body flows

The two-dimensional bluff body flow fields for which we compute the FTLE fields are computed by solving the velocity–vorticity formulation of an incompressible viscous flow with a remeshed vortex method [23]. The no-slip boundary condition is enforced by the Brinkman penalization technique [3,11]:

$$\frac{\partial \omega}{\partial t} + (\mathbf{u} + \mathbf{U}_\infty) \cdot \nabla \omega = \nu \Delta \omega + \lambda \nabla \times \left( \mathbf{U}_\infty - \left( \mathbf{u} + \mathbf{U}_\infty \right) \right) \chi_s, \quad \text{(9)}$$

where $\mathbf{u}(x,t)$ denotes the velocity field, $\nu$ the kinematic viscosity and $\omega = \nabla \times \mathbf{u}$ denotes the vorticity. The characteristic function $\chi_s$ localizes the obstacle by taking the value of 1 inside the obstacle and 0 outside, and the factor $\lambda \gg 1$ is inversely proportional to the porosity of the obstacle.
We approximate the vorticity field with a set of particles which are located at positions $x_p$ and carry vorticity $\omega_p$:

$$\omega_p^h(x) = \sum_p \omega_p W(x - x_p^h),$$  \hspace{1cm} (10)

where $h$ denotes the inter-particle distance (in this case equivalent to the spacing of the mesh used to seed the particles) and $W$ is the reconstruction kernel. In this work we consider $W$ to be the two-dimensional tensor product of the $M_0^4$ kernel [23] given by:

$$M_0^4(x) = \begin{cases} 
0 & \text{if } |x| > 2, \\
\frac{1}{2} (2 - |x|)^2 (1 - |x|) & \text{if } 1 \leq |x| \leq 2, \\
1 - \frac{3}{2} x^2 + \frac{1}{2} |x|^3 & \text{if } 1 \geq |x|.
\end{cases}$$  \hspace{1cm} (11)

The discretization of the velocity–vorticity Navier–Stokes equations yields a set of ordinary differential equations (ODEs) for the particle positions and their vorticity:

$$\frac{dx_p}{dt} = u_p,$$  \hspace{1cm} (12)

$$\frac{D \omega_p}{Dt} = \nu \Delta \omega_p + \lambda \nabla \times [U - (u_p + U_\infty)]/\chi_S.$$  \hspace{1cm} (13)

The velocity field at the particle locations can be computed by solving the Poisson equation $\nabla^2 u = -\nabla \cdot \omega$. In hybrid particle methods [22] a background mesh is used in order to remesh the particle locations at each time step. The far field boundary condition (i.e. $\lim_{x \to \infty} u = 0$) is imposed by extending the computational domain as described in [19]. To compute the particle velocities, we compute the vorticity on the mesh by sampling the vorticity field as described in Eq. (10), and by finding the velocity field at the mesh points with a fast Poisson solver. To sample the velocity at the particle locations, we perform a filtering of the mesh points with the reconstruction kernel. This mesh–particle interpolation, later referred to as "M2P", is defined as follows:

$$\omega_p = \sum_i \omega_{i,\text{mesh}} \cdot W\left(\frac{1}{h} (x_p - x_{i,\text{mesh}})\right).$$  \hspace{1cm} (14)

We note that the interpolation in two dimensions requires four points per direction, thus involving sixteen mesh points, as we use a tensorial product of the $M_0^4$ kernel.

The choice of the time step is constrained by the LCFL condition, $\delta t \leq C_1 \| \nabla \otimes u \|^{-1} \approx C_2 \| \omega \|^{-1}$, where $C_1$ and $C_2$ are constants depending on the time-integration scheme.

To generate the velocity fields used in the computation of the FTLE fields presented in this paper, the LCFL number was set to 0.1.

