

GPU porting of ASCOT5 code for Monte Carlo simulations in fusion plasmas

M. Peybernes, G. Fourestey, S. Äkäslompolo, K. Särkimäki, F. Spiga

5th Fusion HPC Workshop





EUROfusion Advanced Computing Hubs

EUROfusion E-TASC – Theory and Advanced Simulation Coordination between:

- 14 TSVV (Theory, Simulation, Validation and Verification) projects
- 5 ACH (Advanced computing Hubs)

In particular, **3 HPC ACHs** were created in order **to help building power plants through numerical simulations**

- Extension of HLST (Roman Hatsky/IPP)
- Develop efficient, reliable tools
- Modernize and industrialize research codes

In order to **gain insight and predict fusion experiments** (ITER, JT60-SA, DEMO...)



EPFL

How to transition towards GPU codes

SCITAS

Plasma simulation codes are research codes:

- mostly CPU-only,
- written in C,C++ and Fortran,
- MPI and/or OpenMP,
- under active development by physicists, mathematicians, etc...
- main objective: GPU porting

General porting rules:

- least modifications of the code/no rewrite
- "maximum" performance
- portability/no specific target

3 approaches:

Library encapsulation (e.g. Kokkos, PETSc, AmgX, BLAS/Lapack...)

Cuda/R0Cm

Pragma directives (OpenMP offload / OpenACC)





ASCOT5 GPU porting



Outline

- > ASCOT5: particle orbit-following code
- MPI + OpenMP CPU algorithm
- GPU porting strategy
- Benchmarks
- > Profiling
- > Conclusion

EPFL ASCOT5





- ASCOT5 is a test **particle orbit-following** code for toroidal magnetically confined fusion devices
- The code uses the Monte Carlo method to solve the distribution of particles by following their trajectories.
 - The evolution of the distribution function for a test particle species *a* is described by the Fokker-Planck equation $\partial f_a = \nabla f_a + \frac{q_a}{r} (\mathbf{E} + \mathbf{v} \times \mathbf{B}) \nabla f_a = \sum \nabla \nabla f_a = \sum \nabla \nabla f_a + \nabla f_a$

$$\frac{\partial f_a}{\partial t} + \mathbf{v} \cdot \nabla f_a + \frac{q_a}{m_a} (\mathbf{E} + \mathbf{v} \times \mathbf{B}) \cdot \nabla_{\mathbf{v}} f_a = \sum_b -\nabla_{\mathbf{v}} \cdot [\mathbf{a}_{ab} f_a - \nabla_{\mathbf{v}} \cdot (\mathbf{D}_{ab} f_a)]$$

and **approximated by the Langevin equation** for a large number of markers that represent the distributed function:

 $d\mathbf{z} = [\dot{\mathbf{z}} + \mathbf{a}(\mathbf{z}, t)] dt + \boldsymbol{\sigma}(\mathbf{z}, t) \cdot d\boldsymbol{\mathcal{W}}$

- The particles undergo collisions with a static Maxwellian background plasma
- The detailed magnetic fields and the first wall can be fully 3D
- MPI + OpenMP (task-based) and highly vectorized



EPFL

ASCOT5 CPU version

CPU: MPI - OpenMP - Vectorized implementation:

- The time evolutions of each particle are independent from each other
- One + two levels of parallelism:
- MPI: Particles distributed among tasks, fields replicated
- OpenMP: queue based approach
- highly vectorized using the SIMD, originally developed for KNL manycore systems as target
- $\circ~$ to enable multithreading, a number of worker threads, each operating on a single set of N_{SIMD} arrays, are launched and allowed to perform their simulation independently
- \circ swapping mechanism
 - after each iteration, particles that have reached their end condition are stored in an array for completed particles
 - a fresh particle is retrieved from a queue to continue simulation in the particular slot in the N_{SIMD} arrays

Algorithm 1: CPU multithread vectorized algorithm

initialization; #pragma omp parallel while particles are alive in $pack_{NSIMD}$ do #pragma omp simd for particles $\in pack_{NSIMD}$ do move_particle; end #pragma omp simd for particles $\in pack_{NSIMD}$ do collisions; end #pragma omp simd for particles $\in pack_{N_{SIMD}}$ do end condition; end #pragma omp simd for particles $\in pack_{N_{SIMD}}$ do diagnostics; end for particles $\in pack_{N_{SIMD}}$ do if particle reached end condition then store particle and replace it by new one end end end





- GPU porting strategy
- Maintain a single version of the code
- Ensure code portability and readability
- Generic pragma for OpenMP/OpenACC

```
#ifndef gpu commands
#define gpu commands
/ * *
 * @brief Applies parallel execution to loops
 * /
#if defined(GPU) && defined(OPENMP)
#define GPU PARALLEL LOOP ALL LEVELS\
         str pragma (omp target teams distribute parallel for sim)d
#elif defined(GPU) && defined(OPENACC)
#define GPU PARALLEL LOOP ALL LEVELSstr pragma (acc parallel loop)
#else
#define GPU PARALLEL LOOP ALL LEVELSstr pragma (omp simd)
#endif
/ * *
 * @brief Maps variables to the target device
 * /
#if defined(GPU) && defined(OPENMP)
#define GPU MAP TO DEVICE(...) \
         str pragma (omp target enter data map(to: VA ARGS ))
#elif defined(GPU) && defined(OPENACC)
#define GPU MAP TO DEVICE(...) str pragma@cc enter data copyin
( VA ARGS ))
#else
#define GPU MAP TO DEVICE(...)
#endif
. . . . . . . . . . . .
#endif
#endif
```



