Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013 (SNA + MC 2013)
La Cité des Sciences et de l'Industrie, Paris, France, October 27–31, 2013

Vectorization of a 2D–1D Iterative Algorithm for the 3D Neutron Transport Problem in Prismatic Geometries

Salli Moustafa[1], François Févotte[1], Bruno Lathuilière[1], and Laurent Plagne[1]

[1]*EDF R&D, SINETICS department, 1 av. du Général de Gaulle, 92140 Clamart, France*

The past few years have been marked by a noticeable increase in the interest in 3D whole-core heterogeneous deterministic neutron transport solvers for reference calculations. Due to the extremely large problem sizes tackled by such solvers, they need to use adapted numerical methods and need to be efficiently implemented to take advantage of the full computing power of modern systems.
As for numerical methods, one possible approach consists in iterating over resolutions of 2D and 1D MOC problems by taking advantage of prismatic geometries. The MICADO solver, developed at EDF R&D, is a parallel implementation of such a method in distributed and shared memory systems. However it is currently unable to use SIMD vectorization to leverage the full computing power of modern CPUs.
In this paper, we describe our first effort to support vectorization in MICADO, typically targeting Intel© SSE CPUs. Both the 2D and 1D algorithms are vectorized, allowing for high expected speedups for the whole spatial solver. We present benchmark computations, which show nearly optimal speedups for our vectorized implementation on the TAKEDA case.

KEYWORDS: Method of Characteristics (MOC), MICADO, prismatic geometry, SIMD, Intel SSE

I. Introduction

The past few years have been marked by a noticeable increase in the interest in 3D whole-core heterogeneous deterministic neutron transport solvers for reference calculations.(1) This trend has been sustained by the ongoing increase in available computing power, both for personal desktop systems and high performance computing clusters. Due to the extremely large problem sizes[1] tackled by such solvers, they need to use adequate numerical methods and need to be efficiently implemented to take advantage of the full computing power of modern systems.

As far as the numerical methods as concerned, one possible approach – suitable for prismatic reactors – is based on a 2D Method of Characteristics (MOC) to handle unstructured meshes in the radial dimensions. Following CRX,(2) different variants of this approach have been implemented in a wide spectrum of solvers such as CHAPLET-3D(3) or DeCART.(4) In a previous paper, we describe in details the 2D–1D coupling methodology implemented at EDF R&D in the MICADO solver,(5) which is part of the larger COCAGNE platform.

As far as high performance computing is concerned, modern systems routinely present three levels of parallelism: distributed memory parallelism, where independent processing units communicate using a network; shared memory parallelism, where different cores share a fast access to the same Random Access Memory (RAM); and vectorization, where SIMD[2] instructions allow a single processing unit to simultaneously apply the same operation on multiple values. While MICADO takes advantage of the properties of the 2D–1D method to implement distributed and shared memory parallelism,(5) it is currently unable to use SIMD vectorization to leverage the full computing power of modern CPUs. This state of affairs is all the more sorry that such instruction sets have been widely supported by most microprocessors for a long time (SSE[3] has been introduced by Intel in 1999), and is likely to play a more and more important role in the future,(6) with the arrival of new SIMD instruction sets featuring more parallel channels: AVX[4] or AVX-512, which will allow applying the same operation to as many as 16 single precision floating points numbers simultaneously. As stated by NVidia: "all contemporary processors (CPUs and GPUs) are built by aggregating vector processing units".(7)

In this paper, we describe our first effort to support vectorization in MICADO. The targeted architecture is Intel SSE, but we will focus on algorithms and techniques which scale to larger SIMD technologies. In part II, we present the distributed and shared-memory parallel implementation of the 2D–1D method in MICADO. We then move on in part III to the techniques used to vectorize it, before assessing the efficiency of the vectorization in part IV.

[1]the number of degrees of freedom in discretized 3D heterogeneous multigroup S_N problems can often be of the order of 10^{12}.

[2]SIMD: Single Instruction, Multiple Data.

[3]Intel© SSE: Streaming SIMD Extension, allows to apply the same operation to 4 single precision floating point numbers simultaneously.

[4]AVX: Advanced Vector eXtensions, first supported by Intel with the Sandy Bridge processor in 2011, which allows to apply the same operation to 8 single precision floating point numbers simultaneously.

II. Presentation of the MICADO Solver

1. Method

1.1. 2D–1D equations

After discretization of the energetic and angular variables in the neutron transport problem, the monokinetic spatial equation left to solve can be written as:

$$\forall\, \mathbf{\Omega} \in S_N,$$
$$\varepsilon \frac{\partial \psi}{\partial x}(x,y,z) + \eta \frac{\partial \psi}{\partial y}(x,y,z) + \mu \frac{\partial \psi}{\partial z}(x,y,z) + \Sigma \psi(x,y,z) = Q(x,y,z), \quad (1)$$

where S_N represents the chosen angular quadrature formula (i.e. a set of directions discretizing the unit sphere S^2), and ε, η, μ are the three components of direction $\mathbf{\Omega}$ respectively along axes x, y and z. The angular flux at a given point is denoted by $\psi(x,y,z)$, Σ is the total macroscopic cross-section, and $Q(x,y,z)$ represents the source term including fission and scattering.