2.3. Computation of FTLEs

The generation of FTLE fields requires computing flow quantities at arbitrary particle locations, which are obtained by performing mesh–particle interpolation. At the beginning, massless particles are regularly distributed in the domain $\Omega$. Then, the particle positions are integrated in time by a second-order explicit Runge–Kutta scheme, which discretizes Eq. (1). We integrate the position of the particles $x_p^n$ from step $n$ to step $n + 1$ by sampling the velocity field:
\[ x_p = x_p + \delta t u(x_p^n, t^n), \]
\[ x_p^{n+1} = x_p^n + \delta t \frac{u(x_p^n, t^n + \delta t) + u(x_p^n, t^n)}{2}. \]

Because \( u^n \) is represented on a mesh, we perform an M2P operation to evaluate \( u^n(x_{p,\text{mesh}}) \):
\[ u_p^n = \sum_i u^n_{i,\text{mesh}} W\left(\frac{1}{h}(x_p - x_{i,\text{mesh}})\right). \]

As for the bluff-body flow, the choice of time step is constrained by the LCFL condition. The number of steps (i.e. the selection of \( T_\varepsilon \)) is application-dependent and here is found empirically. For the flows considered in this work we performed 10–10^4 steps with an LCFL number of 0.5. In order to compute the flow maps, in this work we use first-order no flux boundary conditions. We note that these special boundary conditions work for bluff body flows, provided that the domain is sufficiently large. However, these boundary conditions may not work for some other types of flows, such as channel flows.

The gradients of the Cauchy–Green deformation tensor are computed with the central difference scheme:
\[ \Delta(x_p) = \left( \frac{\Delta x_{i,j} - \Delta x_{i,j-1}}{2h} \frac{\Delta x_{i,j} - \Delta x_{i,j-1}}{2h} \right), \]

where \( p = (i,j) \) is a two-dimensional index denoting the index of the particle initially displaced at location \((i \cdot h, j \cdot h)\). In terms of memory usage, we note that the FTLE computation of an unsteady flow is algorithmically more expensive than the one for steady velocity fields, as we need to process an array of time-snapshots of the velocity field to correctly integrate the trajectories.

3. Implementation

3.1. OpenCL for many-core architectures

The present implementation of the FTLE field solver relies on OpenCL [27], a standard for multi/many-core computing, providing portability between different hardware, an essential feature for this work as we present implementations and comparisons on different platforms.

The OpenCL execution model is composed of two parts: the kernels that execute on one or more OpenCL devices (here the many-core devices), and a host program that executes on the OpenCL host (the CPU). When a kernel is submitted for execution by the host, a corresponding index space is defined. When the execution of the kernel starts, an instance of the kernel, called work-item, is created for each element in the previously defined index space. These instances are executed in parallel and can process different sets of data.

In order to pass data structures to a kernel, OpenCL provides two different types of memory objects: buffers and images. Buffers are plain one-dimensional arrays whose elements can be addressed with a one-dimensional index. Image objects store 2D/3D arrays in a structure optimized for data reads; the image elements (pixels) can be accessed in a kernel only with built-in functions of OpenCL. GPUs and APUs have hardware support for reading images and therefore these operations are fast. Image elements consist of a 4-component vector (RGBA), which means that 4 single precision components can be retrieved with a single read instruction. For many-core devices each core processes work-items in a single-instruction multiple-data (SIMD) fashion. As the number of resources available per-core is limited, the work-items have to be fine-grained: many threads performing small processing on a small amount of data. Because of their fine-grained properties, the life time of a work-item is expected to be in the order of microseconds or less. At the same time the amount of work-items is expected to be big, at least one order of magnitude larger than the amount of cores. From these considerations, we deduce that optimal performance is achieved when work-items are homogeneous, i.e. when the small tasks perform the same operations.

We note that the performance gain achieved by GPUs and APUs is very fragile: performance degradation may be observed due to low kernel occupancy (high per-work-item register usage), non-coalesced memory transactions, diverging control flow instructions and memory indirections.

3.2. FTLE on many-core architectures for steady flows

The host representation of the velocity field consists of two scalar meshes, \( u \) and \( v \), stored as two-dimensional contiguous arrays. In this work the overall memory size of \( u \) and \( v \) fits within one device. Fig. 3 shows the data flow of the many-core-FTLE computation. The arrows encode the dependencies between the blocks, the green\(^1\) boxes represent kernels or collections of kernel, which are executed in parallel for all the particles.

