A generalized parallelization algorithm for Particle-In-Cell Simulations

R. A. Fonseca[1,2]

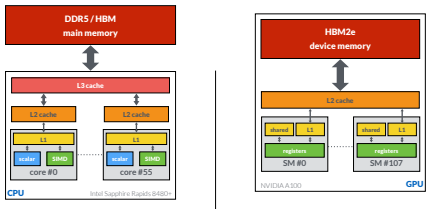
[1] GoLP/IPFN, Instituto Superior Técnico, Lisboa, Portugal
[2] DCTI, ISCTE-Instituto Universitário de Lisboa, Portugal

Abstract

Particle-in-cell (PIC) codes have been a cornerstone of plasma-based accelerator development. These work at the most fundamental, microscopic level, making few physics approximations, and are ideally suited to this problem. However, this makes them some of the most computationally expensive models in plasma physics. The current ecosystem of scientific computing systems relies on many different hardware approaches and vendors, each with specific programming models, memory architectures, and processor types, and efficiently deploying PIC codes on these architectures is paramount.

In this paper, we present a generalized parallelization algorithm for PIC simulations that is shown to work across all of the main architectures available today, including both CPUs (x86 / Arm) and GPUs (NVIDIA, AMD, Intel). The algorithm is based on a micro-spatial domain decomposition, with a high-performance particle manager to move particles between domains. Each domain is then assigned to a different thread (CPU) or thread block (GPU), achieving good parallel load balancing even for realistic simulation scenarios. The implementation is done using different programming models for different architectures, namely OpenMP (CPU), CUDA, ROCm (GPU), and SYCL (CPU/GPU/FPGA). While the implementations are effectively different code bases, given that the overall algorithm is the same, there are great similarities between all the implementations, making porting between them relatively straightforward. We present a performance comparison between different architectures/programming models for a test 2D problem, demonstrating very high performance for the architectures explored.

Modern parallel processors: CPU vs GPU



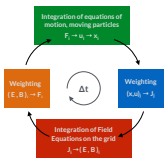
- Many-core architectures with up to ~ 64 cores per socket
- Very good scalar performance
- SIMD vector units up to 512 bits wide
- Multi-layer memory hierarchy, with large cache sizes per core
- DDR5 / HBM memory up to ~ 512 GB / 64 GB
- Good performance on non-sequential access

- Up to ~ 108 SM (streaming multiprocessors) per socket with up to 2048 threads per SM
- Very good thread engine
- Threads are executed in warps of 32 threads
- Explicitly controlled shared memory inside SM
- HBM2e memory up to 80 GB
- High-bandwidth for sequential access

Parallelizing the Particle-In-Cell algorithm

PIC algorithm

$$\frac{d\mathbf{u}}{dt} = \frac{q}{m}\left(\mathbf{E} + \frac{1}{\gamma c}\mathbf{u} \times \mathbf{B}\right)$$



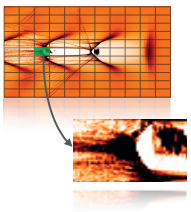
$$\frac{d\mathbf{E}}{dt} = c\nabla \times \mathbf{B} - 4\pi\mathbf{j}$$

$$\frac{d\mathbf{B}}{dt} = -c\nabla \times \mathbf{E}$$

- **PIC codes are good candidates for parallelization**
  - Work done on each particle is essentially independent
- **Particle advance/deposit represents the biggest challenge**
  - Most compute time is devoted to the particle advance
  - Each particle advance is generally independent from each other
- **Main difficulty relates to memory access**
  - Field interpolation can be very costly
    - Random access to global memory inefficient
    - Also affects current deposition
  - Current deposition may lead to memory collisions
    - 2 particles handled in parallel may deposit to the same cell