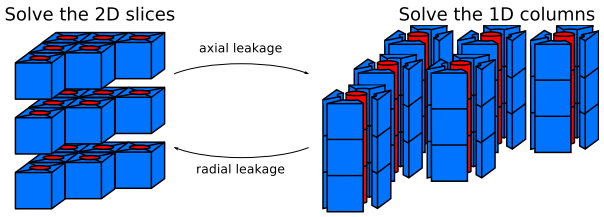

Figure 1: Alternating resolution on 2D slices and 1D columns

In prismatic geometries, space can be discretized as the Cartesian product of a radial mesh and an axial one. Let us denote by $\mathcal{R}_i, i \in [\![1, N_i]\!]$ the regions of the potentially unstructured radial mesh, and $z_j, j \in [\![1, N_j + 1]\!]$ the boundaries of the axial mesh. As detailed in a previous paper,[5] equation (1) can be numerically approximated by an iterative system coupling the solutions of a set of 2D equations on "slices" of the geometry, with 1D problems on columns (Figure 1), using leakage terms.

Algorithm 1: 2D–1D coupling

```
▷ Convergence loop
while ψ_R ≠ ψ_Z do
    ▷ Solve 2D MOC equations
    forall j ∈ [[1, N_j]] :
        forall Ω_k ∈ S_N :
            ▷ Solve equation (2)
        end
    end
    ▷ Solve 1D MOC equations
    forall i ∈ [[1, N_i]] :
        forall Ω_k ∈ S_N :
            ▷ Solve equation (3)
        end
    end
end
```

This leads to algorithm 1, where for any angular direction $\mathbf{\Omega}_k$, the set of 2D equations for every slice $[z_j, z_{j+1}]$

$$\int_{z_j}^{z_{j+1}} \left(\varepsilon_k \frac{\partial \psi_{k,R}}{\partial x} + \eta_k \frac{\partial \psi_{k,R}}{\partial y} + \Sigma \psi_{k,R} \right) dz = \int_{z_j}^{z_{j+1}} \left(Q - \mu_k \frac{\partial \psi_{k,Z}}{\partial z} \right) dz \quad (2)$$

is coupled to the set of 1D equations for every column extruded above region $\mathcal{R}_i$

$$\int_{\mathcal{R}_i} \left(\mu \frac{\partial \psi_{k,Z}}{\partial z} + \Sigma \psi_{k,Z} \right) dx\, dy = \int_{\mathcal{R}_i} \left(Q - \varepsilon_k \frac{\partial \psi_{k,R}}{\partial x} - \eta_k \frac{\partial \psi_{k,R}}{\partial y} \right) dx\, dy \quad (3)$$

1.2. Discretization using the MOC

In practice, equations (2)–(3) are discretized using the Method of Characteristics[8] (MOC). We will not enter the details of the method here, but just describe general principles. For the sake of simplicity, let us rewrite equation (2) in a more classical form, without leakage source terms coming from the 2D–1D iterations:

$$\mathbf{\Omega} \cdot \nabla_{x,y} \psi(\Omega, x, y) + \Sigma(x,y)\, \psi(\mathbf{\Omega}, x, y) = Q(x,y). \quad (4)$$

In the following, we will describe the resolution process of the 2D equation in a given axial slice $[z_j, z_{j+1}]$. The method of characteristics uses the discretization of the 2D radial domain in regions $\mathcal{R}_i, i \in [\![1, N_i]\!]$. This mesh is potentially unstructured, the regions being of arbitrary shapes; the only requirement is that they should cover the entire domain without overlapping. Although higher order schemes exist,[9] we suppose in our implementation that the source term Q and total cross-section Σ are constant region-wise.

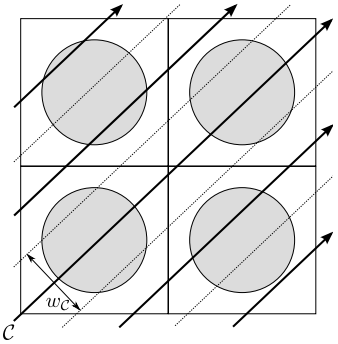

Figure 2: Tracking of a 2D unstructured geometry

The method of characteristics relies on a *tracking* to capture details of the geometry: for each direction in the quadrature formula, a set of parallel lines is defined, covering the whole geometry (fig. 2). Each of these lines is tracked, starting from the domain boundary, and the sequence of crossed regions is stored along with associated intersection lengths.