Initially, the flow quantities \( u \) and \( v \) reside in the system memory of the host (red box) and are transferred to the device as image objects. The first processing step is executed by a collection of kernels called “Integration”. These kernels initialize the

\(^1\) For interpretation of color in Figs. 2, 3, 9–11, the reader is referred to the web version of this article.
particles and compute particle trajectories following the scheme described in Eq. (15). For steady flows this is done in a single pass, since the velocity field does not change in time. The particles trajectories are advected with the flow until $T_{LE}$ is reached. In order to update their positions, these kernel perform intensive resampling of the velocity field by performing M2P operations. Because the particles freely move in the domain, the sampling of the velocity field takes place at arbitrary locations. This prevents the kernel to perform coalesced reads, thus one cannot afford to use buffer objects to represent $u$ and $v$. Instead, the representation of the velocity field as image objects is better suited, as it is often the case that nearby particles show spatial locality in the resampling process.

The output of the kernel “Integration” consists of two buffer objects storing the final position of the particles. These buffers are converted into image objects and are sent as input of the “FTLE” kernel where we approximate the Cauchy–Green deformation tensor, and compute the FTLE values. We perform buffer-image conversion to avoid non-coalesced access of the particle positions when we evaluate the scheme of Eq. (18). The output of the kernel is a buffer object representing the FTLE field, which is eventually sent back to the host.

### 3.3. FTLE on many-core architectures for unsteady flows

Fig. 4 shows the data flow of the FTLE computation on many-core for unsteady flows. The first kernel involved in the processing is called “Initialize”, and it is responsible for initializing the particle positions. Its output is made of two buffer objects representing the $x$- and the $y$-component of the particle positions. These buffers are sent to the collection of kernels called “Integration”. Differently from steady flows case, the kernel is executed in multiple passes, one per integration step. To update the particle positions at time $t_{n+1}$, we need to transfer velocity field $u(t_{n+1})$ to the device before the kernel execution. The result is a pair of buffer objects representing the updated locations of the particles, which are sent back as input in the next call of the same kernel. After having performed the required number of integrations steps, the two buffers containing the final locations are converted to images and sent to the FTLE kernel, already described in Section 3.2.

We emphasize that the dependency graph of all the operations involved in this FTLE computation is not trivial. An efficient implementation that follows this dependency graph with the explicit use of synchronization point is challenging.
especially if one takes into account the multi-pass execution involved in the processing. With OpenCL however, we do not have to handle these synchronization points explicitly, as we can describe the complete sequence of actions by means of OpenCL events. The commands to transfer flow quantities, process them and transfer back the FTLE fields can be submitted to the command queues with an associated event without blocking.

3.4. FTLE on many-cores with intermediate flow maps for unsteady flows

In order to accelerate the “integration” phase, which is the most expensive one, we introduce a hierarchy of intermediate flow maps. The method was proposed by Brunton and Rowley [10] and uses intermediate flow maps to decrease the time spent in the computation of a series of FTLE fields.

This method builds a hierarchy of intermediate flow maps starting from the velocity fields. Fig. 5 shows the diagrams for the construction of the flow map hierarchy: the first level of this hierarchy is built by doing a single time integration step on each velocity field and storing the resulting flow maps (Fig. 5(i)). Successive levels can be built by composing consecutive flow maps (Fig. 5(ii)). It is however not necessary to build the full hierarchy as the higher the map is in the hierarchy the lower is the number of times it can be used.

Integrating the trajectories to compute an FTLE field then means choosing the smallest set of flow maps that covers exactly the interval considered and composing them to get the complete approximate flow map. The process for computing the FTLE field from the flow map hierarchy is shown in Fig. 6.

Each composition of flow maps can be done with the M2 kernel: a flow map representing the initial state and the second one being the quantities that have to be interpolated on the first. To reduce the memory footprint on the many-core devices and to be able to compute the FTLE fields as the data is being produced, we update the set of intermediate flow maps with each FTLE field we compute. With this additional phase, we can outline our implementation in four steps. In the first one we remove the flow maps that will not be used anymore, in order to free resources. In the second step we use the new velocity fields to compute the “children” flow maps that are inserted in the hierarchy. In the third step, bottom-up, we create the “parent” flow maps of higher level in the hierarchy by composing the children flow maps. In the final step we conclude the process of the FTLE computation by evaluating the Cauchy–Green tensor.

The management of the hierarchy, i.e. the decision of which flow map to remove, to add and to compose, is delegated to the host employing a specific logic that keeps track of the statuses on the fly. The information provided is then used by OpenCL to remove/add memory objects accordingly. In our program, we keep the entire hierarchy of intermediate flow maps on the GPU to avoid communication with the CPU therefore trading memory for performance.