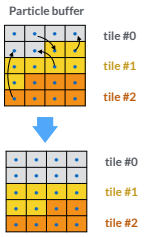
Generalized parallelization algorithm

- **Partition the simulation into small tiles**
  - Particles and grids are organized by small spatial regions on the order of ~ 2^D cells
  - Fields are copied from main memory to faster/local memory
  - Current deposit operates on fast / local memory
- **Computation in each tile is independent / local**
  - All tiles may be processed in parallel
  - After advancing the particles these may need to be assigned to different tiles
  - After advancing the fields edge values must be updated from neighboring tiles
- **Additional parallelism is used inside each tile**
  - Tile particles / cells may also be processed in parallel
  - Use atomic operations / serialization to avoid memory collisions during current deposit
- **Maintaining the particles organized by tile represents the biggest challenge**
  - Must also be efficiently performed in parallel



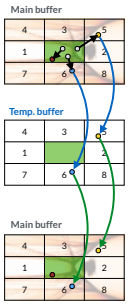
Particle tile sort

- **Main challenges**
  - Minimize data motion
  - **Keep particle buffer contiguous**
    - Minimizes memory requirements
    - Avoids growing buffers
- **Temporary buffers in main/device memory**
  - Indices of particles moving to another tile
  - Particles being moved to another position in the particle buffer
- **Process particles in 4 steps**
  1. Count particles crossing boundaries
  2. Get new tile offsets in particle buffer
  3. Copy particles moving away from tile to temp. memory
  4. Copy particles from temp. memory into tiles
- **Each step may be performed in parallel**
  - Action in each tile is (mostly) independent



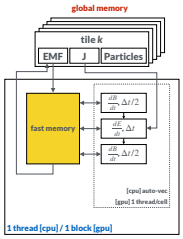
Particle tile sort

1. **Count particles crossing boundaries: `bnd_check`**
   - Loop over all particles and count particles staying / moving to other tiles
   - Store indexes of particles moving from tile
   - Get new number of particles in each tile
2. **Get new offsets: `update_offset`**
   - Prefix scan of number of particles per tile
   - Add room for new particles if needed
3. **Copy particles moving away from tile to temp. memory: `copy_out`**
   - Reserve space in temp. memory (using new offsets)
   - If particle moving to another node copy to tmp. memory and fill hole
   - If particle needs shifting (e.g. change of offset) also copy to temp. memory
4. **Copy particles from tmp. memory: `copy_in`**
   - Particles are stored at the space left at the end of local tile data



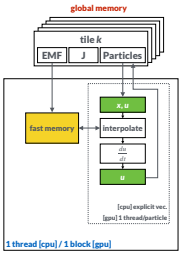
EM Field Advance

- **Main challenge**
  - Nonlinear memory access pattern (finite difference rotational operators)
- **Each thread (cpu) / block (gpu) handles 1 tile**
  - [gpu] use 1 thread per cell
  - [cpu] use auto vectorization
- **Copy tile E and B to fast memory**
  - Use block shared [gpu] or local [cpu] memory
  - [gpu] Fast access by all threads inside the block
- **Advance fields (Yee scheme)**
  - Do half B advance
  - Do full E advance
    - Use J directly from global memory
  - Do half B advance
- **Copy E and B back to main memory**
  - [gpu] Use coalescent memory access



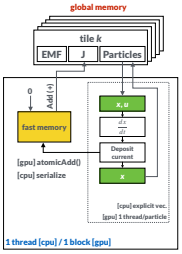
Momentum advance

- **Main challenge**
  - Random access to global memory
- **Each thread (cpu) / block (gpu) handles 1 tile**
  - [gpu] use 1 thread per particle
  - [cpu] use **explicit** vectorization
- **Copy tile E and B to fast memory**
  - Use block shared [gpu] or local [cpu] memory
  - [gpu] Fast access by all threads inside the block, similar to a memory cache
- **For each particle**
  - Read position and momentum from global memory
  - Interpolate EM fields
  - Advance momentum (Boris / Euler)
  - Store new momentum in global memory