For each characteristic line crossing a region in the geometry, integrating equation (4) along the segment yields transmission and balance equations of the following form:

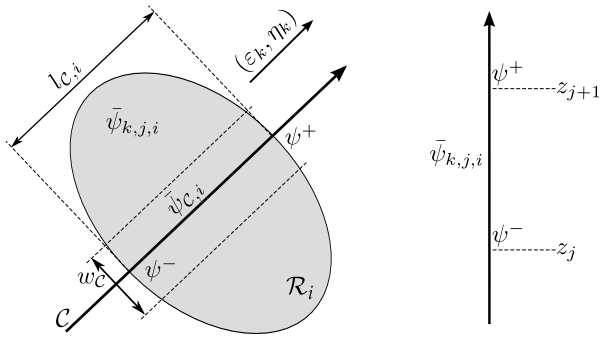

Figure 3: Flux transmission in direction $\mathbf{\Omega}_k = (\varepsilon_k, \eta_k, \mu_k)$:
(left) – 2D characteristic crossing radial region $\mathcal{R}_i$ in slice $[z_j, z_{j+1}]$
(right) – 1D transmission through slice $[z_j, z_{j+1}]$ in column $\mathcal{R}_i$ ($\mu_k > 0$)

$$\psi^+ = T(\psi^-, l_{C,i}, Q, \Sigma), \tag{5}$$

$$\begin{aligned}\bar{\psi}_{C,i} &= \int_{C \cap \mathcal{R}_i} \psi(\mathbf{\Omega}_k, x, y)\, dr \\ &= B(\psi^-, \psi^+, l_{C,i}, Q, \Sigma),\end{aligned} \tag{6}$$

with the notations of figure 3: ψ^- and ψ^+ denote the flux respectively entering and exiting region $\mathcal{R}_i$ through characteristic line C. $l_{C,i}$ represents the intersection length between C and $\mathcal{R}_i$, and $\bar{\psi}_{C,i}$ is the average angular flux in region $\mathcal{R}_i$ along C. The real expressions used for transmission T and balance B operators will be discussed in the next section.

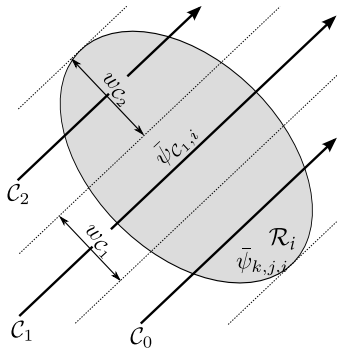

Figure 4: Integration of the average flux in a region

A transverse integration formula is then used to compute region-averaged angular flux $\bar{\psi}_{k,j,i}$:

$$\begin{aligned}\bar{\psi}_{k,j,i} &= \int_{\mathcal{R}_i \times [z_j; z_{j+1}]} \psi(\mathbf{\Omega}_k, x, y)\, dx\, dy\, dz, \\ &= \sum_n w_{C_n}\, \bar{\psi}_{C_n,i},\end{aligned} \tag{7}$$

where notations are explained in figure 4: w_{C_n} is a transverse weight associated to characteristic line C_n in the tracking.

1.3. Integration schemes

Two different integration schemes are implemented in MICADO, leading to different expressions for transmission (T) and balance (B) functions:

Step Characteristics (SC) : this scheme is the most usually implemented in the method of characteristics, and can be obtained by integrating the transport equation along a characteristic line crossing a region:

$$\begin{aligned}T(\psi^-, l, Q, \Sigma) &= \psi_i^-\, e^{-\Sigma l} + \frac{1 - e^{-\Sigma l}}{\Sigma} Q, \\ B(\psi^-, \psi^+, l, Q, \Sigma) &= \frac{\psi^- - \psi^+}{\Sigma} + \frac{Q\, l}{\Sigma}.\end{aligned}$$

Diamond Differencing (DD) : this scheme can be derived[(10)] by supposing the averaged flux to be the half sum of entering and exiting fluxes; the obtained *ansatz* for exiting flux ψ^+ is injected into the transport equation to yield the transmission equation:

$$\begin{aligned}T(\psi^-, l, Q, \Sigma) &= \frac{2 - \Sigma\, l}{2 + \Sigma\, l}\, \psi^- + \frac{2\, l}{2 + \Sigma\, l}\, Q, \\ B(\psi^-, \psi^+, l, Q, \Sigma) &= \frac{l}{2}\left(\psi^- + \psi^+\right),\end{aligned}$$

It is interesting to note at this stage that the Step Characteristics scheme presents a higher arithmetic intensity than the Diamond Differencing: although both schemes require the same information being fetched from memory[5] (l, Σ, Q), the DD scheme requires only a few operations to process them, whereas the SC scheme requires computing an expensive exponential (even though tabulating the exponential function somewhat reduces its computational cost,[(11)] the SC scheme still remains more arithmetically intensive than the DD scheme).

1.4. Sweeping algorithms

The equations detailed above lead to the resolution of the set of 2D equations (2) using a sweep described in algorithm 2.

For the sake of simplicity, we will suppose here the boundary conditions to be null incoming flux (thus modelling void). Supporting other boundary condition types leads to more complicated algorithms,[(12)] but does not affect the matters discussed in this paper. For the sake of readability, the last two arguments Q and Σ to transmission and balance operators T and B have been omitted in the algorithm; they are respectively taken to be the source term and total cross section in the current region.