3.5. The numerical integration of the flow map

In the numerical integration of the flow map (the “Integration” box) a considerable amount of time is spent performing mesh–particle interpolations. In this work we discuss two implementation techniques (Technique 1 and Technique 2) for the numerical integration of the flow map, which rely on efficient M2P interpolations. Technique 1 is the OpenCL extension of the techniques used in [35,33] (written in OpenGL). Technique 2 is a newer implementation that has been presented in [34] and in this work we demonstrate that it performs better for FTLE fields computation. For both techniques, each work-item within a kernel is associated to one particle and the register usage per kernel is kept as low as possible by splitting the algorithm over several kernels, which are optimized to further reduce the number of registers required and achieve high enough occupancies (in our experience an occupancy larger than 0.1 is sufficient for this objective). For the mesh–particle interpolation kernels, the velocities at the mesh points are stored in image objects. In this way we achieve optimal “read” memory bandwidth and avoid coalescing problems due to the non-trivial access pattern.

Technique 1. Fig. 7 shows the data flow and the data dependencies of Technique 1. This technique works with buffers where vector components, \((x, y)\) and \((u, v)\), are interleaved in memory. As depicted in the diagram, this technique rearranges

![Fig. 5. Steps in the construction of the flow maps hierarchy. (i) shows the creation of the first level whereas (ii) shows how successive levels are built.](image-url)
the velocities and stores them as image objects where each pixel contains the velocity components of one mesh point (out of 4 channels per pixel). The velocities are sampled at the particle locations with the M2P kernel, which produces a single buffer of velocities whose directional component are interleaved. The M2P output is processed by the “Euler” kernel, which produces a buffer object containing the updated particle positions. This buffer is then used as input to a second instance of the M2P kernel, which produces, according to Eq. (16), the components of \( \frac{\partial u}{\partial x} \), \( \frac{\partial u}{\partial t} + \frac{x}{t^2} \), \( \frac{\partial v}{\partial y} \). The final result is obtained by concatenating six instances of the three kernels.

This technique benefits from a reduced number of synchronization points (with respect to Technique 2) and the computation of the filter weights in the mesh–particle interpolation (Eq. (17)) is efficient, since it is done once per particle. Nonetheless Technique 1 has three performance drawbacks. Firstly, we introduce an additional kernel to rearrange mesh points into a single image, which is time consuming as it is forced to perform non-coalesced memory transactions. Secondly, only half of the image memory is used as we have a one-to-one mapping between mesh points (containing \( u, v \)) and pixels (containing four channels). A third problem is that we consider the per-particle work to be the finest granularity of the parallelism in this kernel, although the mesh–particle interpolation is performed on two independent quantities \( u \) and \( v \).

**Technique 2.** As the details of Technique 2 are already discussed in [34], we will only shortly outline the method. Fig. 8 gives an overview of the data flow. The quantities \( u \) and \( v \) are represented with two image objects. Within a pixel, all the image channels are used. The resulting image length and width are the system size in \( x \)-direction divided by four and the system size in \( y \)-direction, respectively. The mesh–particle interpolation is performed independently on \( u \) and \( v \), by executing the M2P kernel twice per particle. The Euler kernel is also performed separately for each component, and produces a buffer object containing the updated component of the associated particle.
Overall, the integration step is composed of eight different kernel instances and we map work-items to process the single components of the particles. This leads to a double amount of work-items, which increases the potential degree of parallelism and better expresses the dependency between the different kernels. This implementation uses the memory more efficiently and the absence of kernels for re-arranging the mesh points before the interpolation. The main drawback is that the mapping to find the correct velocity values from the images is non-trivial, as it is designed to minimize the number of times that the velocity image is accessed. This implementation additionally leads to a larger number of dependencies between the various kernels.

Transfer-computation overlap. To further improve Technique 2 on GPU (that performs better than Technique 1, as will be shown in Section 4) we introduce the overlapping of computation with data transfer from CPU to GPU [28], i.e. we reorganize the tasks of Technique 2 so that we have one CPU–GPU transfer for each block of two M2P and two Euler kernel executions (we only need to transfer two velocity fields per integration).