Move / deposit

- **Main challenge**
  - Random access to device memory is costly
  - [gpu] atomic operations in device memory are costly
- **Each thread (cpu) / block (gpu) handles 1 tile**
  - [gpu] use 1 thread per particle
  - [cpu] use **explicit** vectorization
- **Create tile current grid in fast memory**
  - Zero this grid
- **For each particle**
  - Read position, momentum
  - Move particle (leap frog)
  - Split trajectory into segments fitting inside a single cell
  - Deposit current for each segment avoiding memory conflicts
    - [gpu] Use atomic add operations
    - [cpu] Serialize deposit
- **Add local current grid to global current grid**
  - Other species will also add to this grid



Implementation

- **Ideally we would like to use the same code base for all architectures**
  - The overall algorithm is very similar on all architectures
  - A single code base would simplify development
- **However, this leads to a significant performance penalty**
  - "One size fits all" programming models enforce hardware abstraction models that not always fit our algorithm
  - Achieving best performance requires version specializations which limit the benefits of these programming models
- **Best performance is naturally achieved with "native" programming models**
  - Explicitly manage hardware details
  - OpenMP / SIMD intrinsics for CPUs
  - CUDA / RoCM for NVIDIA / AMD GPUs
  - SYCL for Intel GPUs (also works with Intel CPUs)
- **Using the same algorithm greatly simplifies development**
  - Porting across platforms is straightforward
  - Many code blocks are identical / very similar

CPU parallelism

- **Main parallelism on CPU* is done through OpenMP**
  - Launch 1 thread per tile
  - Copy EM fields and current to local (stack) memory
  - Due to small tile size they essentially fit on L1 cache, but this is not enforced in any way
- **Additional parallelism through the use of vector (SIMD) explicit code**
  - Main issue are memory collisions in current deposit
  - At the final step current deposit is serialized
- **Implemented with hardware agnostic layer**
  - Particle advance implemented using generic vector instructions
  - Implement generic vector instructions using specific hardware intrinsics
- **All major current SIMD units are supported**
  - x86 AVX2 and AVX-512
  - Arm NEON and SVE


```
vfloat move( vfloat x0, vfloat vel, float dt ) {
  return vec_add( x0, vec_mul( vel, dt ) );
}
```


* SYCL implementation is also available

GPU Parallelism

- **GPU accelerators have a separate memory space**
  - Avoid CPU / GPU communication: run everything in the GPU
  - Only communication with the CPU is when doing I/O
- **Assign 1 thread-block per tile**
  - Copy EM fields and current to shared block memory
  - Use shared memory as cache
- **Assign multiple (~ 1k) threads per block**
  - Each threads handles 1 particle
  - Avoid memory collisions in current deposit through atomic operations in shared memory
- **All major current GPU units are supported**
  - **NVIDIA** : CUDA / ROCm / SYCL
  - **AMD** : ROCm
  - **Intel**: SYCL

Same algorithm, multiple toolkits

Launch parallel kernel

```
auto block = 256;
dim3 grid(
    particles -> ntiles.x,
    particles -> ntiles.y,
    1 );

kernel::move_deposit <<< grid, block >>> (
    *particles,
    J -> d_buffer, J -> offset, J -> ext_nx,
    dt_dx, q, qnx
);
```

cuda / rocm

```
auto alltiles = particles -> ntiles.y *
                particles -> ntiles.x;

#pragma omp parallel for schedule(dynamic)
for( int tid = 0; tid < alltiles; tid ++ ) {

    const uint2 tile_idx = make_uint2(
        tid % particles -> ntiles.x,
        tid / particles -> ntiles.x
    );
    move_deposit_kernel( tile_idx,
        *particles,
        J -> d_buffer, J -> offset, J -> ext_nx,
        dt_dx, q, qnx
    );
}
```