In this algorithm, as well as those presented in the rest of the paper, the **forall** keyword is used for a loop whose order is unimportant (and all iterations can potentially be realized in parallel, except for some write operations on shared variables, which should be considered as reductions). On the other hand, the **foreach** keyword has been used for loops in which the order of iterations is important. These loops are much less easily parallelized.

The incoming flux in the first region crossed by a characteristic line is initialized using the boundary conditions. The transmission equation (5) is then repeatedly applied to successive regions crossed by the characteristic line, allowing computation of an exiting flux, which will be taken as the incoming flux in next region. For each region, the segment averaged flux is computed using the balance equation (6), and accumulated

[5] temporary variables like ψ^- and ψ^+ stay in registers.

Algorithm 2: 2D resolution

1 ▷ *Loop over axial slices*
2 **forall** *slices* $j \in [\![1; N_j]\!]$ **:**
3 ▷ *Loop over directions*
4 **forall** *directions* $\mathbf{\Omega}_k \in S_N$ **:**
5 ▷ *2D sweep in direction* $\mathbf{\Omega}_k$
6 **forall** *characteristic lines* $C \in \mathcal{T}_k$ **:**
7 $\psi^- = 0$; ▷ *Initialize incoming flux*
8 **foreach** *region* r_i *crossed by* C **:**
9 ▷ *Apply transmission and balance eqs:*
10 $\psi^+ = T(\psi^-, \frac{l_{C,i}}{\mu_k}, ...)$;
11 $\bar{\psi}_{k,j,i} = \bar{\psi}_{k,j,i} + w_C \; B(\psi^{\pm}, \frac{l_{C,i}}{\mu_k}, ...)$;
12 ▷ *Forward outgoing flux to next region:*
13 $\psi^- = \psi^+$;
14 **end**
15 **end**
16 **end**
17 **end**

into the region-averaged flux using the transverse integration formula (7). Repeating this sweep for all characteristic lines, all angular directions and all slices yields the solution of the set of 2D equations in the whole domain.

Algorithm 3: 1D resolution

1 ▷ *Loop over directions*
2 **forall** *directions* $\mathbf{\Omega}_k \in S_N$ **:**
3 ▷ *Loop over radial regions*
4 **forall** *regions* $i \in [\![1; N_i]\!]$ **:**
5 $\psi^- = 0$; ▷ *Initialize incoming flux*
6 ▷ *Loop over axial slices, in ascending or*
7 *descending order depending on the sign of* μ_k
8 **foreach** *slice* $j \in [\![1; N_j]\!]$ **:**
9 ▷ *Apply transmission and balance equations:*
10 $\psi^+ = T(\psi^-, \frac{z_{j+1}-z_j}{\sqrt{1-\mu_k^2}}, ...)$;
11 $\bar{\psi}_{k,j,i} = \bar{\psi}_{k,j,i} + B(\psi^{\pm}, \frac{z_{j+1}-z_j}{\sqrt{1-\mu_k^2}}, ...)$;
12 ▷ *Forward outgoing flux to next region:*
13 $\psi^- = \psi^+$;
14 **end**
15 **end**
16 **end**

The set of 1D equations (3) is solved using a similar process, except that there is no such thing as a tracking in a 1D geometry: the only characteristic line is the z axis, which crosses all regions in ascending or descending order depending on the sign of μ (figure 3). Associated intersection lines are simply the slices thickness. This makes the 1D sweep described in algorithm 3 much simpler.

2. Parallel Implementation

As explained in a previous paper,[5] MICADO implements parallel versions of algorithms 2 and 3, using the following strategy:

- Distributed memory parallelization using MPI: axial slices are distributed among processors, which leads to:
  - parallelizing the loop on slices at line 2 in algorithm 2 ;
  - pipelining the loop on columns at line 4 in algorithm 3.
- Shared memory parallelization using Intel© Threading Building Blocks (TBB):
  - the 2D solver (algorithm 2) uses parallel loops for slices (line 2) and angular directions (line 4);
  - the 1D solver (algorithm 3) uses parallel loops for the radial regions (line 4).

Therefore, although the loop on slices would be very suitable to vectorization in the 2D algorithm, we decided against vectorizing it, choosing instead to preserve the ability to distribute one slice per parallel node.

III. Vectorization of MICADO

1. Intel SSE

Targeted architectures for the vectorization of MICADO are not GPUs, but rather standard CPUs used in EDF R&D cluster Ivanoe[13] featuring the Intel© SSE (Streaming SIMD Extensions) vectorized instruction set. Using this technology, a single operation can be applied simultaneously to 4 single precision (or 2 double precision) floating point numbers stored in appropriate registers.