Fig. 9 shows how the overlapping is accomplished for Technique 2: the velocity field \( u_{t+2} \) is transferred while the GPU is processing the second phase of RK2 with \( u_{t+1} \), then \( v_{t+2} \) is copied to the GPU while the first part of the time integration is being computed. To reach this objective, we are required to offset the transfers: in the first semi-iteration, we transfer \( u_{t+1}, v_{t+1} \) and \( v_{t+2} \) and we compute the first part of RK2, i.e. two M2P and two Euler operations. In the second semi-iteration we only compute the second part of RK2 without transferring any velocity field.

From an implementation point of view, this overlap is obtained by using two OpenCL queues and the `clFlush` instruction that flushes a sequence of commands to the queues.

The maximum overlap that we can expect with our algorithm is given by the minimum time between the total transfer time per iteration and the total computation time per iteration.

We note that this technique is not required on the APU as it is able to do zero-copy that allows the GPU component to directly access memory in the CPU address space [1].

4. Results

In the first part of this section we report and discuss the performance observed for FTLE computations of bluff body flows, considering different system sizes and integration steps. In the second part we present a series of FTLE fields obtained with our implementation and we discuss the main features of the corresponding flow.
The performance results are obtained on a 2P NUMA system composed of AMD “Istanbul” Opteron 2435 with six cores at 2.66 GHz.

We also consider an Nvidia Tesla C2070 GPU that shows a peak GPU bandwidth of 78 GB/s out of the theoretical 148 GB/s and a peak single precision floating point performance of 1015 GFLOP/s out of the theoretical 1030 GFLOP/s.

Additionally, we consider an AMD APU (Accelerated Processing Unit), a new class of processors that puts a multicore CPU and a SIMD engine on the same chip sharing DRAM memory. Specifically, we use an A8-3850 “Lynx” with a quad-core at 2.9 GHz and a Radeon HD 6550D GPU. The peak performance measured for the SIMD engine is 452 GFLOP/s out of the theoretical 480 GFLOP/s. As the GPU is using the DRAM memory, its bandwidth is dependent on the type of memory used. In our case, we measure a bandwidth of 18 GB/s of the theoretical 21 GB/s. A main advantage of the APU is that host-device data transfers do not involve the PCIe bus, which usually degrades the performance of heterogeneous platforms because of its low memory bandwidth. The tradeoff however is the loss of the GPU’s internal bandwidth. This means that by design an APU is better suited for tasks that require an large amount of communication between CPU and GPU. Another consequence of this model is that the programmer is not forced to move the entire computation on GPU, even the parts of the code that are not suited for the device, to minimize communication through PCIe. APUs allow to execute each task of the program where it runs best: serial code and regions with lower parallelism on CPU and highly parallel code on the SIMD engine.

All the kernels that run on Nvidia GPUs have been written and executed with the OpenCL 1.0 implementation of Nvidia whereas on the AMD APU we used the OpenCL 1.1 implementation of AMD. The reference CPU implementation of the FTLE computation was written in C++ with OpenMP and with SSE intrinsics, and compiled with the Intel C/C++ compiler with the flags icc -O3 -openmp -xHOST -ipo -ip.

4.1. Performance of the integration step: Technique 1 and Technique 2

The “Rearrange”, “Euler” and “M2P” kernels of Technique 1 use 18, 7 and 27 GPU registers and their occupancy is of 0.75, 1 and 0.5, respectively. For Technique 2, the “Euler” and “M2P” kernels use 5 and 30 registers and have occupancies of 1 and 0.5, respectively. The occupancy is high enough for all the kernels to hide arithmetic latencies for devices of any compute capability as well as to hide register dependencies on the C2070.

Table 1 shows a comparison of the execution times for both techniques over 1000 iterations of the trajectory integration step and with different problem sizes, and the average GFLOP/s of the M2P kernels for one Tesla C2070 GPU. The timings reported in the Table also consider the time required to transfer the velocity fields to the GPU.

We observe that the performance difference between Technique 1 and Technique 2 varies depending on the system size. For the smallest size, the execution time of Technique 1 is 2.5% more than Technique 2. For larger system sizes Technique 2 is 20% faster than Technique 1.