openmp

Parallel Computation

```
for( int i = threadIdx.x; i < np; i+= blockDim.x ) {
    float3 pu  = u[i];
    float2 x0  = x[i];
    int2   ix0 = ix[i];

    // Get 1 / Lorentz gamma
    float rg = rgamma( pu );

    // Get particle motion
    float2 delta = make_float2(
        dt_dx.x * rg * pu.x,
        dt_dx.y * rg * pu.y
    );

    // Advance position
    float2 x1 = make_float2(
        x0.x + delta.x,
        x0.y + delta.y
    );

    //(...)
}
```

cuda / rocm

```
for( int i = 0; i < np_vec; i+= vecwidth ) {
    vfloat3 pu      = vec_load_s3( (float *) & u[i] );
    vfloat2 const x0  = vec_load_s2( (float *) & x[i] );
    vint2   const ix0 = vec_load_s2( (int *) & ix[i] );

    // Get 1 / Lorentz gamma
    vfloat const rg = rgamma( pu );

    // Get particle motion
    vfloat2 const delta = {
        vec_mul( vec_mul( rg, pu.x ), dt_dx.x ),
        vec_mul( vec_mul( rg, pu.y ), dt_dx.y )
    };

    // Advance position
    vfloat2 x1 = {
        vec_add( x0.x , delta.x ),
        vec_add( x0.y , delta.y )
    };

    // (...)
}
```

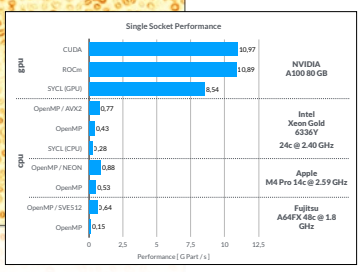
openmp (simd)

Performance per socket - Weibel simulation

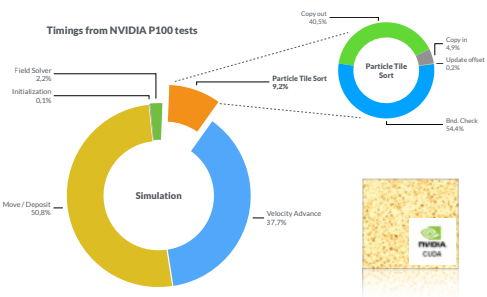
Simulation setup
- Collision of an electron and a positron plasma cloud
- 2D simulation in the perpendicular plane

B-field

$$f = 35\,\omega_p^{-1}$$

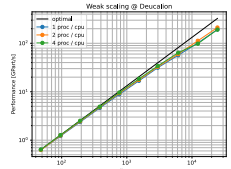


Particle advance takes ~ 90% of loop time



Using multiple nodes - MPI

- **The algorithm works well in combination with MPI**
  - Launch 1 MPI process per processing element (CPU / GPU)
- **The particle manager needs to be made MPI aware**
  - Exchange messages with all (8) neighbors
    - Number of particles leaving the node
    - Particle data
- **Interleave with single node algorithm**
  - After **`check_bnd`** exchange number of particles leaving edges with neighboring nodes
  - New offsets calculated in **`update_offset`** account for incoming particles
  - **`copy_out`** also copies particles into send message buffers
  - While particle data messages are being exchanged proceed with **`copy_in`**
  - Add particles received from other nodes to main buffers



Simulation setup
- Collision of an electron and a positron plasma cloud
- 2D simulation in the perpendicular plane
- Up to 16k × 8k cells, 17.2 G particles

Overview

- **The two prevailing computing architectures (cpu/gpu) share many commonalities**
  - High number of processing elements, SIMD parallelism
  - Layered memory hierarchy, with different sizes / speed
- **The same parallel algorithm may be used efficiently on both platforms**
  - The generalized PIC algorithm presented is shown to have top level performance on either platform
  - Implementation in native programming models is required, but porting between platforms is straightforward
- **The algorithm is easily extendable to distributed memory systems**
  - Can be scaled for use in large-scale HPC systems
- **Ongoing / Future work**
  - Finalize GPU MPI code
  - Python wrappers for interactive use
  - Implement using Kokkos
- **Full code available on github**
  - https://github.com/ricardo-fonseca/zpic-parallel

zpic@edu

GitHub