Although SSE is our current target architecture, it is important to keep in mind that new SIMD instruction sets are coming, such as AVX[14] or AVX-512 which will allow processing as many as 8 or 16 single precision floating point numbers at once. It is therefore of capital importance that the vectorization techniques employed be scalable to more than 4 channels. On the other hand, our vectorized algorithm does not need to scale to thousands of simultaneous SIMD operations. Solvers which aim at supporting such massively vectorized operations, for example targeting GPU-based systems, often transform the energetic multigroup Gauss-Seidel iterations into a Jacobi iterative solver; such non-optimal changes in algorithm should be avoided in our case.

A very important thing to keep in mind about vectorization is that all data meant to be processed vectorially should be stored contiguously in memory. Failing to comply with this requirement results in extremely low performances.

2. Quadrature formulas

As one can see in algorithm 2, apart from the source term Q and cross-section Σ fields, the transmission and balance equations only use geometric information such as list of regions r_i crossed by characteristics, and associated intersection length $l_{C,i}$. As one can see on figure 5, these only depend on the azimuthal direction given by (ε, η). These information are stored in the tracking, which is computed at the beginning of the computation. On the other hand, the polar direction (given by μ) only appears as a scaling on intersection lengths.

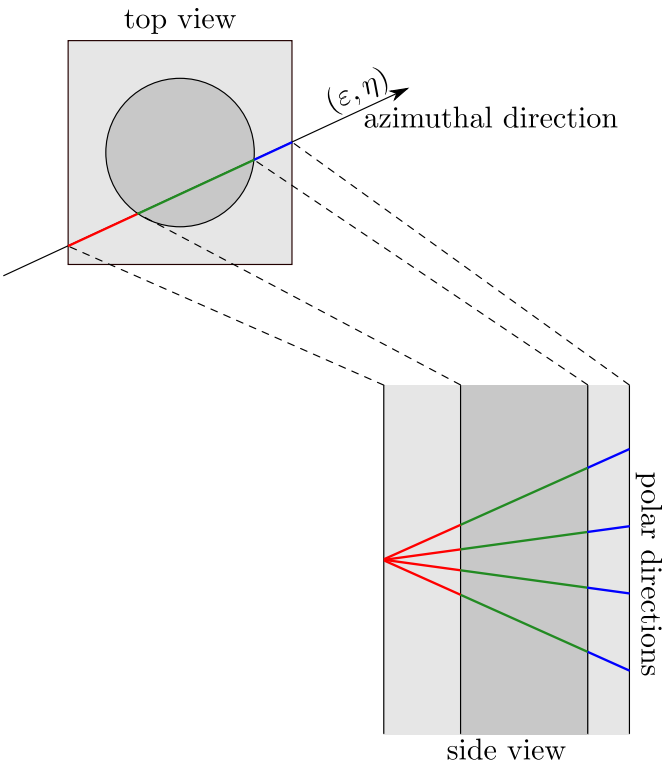

Figure 5: Sharing of azimuthal tracking in product quadrature formulas

This leads to an interesting optimization if the angular quadrature formula is constructed as the Cartesian product of an azimuthal quadrature and a polar one :

$$\left\{ (\varepsilon_{k_a}, \eta_{k_a}), k_a \in [\![1, N_a]\!] \right\} \times \left\{ \mu_{k_p}, k_p \in [\![1, N_p]\!] \right\}.$$

Indeed, in such cases, tracking information can be computed only for azimuthal directions, and shared between all polar directions.

This leads to modified algorithm 4, where the loop over angular directions is split into two: an outer azimuthal loop, and an inner polar loop. A major difference in this algorithm is that the temporary variables $\psi^{\pm}$ – storing incoming and outgoing fluxes along a characteristic line – are now vectors indexed by the polar direction.

Although using only product quadrature formulas can be a constraint (especially for comparisons with solvers using general quadratures such as Level Symmetric), it is believed to be an acceptable restriction in the author's opinion. Indeed, most solvers based on the Method of Characteristics, whether in 2D or 3D, favor such quadrature formulas[12,15] to reduce the tracking storage requirements.

Algorithm 4: 2D resolution – product quadrature formula

```
1  ▷ Loop over axial slices
2  forall slices j ∈ [[1; N_j]] :
3      ▷ Loop over azimuthal directions
4      forall directions (ε_{k_a}, η_{k_a}), k_a ∈ [[1, N_a]] :
5          ▷ 2D sweep
6          forall characteristic lines C ∈ T_{k_a} :
7              foreach regions r_i crossed by C :
8                  ▷ Loop over polar directions
9                  forall polar angles μ_{k_p}, k_p ∈ [[1, N_p]] :
10                     ▷ Apply transmission and balance eqs:
11                     ψ⁺_{k_p} = T(ψ⁻_{k_p}, l_{C,i}/μ_{k_p}, ...);
12                     ψ̄_{k,j,i} = ψ̄_{k,j,i} + w_C B(ψ^±_{k_p}, l_{C,i}/μ_{k_p}, ...);
13                     ▷ Update incoming flux in next region:
14                     ψ⁻ = ψ⁺ ;
15                 end
16             end
17         end
18     end
19 end
```

3. Vectorization of the 2D solver

An interesting property of algorithm 4 is that the inner polar loop (line 9) can be vectorized, since the exact same operations are applied in the same order to a set of N_p values.