The performance of the mesh–particle interpolation (M2P) of Technique 2 is constantly better than the one of Technique 1. The latter reaches a peak of 96 GFLOP/s, whereas Technique 2 reaches a peak of almost 130 GFLOP/s. We estimate that the operational intensity [41] of the particle–mesh interpolation is about 1.9 FLOP/bytes (see [34] for a detailed explanation). Because we observe a (peak) GPU internal bandwidth of 78 GFLOP/s, we estimate that the highest computing performance of mesh–particle interpolation is 148 GFLOP/s. Technique 2 therefore achieves 86% of the maximum achievable performance on GPU.

Tables 2 and 3 show the time distribution of Technique 1 and Technique 2 for different system sizes for the trajectory integration step of the FTLE computation. We note that the major bottleneck of both techniques is the data transfer from CPU to GPU, which takes 23–48% of the execution for Technique 1 and 33–54% for Technique 2. Because of this and the overlapping of transfers and computations offered by OpenCL when using a single GPU, we expect that using more than one GPU could actually slow down the FTLE computation as the data transfers to the various GPUs get sequentialized.

We also note that the most expensive computational step is the mesh–particle interpolation of the velocities (35–55%) and we observe that its ratio decreases with the system size as the communication time increases. As previously mentioned,
The highest operational intensive kernel is the M2P with 1.9 FLOPs/Byte. The “Euler” kernel has an operational intensity of approximately 0.17 FLOP/Byte (two floating point operations for each three read/write operations), whereas the “Rearrange” kernel involves only read/shuffling/write operations, thus all the computational steps have a performance that is bounded by the memory bandwidth.

The partitioning of time for Technique 2 on the APU is shown in Table 4 (the higher system sizes are not shown as they are too large to be handled by the APU). As in the case of the GPU, the execution time is dominated by memory transfers (even though the APU has a higher host-device memory bandwidth) and M2P is the computational bottleneck. We note that the memory transfers on the APU are different from the GPU as it is a simple movement of data in the DRAM from one address space to another. Moreover the APU is able to do zero-copy, meaning that, in theory, this data transfer is not needed as the GPU is able to read the data directly from the CPU address space. However the feature is currently only available on Windows operating systems and we are therefore not able to use it [1]. If we consider the time differences between the two system sizes (shown in Table 5) we observe that the transfer time increases by a factor 1.46, Euler by 3.73 and M2P by 2.1 as the system increases in size by a factor four. The improvement of the M2P is also reflected in the change in the GFLOP/s performance of the operator: 67 GFLOP/s for the smallest system size and 131 GFLOP/s for the 1024 × 512.

However, considering the bandwidth of the APU and the operational intensity of the M2P operator, we would expect a performance of 34 GFLOP/s. The unexpectedly positive result could be explained by the way the data of an image object is cached on the APU, increasing the operational intensity of the M2P.

4.2. Overlapping of memory transfers and computation

Table 6 shows a time comparison between the Technique 2 with and without overlap on the Tesla C2070. As can be seen on Table 3, for the Tesla C2070, the maximum overlap is going to be determined by CPU–GPU communication for small system size (33–36%) and by computation for larger ones (46–49%).

Table 6 shows that the actual benefit we measure is smaller (around half of what we estimate possible): 13–18% for the small sizes and 23–29% for large systems.

As the APU is not able and does not need overlapping, we will not provide results for this implementation.

4.3. Speedups for FTLE fields of steady flows

The left graph of Fig. 10 shows the overall performance improvement over a single CPU thread for the GPU (green) and for a 12-threaded execution (blue) for 100 iterations. The speedup for the CPU code displays a 8–10X improvement with respect to the single threaded run with some performance degradation for larger system sizes. The GPU improvement, on the other hand grows with the system size from 1.4X to 4.6X over the multithreaded execution.
On the right plot in Fig. 10 we can observe the speedups measured on the APU over 100 iterations. As for the Opteron CPU, we do not get perfect scaling on CPU: the speedup for four threads is stable around 3.2X. The speedup improves with the system size, however, for the largest problem we observe a performance drop. Overall, the SIMD engine shows an improvement over a host-only multi-threaded execution that varies between 1.8X and 3X.

4.4. Speedups for FTLE fields of unsteady flows

The left picture of Fig. 11 shows the observed speedup of the FTLE computation of an unsteady field over the single-threaded CPU execution, obtained with 12 CPU threads and on a single Tesla C2070 GPU with and without overlap. For unsteady velocity fields, we achieved a maximum speedup of 9.6X whereas on the Tesla C2070 we obtain improvements of 13.7X and 28.6X, without and with overlap, respectively.