Such a vectorization does however require rather drastic changes in the storage policy used for spatial fields. The natural storage policy for algorithm 2 consists in storing and accessing fields as multidimensional arrays indexed in the following order:

```
psi[j][k][i],
```

where the notations are consistent with the rest of the paper: `i` indexes radial regions, `j` indexes axial slices, and `k` indexes angular directions. Vectorizing algorithm 4 however requires that values relative to consecutive polar angles be stored contiguously in memory, thus imposing the following type of addressing:

```
psi[j][ka][i][kp],
```

where index `k` on directions has been split into `ka` and `kp`, respectively indexing azimuthal and polar directions.

4. Vectorization of the 1D solver

Such a storage policy being chosen, one would favor vectorizing the 1D solver along polar angles, which is the only way to avoid shuffling data in memory. This is however not feasible: as shown in figure 6, 1D sweeps must be done according to the sign of μ. For a product quadrature formula with 4 polar angles, two of these angles lead to data flowing upwards, while the two others lead to data flowing downwards during the sweep.

Unless the quadrature formula counts enough polar angles, it is therefore not scalable to sweep them vectorially.

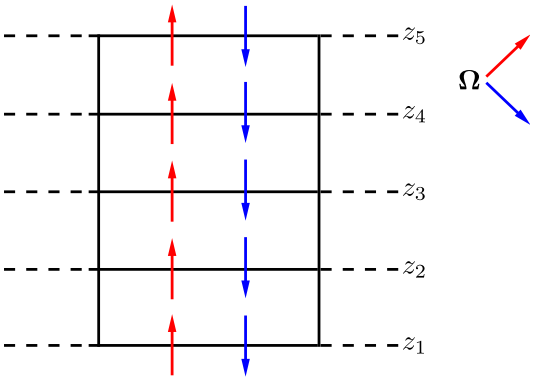

Figure 6: Flow of data in a 1D sweep, for directions going upward ($\mu > 0$, in red) or downward ($\mu < 0$, in blue)

Studying algorithm 3, another candidate for vectorization is the loop over radial regions (line 4). Indeed, for any given direction in the product quadrature formula, the sweep of any 1D column leads to the same slices to be visited in the same order. This technique should moreover be scalable since, even in distributed memory environments where this loop is pipelined, the pipeline block size can be chosen large enough for the vectorization to take place.

Vectorizing this loop does however entail a data relocation, since data referring to neighbouring regions should now be stored consecutively in memory. This causes stored fields to be stored and accessed in the following fashion:

`psi[j][ka][kp][i]`,

with the indexes described above. This data relocation is only done blockwise in the pipeline; the entire vector does not need to be duplicated in memory.

IV. Numerical Results

1. Benchmark

The results presented hereafter were obtained on the TAKEDA benchmark,[16] using the following parameters:

- uniform $50 \times 50 \times 50$ Cartesian spatial discretization;
- Gauss-Legendre 8×4 product angular quadrature formula;
- single precision floating point numbers;
- all parallelization options turned off.

For 2D results, only the central plane of the TAKEDA geometry is considered, discretized in a 500×500 Cartesian spatial mesh.

Different versions of the MICADO solver are compared, which we will refer to using the following names in slanted font to avoid ambiguity in the results:

general: algorithm 2, using a general quadrature formula, without vectorization; this version will be used as a reference.

product: algorithm 4, adapted to product quadrature formulas, without vectorization. When used for 3D calculations, the 1D axial solver also uses a modified algorithm where the loop over directions has been split into nested azimuthal and polar loops.

sse: same algorithms as the *product* version, with a vectorized implementation using Intel© SSE to implement the strategies described in sections III.3 and III.4.

It is important at this point to stress that all versions of the solver produce the exact same results. These results were proven to be accurate with respect to the benchmarked MCNP reference in previous work.[5]

In all tables presented below, computing times will be presented in seconds elapsed. Measurements were performed on an Intel© Xeon© E5620 CPU (Westmere© microarchitecture). This CPU features 128-bit SIMD registers allowing to apply simultaneously the same operation on 4 single precision floating point numbers. With our settings, the maximum obtainable speedup between the *product* and *sse* versions is thus 4.

Different parts of the computation were timed. Timings for solver iterations were obtained by averaging a few iterations for better accuracy.

tracking: computation and storage of all characteristic lines. This is a preprocessing step, needed only once at the beginning of the computation and re-used for all iterations. Tracking computing times are therefore not really relevant for realistic computations where several inner iterations will be performed; they are shown here only for reference purposes.

2D solver: solution of the set of 2D fixed-source neutron transport problems for all axial slices, and computation of axial leakage source terms. This is the most expensive part in the computation, and the one we aim at reducing in priority.