The right picture of Fig. 11 shows the observed speedup for the APU. For this hardware configuration we observe a maximum speedup of 3X for the four threaded C++ run and of 7.6X for the SIMD engine.

A comparison of the plots in Fig. 11 with the corresponding results shown in Fig. 10 clearly show the influence of the CPU–GPU communication: for the GPU computation with a Tesla C2070 the performance for unsteady flows drops dramatically in contrast to the CPU performance that remains unvaried. The same behavior is observed on the APU but, due to the higher bandwidth for moving data between CPU and GPU the impact is smaller in the middle system sizes.

4.5. FTLE field computation with intermediate flow maps

Tables 7 and 8 show the timing comparison of the method with intermediate flow maps with the direct implementation of the FTLE field computation for unsteady flows for a sequence of 32 FTLE fields from a total of 128 velocity fields (thus a single FTLE field is computed from 96 velocity fields) on the Tesla C2070 GPU. The expected speedup $S$ over the direct method is estimated by the ratio of the integration steps in the two implementations:

$$S = \frac{\text{Integration steps in direct method}}{\text{Integration steps in intermediate flow maps method}}$$
\[ S = \frac{(N_{\text{vel}} - N_{\text{FTLE}})N_{\text{FTLE}}}{2N_{\text{vel}}}, \]  

where \( N_{\text{vel}} \) is the total number of velocity fields we use and \( N_{\text{FTLE}} \) is the number of FTLE fields we compute, assuming that the velocity fields are regularly distributed in time.

Table 7 shows the variation of the time to solution with respect to the size of the problem whereas Table 8 shows the improvement brought by the use of intermediate maps with respect to the number of FTLE fields to compute.

We observe that the speedups measured reflect the speedup predicted and we note that the benefit provided by the use of intermediate flow maps is dependent on the number of flow maps to be computed and the number of velocity fields at our disposal. This method provides an algorithmic improvement of one of magnitude over the standard method.

Table 9 shows the performance comparisons for the APU: we are not showing the larger system sizes as they are too large for the APU’s memory to handle. We note that in terms of device memory requirements, the standard implementation is substantially cheaper as it only needs to keep four velocity fields on the device at any time whereas for the computation with intermediate flow maps, the number of flow maps to be stored on the device varies and depends on the number of integration steps required, unless the CPU–GPU communication is greatly increased. Moreover, for the cases where the APU has enough resources for the computation, we see that the improvement that the intermediate flow maps bring is smaller than expected, although still considerable.

### 4.6. C-start of a larval fish

We employ the FTLE methodologies presented in this paper in order to analyze the flow field of a larval fish rapidly accelerating from rest (C-start). Fig. 12 shows six forward FTLE fields produced by a simulated larval fish while starting to move from a relaxed position (top left picture). FTLE fields can be used to understand how the fish is propelling itself: from the initial position (top left) the fish bends and creates a pocket of trapped fluid (top right). The fish then accelerates this pocket of fluid opposite to its swimming direction, thus gaining momentum (middle left). The cycle then starts again with a larger pocket (middle right) that the fish can use to move forward while returning to a straight position (bottom left). Finally the fish bends again to the left for the second part of the cycle, this time producing a larger region of trapped fluid.

The velocity fields used for the FTLE computation are the results of the optimization to the movement of a larval fish from still position to forward swimming so that it maximizes the traveling velocity, as further elaborated in [16]. The data set used is originally composed of approximately 7000 2048 \( \times \) 2048 velocity field which, for the purpose of computing the FTLE fields have been temporally coarsened to 700 fields. To compute the FTLE fields shown in Fig. 12, we considered a time window involving 230 velocity fields.

### Table 7

<table>
<thead>
<tr>
<th>System size</th>
<th>Direct (s)</th>
<th>Indirect (s)</th>
<th>( S_{\text{measured}} )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 ( \times ) 256</td>
<td>3.52</td>
<td>0.26</td>
<td>13.31</td>
<td>12</td>
</tr>
<tr>
<td>1024 ( \times ) 512</td>
<td>10.12</td>
<td>0.74</td>
<td>13.59</td>
<td>12</td>
</tr>
<tr>
<td>2048 ( \times ) 1024</td>
<td>39.93</td>
<td>3.25</td>
<td>11.04</td>
<td>12</td>
</tr>
<tr>
<td>4096 ( \times ) 2048</td>
<td>135.20</td>
<td>11.82</td>
<td>11.43</td>
<td>12</td>
</tr>
</tbody>
</table>
5. Conclusions

We present techniques to efficiently compute the FTLE fields on many-core architectures with OpenCL. These techniques rely on a data-parallel mesh–particle interpolation scheme and its inherent fine-grained concurrency, used to resample the flow velocity field. The implementation of the mesh–particle interpolation employs efficiently image objects, which are hardware-supported. We also presented a many-core implementation of the algorithm proposed by Brunton and Rowley [10] for the computation of FTLE fields that relies on the efficient implementation of the M2P operator shown in [34]. We observed that the algorithmic improvements provided by this method are significant and depend on the number of FTLE fields to compute as well as the amount of involved velocity fields.

The current implementation achieves a maximum computing performance of almost 130 GFLOP/s on a Tesla C2070. Surprisingly, the APU A8-3850 was able to obtain comparable performance, even though both the bandwidth and the peak floating point performance is lower than a Tesla C2070. For unsteady velocity fields, results indicate an overall improvement of up to 30X with respect to the single-threaded CPU execution and 4.5X over a 12-threaded CPU computation. Due to the high cost of transferring data between CPU and GPU, including more than one GPU would not benefit the computation, as the

Table 8
Timing results on the C2070 for the implementations with (indirect) and without the use of intermediate flow maps (direct) for varying number of integration steps for a system size of 2048 × 1024 and 128 velocity fields.

<table>
<thead>
<tr>
<th>N_{FTLE}</th>
<th>Direct (s)</th>
<th>Indirect (s)</th>
<th>S_{measured}</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7.22</td>
<td>3.04</td>
<td>2.38</td>
<td>1.94</td>
</tr>
<tr>
<td>8</td>
<td>11.36</td>
<td>3.08</td>
<td>3.69</td>
<td>3.75</td>
</tr>
<tr>
<td>16</td>
<td>20.33</td>
<td>2.89</td>
<td>7.02</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>33.60</td>
<td>3.03</td>
<td>11.08</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 9
Timing results for the implementations with (indirect) and without the use of intermediate flow maps (direct) for varying system size. The number of FTLE fields computed is kept constant at 32 and the number of velocity fields is 128.

<table>
<thead>
<tr>
<th>System size</th>
<th>Direct (s)</th>
<th>Indirect (s)</th>
<th>S_{measured}</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 × 256</td>
<td>22.87</td>
<td>3.67</td>
<td>6.23</td>
<td>12</td>
</tr>
<tr>
<td>1024 × 512</td>
<td>38.14</td>
<td>4.86</td>
<td>7.85</td>
<td>12</td>
</tr>
</tbody>
</table>

Fig. 12. Sequence of six FTLE fields for a fish rapidly accelerating from rest. The FTLE values range shown range from 2 (white) to 9 (black). Values below 2 (the minimum is −0.83) are saturated to 2 as are values above 9 (the maximum FTLE is 11.66) so that the pictures are easier to interpret.
CPU–GPU data transfers to multiple GPUs are sequentialized. On the APU, the computation of FTLE fields of unsteady flows with the SIMD engine delivers a performance of up to 3X with respect to the multi-threaded version.

Additionally, we evaluated the use of APUs, a new class of processors that puts CPUs and GPUs together on the same die, in the context of FTLE field computation of unsteady flows. The first generation APUs used in this work shows promising results, especially considering the energy requirements and economical convenience of such devices. We showed that this architecture is able to provide performance comparable to latest generation, high-end GPUs. Although our APU configuration was not able to handle all the problem sizes we considered in this work, mainly due to the memory limitations typical of mid-end desktop computers, we could measure a speedup over the multicores implementation of 2.5X. Ongoing work is concerned with the zero-copy capabilities of APUs that can provide an even larger advantage for this architecture when a tight collaboration between the multicore component and the SIMD engine is required. Future works include the extension of this technique to three-dimensional flows, and the use of wavelet-based adaptive particles [4,32] to reduce the algorithmic cost of computing FTLE fields.

References