1D solver: solution of the set of 1D fixed-source neutron transport problems for all columns, and computation of radial leakage source terms. Although this part is less computationally intensive than the 2D solver, it can become limiting in distributed memory environments. In the vectorized version (*sse*), 1D Solver computing times include in-memory data relocations needed to adapt data storage to SSE constraints.

2D–1D solver: one iteration of the mono-kinetic, fixed-source spatial solver (algorithm 1). The time spent in this part is essentially (98–99%) composed of the sum of the 2D and 1D solvers.

2. Results for the 2D case

Results for the 2D case are presented in tables 1 and 2, respectively for the Step Characteristics Scheme and the Diamond Differencing scheme.

As expected, using an algorithm adapted to product quadrature formulas greatly reduces the tracking time, though not as

	Tracking (speedup)	2D Solver (speedup)
general	5.18	5.50
product	1.96 (×2.64)	6.29 (×0.88)
sse	1.57 (×3.30)	1.48 (×3.73)
product/fixed	1.96	5.82 (×0.94)
product/fast-exp	1.96	4.75 (×1.15)

Table 1: Computation times (s.) for different code versions, using the Step Characteristics scheme in 2D.

much as one might have anticipated: although the workload has decreased by a factor 4 (the number of polar angles), speedups are not higher than 2.64. This might be due to fixed overheads such as memory allocations and file inputs to read the data set. Slightly better speedups (3.3) can be obtained by vectorizing the computation of numerical volumes at the end of the tracking.

As for the 2D solver computing time, we can observe interesting results. When using the Step Characteristics scheme, using a product quadrature formula slightly decreases performances by approximately 12%. This is partly incurred by the added inner loop on polar angles, as we can see by introducing a new version of the code, named *product/fixed*. This implementation is identical to the *product* version, but the loop on polar angles has been fixed and unrolled at compile time. Part of the performance is regained; the remaining performance drop can probably be explained by the change in data storage policy.

Vectorizing the inner loop however proves to be effective since we obtain speedups of 3.73, close to the maximal value 4. Speedup between the *product* and *sse* version is even larger than 4, which might be due to the use of a fast vectorized exponential implementation coming from the Eigen[17] library in the *sse* version, whereas the standard `glibc` version was used in the *product* implementation. This effect can be measured by introducing another test version, called *product/fast-exp*. In this version, the implementation is identical to *product*, except that exponentials in the Step Characteristics scheme are computed using the Eigen library. The speedup between *sse* and *product/fast-exp* is 3.21, which is back under the maximal SSE speedup value of 4.

	Tracking (speedup)	2D Solver (speedup)
general	5.02	2.19
product	1.84 (×2.73)	1.93 (×1.14)
sse	1.61 (×3.12)	1.44 (×1.52)

Table 2: Computation times (s.) for different code versions, using the Diamond Differencing scheme in 2D.

Due to the low arithmetic intensity of the Diamond Differencing scheme, it benefits from better data locality in the *product* implementation, compensating for the cost of the extra inner polar loop and even allowing for a 1.14 speedup. However, the lower arithmetic intensity also makes vectorization less effective than in the SC scheme, leading to only 1.5 speedup.

3. Results for the 3D case

Computing times for 3D calculations are reported in tables 3 and 4 respectively for the Step Characteristics and Diamond Differences schemes.

	2D (speedup)	1D (speedup)	2D–1D (speedup)
general	2.69	0.21	2.93
product	3.04 (×0.89)	0.23 (×0.91)	3.30 (×0.89)
sse	0.52 (×5.21)	0.10 (×2.18)	0.65 (×4.51)

Table 3: Computation times (s.) for different code versions, using the Step Characteristics scheme in 3D.

We can first observe that in the general implementation, the 2D solver uses more than 90% of the computing time of a spatial solution, which explains our vectorization choices in favor of the good efficiency of 2D algorithms.

Vectorization efficiency of the 2D solver in a 3D environment is not different than what was observed for 2D cases: the added inner loop on polar angles incurs a slight loss of performances (around 10%), which is largely compensated for by the obtained vectorization gain (5.21 speedup, once again larger than the maximal theoretical value thanks to Eigen's fast exponential implementation).

As for the 1D part, we observe the same loss of performance (10% between the *general* and *product* version) due to the added loop on polar angles. As expected vectorizing the 1D solver does not yield as much speedup as the 2D solver due to the cost of data relocation in the *sse* version: the measured speedup is 2.18 with respect to the *general* version (or 2.4 with respect to the *product* implementation). This still allows for substantial global speedups for the spatial solver: 4.5 between the *general* and *sse* versions.

	2D (speedup)	1D (speedup)	2D–1D (speedup)
general	0.83	0.04	0.88
product	0.67 (×1.24)	0.09 (×0.51)	0.77 (×1.14)
sse	0.41 (×2.03)	0.07 (×0.58)	0.49 (×1.79)

Table 4: Computation times (s.) for different code versions, using the Diamond Differencing scheme in 3D.

Results for the Diamond Differencing scheme follow the same trend as in the 2D case: the lower arithmetic intensity limits the vectorization efficiency. We still obtain speedups around 2 for the 2D part, and 1.8 overall.

V. Conclusions & Future Work

We proposed in this paper a vectorized implementation of MICADO, a neutron transport solver based on a 2D–1D spatial resolution method. Both the 2D and 1D algorithms were vectorized, which required large changes in the implementation. Vectorization of the 2D Method of Characteristics (MOC) solver relies on the use of product quadrature formulas, which is quite acceptable in the authors' opinion since such quadrature formulas are routinely used in MOC calculations to reduce the storage needs for tracking information.

The implementation targets Intel SSE CPUs, but should also be suitable for more recent SIMD instruction sets such as AVX and AVX-512. The efficiency of this vectorized implementation was assessed on 2D and 3D geometries from the TAKEDA neutron transport benchmark, where nearly optimal speedups were obtained. Indeed, for the Step Characteristics scheme in single precision, the vectorized 2D solver demonstrates a speedup of 3.7 (for a maximal obtainable value of 4). The fully vectorized 3D spatial solver (including both the 2D MOC and 1D MOC-like solvers) allows obtaining speedups higher than 4.5 (which is above the theoretical maximal value thanks to the use of a fast vectorized exponential implementation).

All these results were performed using the solver in sequential settings. Perspectives include merging the new vectorized implementation with the parallel algorithms already implemented in MICADO. Since the distributed memory parallelism works on 2D slices, its performances should not be affected by the vectorization. Decreases in shared-memory parallel efficiency should however be expected since the current implementation parallelizes the loop on angular directions which was modified by the vectorization. A new shared-memory parallelization strategy could therefore need to be developed. Parallelizing the 2D sweep seems to be an idea worth investigating.

References

1) R. Sanchez, "Prospects in Deterministic Three-dimensional Whole-Core Transport Calculations," *Nuclear Engineering and Technology*, **44**, *2*, 113–150 (2012).
2) N. Cho, G. Lee, and C. Park, "Fusion of Method of Characteristics and Nodal Method for 3-D Whole-Core Transport Calculation," , **86**, 322–324 (2002).
3) S. Kosaka and T. Takeda, "Verification of 3D Heterogeneous Core Transport Calculation Utilizing Non-linear Iteration Technique," *Journal of Nuclear Science and Technology*, **41**, *6*, 645–654 (2004).
4) J. Y. Cho and H.-G. Joo, "Solution of the C5G7MOX benchmark three-dimensional extension problems by the DeCART direct whole core calculation code," *Progress in Nuclear Energy*, **48**, 456–466 (2006).
5) F. Févotte and B. Lathuilière, "MICADO : Parallel Implementation of a 2D–1D Iterative Algorithm for the 3D Neutron Transport Problem in Prismatic Geometries," *Proc. Proc. M&C2013*, Sun Valley, ID, USA, May, 2013.
6) F. Petrini et al., "Multicore surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine," *Proc. Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, p. 1–10, IEEE, 2007.
7) P. Micikevicius, "GPU Performance Analysis and Optimization," *Proc. GPU Technology Conference*, 2012.
8) M. J. Halsall, "CACTUS, a Characteristics Solution to the Neutron Transport Equations in Complicated Geometries," AEEW-R 1291, Atomic Energy Establishment (1980).
9) S. Santandrea, R. Sanchez, and P. Mosca, "A Linear Surface Characteristics Scheme for Neutron Transport in Unstructured Meshes," *Nuclear Science and Engineering*, **160**, 22–40 (2008).
10) A. Hébert, *Applied Reactor Physics*, Presses Internationales Polytechnique (2009).
11) A. Yamamoto, Y. Kitamura, and Y. Yamane, "Computational Efficiencies of Approximated Exponential Functions for Transport Calculations of the Characteristics Method," *Annals of Nuclear Energy*, **31**, 1027–1037 (2004).
12) R. Sanchez, L. Mao, and S. Santandrea, "Treatment of Boundary Conditions in Trajectory-Based Deterministic Transport Methods," *Nuclear Science and Engineering*, **140**, 23–50 (2002).
13) "Ivanoe – iDataPlex, Xeon X56xx 6C 2.93 GHz, Infiniband," http://top500.org/system/177030.
14) P. Gepner, V. Gamayunov, and D. L. Fraser, "Early performance evaluation of AVX for HPC," *Procedia Computer Science*, **4**, 452–460 (2011).
15) A. Yamamoto, M. Tabushi, N. Sugimura, T. Ushio, and M. Mori, "Derivation of Optimum Polar Angle Quadrature Set for the Methodof Characteristics Based on Approximation Error for the Bickley Function," *Journal of Nuclear Science and Technology*, **44**, *2*, 129–136 (2007).
16) T. Takeda and H. Ikeda, "Final Report on the 3-D Neutron Transport Benchmarks," OECD/NEA Comittee on Reactor Physics (1991).
17) "Eigen: a C++ template library for linear algebra," http://eigen.tuxfamily.org/index.php?title=Main_Page.