GPU_LC	OOP_ALL_LEVELS							
for(i	= 0; i < n_queue_size;	i++) {						
if(p	->running[i]) {							
	<pre>posxyz[0] = posxyz0[0]</pre>	+ pxyz[0]	* h[i]	/	(2.0	*	gamma	1
mass);								
	<pre>posxyz[1] = posxyz0[1]</pre>	+ pxyz[1]	* h[i]	/	(2.0	*	gamma	
mass);								
	<pre>posxyz[2] = posxyz0[2]</pre>	+ pxyz[2]	* h[i]	/	(2.0	*	gamma	
mass);								
}								
GPU_EN	ND_LOOP_ALL_LEVELS							





- First implementation History-Based:
 - parallelism is expressed at a high level, emphasizing the independence of individual particles, allowing for concurrent execution of their respective histories from birth to death
 - each GPU processing unit is used to deal with the entire history of one or more particles until all of the particles have reached their end condition
 - this parallelism is implemented through a single monolithic GPU kernel

Algorithm 2: GPU algorithm - History-based initialization: #pragma acc parallel loop for all particles $\in \{1...N_{tot}\}$ do while particle is alive do move particle; collisions; end condition; diagnostics; end end





- The original implementation is not GPU-friendly:
 - one very large kernel
 - events depend on the previous event
- Implement a new version by splitting the initial kernel:
 - Parallelize over events instead of particles
 - small kernels independent of each other

Algorithm 2: GPU algorithm - History-ba	ased
initialization;	
#pragma acc parallel loop	
for all particles $\in \{1N_{tot}\}$ do	
while particle is alive do	
move_particle;	
collisions;	
end_condition;	
diagnostics;	
end	
end	

```
Aalto University
```

Algorithm 3: GPU algorithm - Event-based initialization: while number of particles alive > 0 do #pragma acc parallel loop for all particles $\in \{1...N_{tot}\}$ do if particle alive then move_particle; end end #pragma acc parallel loop for all particles $\in \{1...N_{tot}\}$ do if particle alive then collisions; end end #pragma acc parallel loop for all particles $\in \{1...N_{tot}\}$ do if particle alive then end condition; end end #pragma acc parallel loop for all particles $\in \{1...N_{tot}\}$ do if particle alive then diagnostics; end end end







- parallelize over events instead of particles
- small kernels independent of each other
- pack particles

Algor	ithm 2: GPU algorithm - History-based
initia	lization;
#pra	gma acc parallel loop
for a	all particles $\in \{1N_{tot}\}$ do
1	while particle is alive do
	move_particle;
	collisions;
	end_condition;
	diagnostics;
e	end
end	

Algorithm 3: GPU algorithm - Event-based initialization; while number of particles alive > 0 do #pragma acc parallel loop for all particles $\in \{1...N_{tot}\}$ do if particle alive then move_particle; end end #pragma acc parallel loop for all particles $\in \{1...N_{tot}\}$ do if particle alive then collisions; end end #pragma acc parallel loop for all particles $\in \{1...N_{tot}\}$ do if particle alive then end condition; end end #pragma acc parallel loop for all particles $\in \{1...N_{tot}\}$ do if particle alive then diagnostics; end end end

Algorithm 4: GPU algorithm - Event-based - packing initialization; $N_{pack} \leftarrow N_{tot};$ while number of particles alive > 0 do #pragma acc parallel loop for packed particles still alive $\in \{1...N_{pack}\}$ do move particle; end #pragma acc parallel loop for packed particles still alive $\in \{1...N_{pack}\}$ do collisions: end #pragma acc parallel loop for packed particles still alive $\in \{1...N_{pack}\}$ do end condition; end #pragma acc parallel loop for packed particles still alive $\in \{1...N_{pack}\}$ do diagnostics: end if $(N_{pack} - N_{running} > \alpha \cdot N_{tot})$ then pack particles: $N_{pack} \leftarrow N_{running};$ end end

EPFL Benchmarks



- Benchmark:
 - Collisional full-orbit simulation of prompt-losses of fusion alpha particles
 - o 2D wall; ITER-like but circular equilibrium interpolated with cubic splines
 - 2D wall rectangular, coulomb collisions, gyro orbit, simulation time = 0.0001s, fixed time step
 - Leonardo: A100, nvhpc/23.1
 - Comparison of three GPU implementations on GPU A100
 - Event-based packing algorithm is most efficient in all cases
 - Impact of Packing:
 - test_loadBalanced: Minimal impact due to majority of particles reaching end of simulation
 - test_loadUnbalanced: Significant impact with speedup of up to 1.41 compared to history-based algorithm and up to 1.22 compared to event-based one.



Comparison of the 3 particle-following GPU implementations - 1 Millions markers - 1 A100 Comparison of the 3 particle-following GPU implementations -10 Millions markers - 4 A100

EPFL Profiling Nsys



- Lower Local Memory Use: Event-based packing uses multiple smaller kernels, reducing local memory demands versus the history-based version.
- Efficient Data Transfer: Minimal data transfer overhead as all kernels run on the GPU.
- **Optimized Memory Access**: Contiguous, coalesced memory access through packing enhances efficiency.
- Reduced Loop Bounds: Through packing step, dynamic loop bounds improve runtime performance, with only ~30% particles active per timestep.





EPFL Profiling Nsight-Compute



100.0

80.0

90.0

- GPU Speed Of Light Throughpu

3 31				
igh-level overview of the throughput for compute and memory resources of the GPU. For each unit, th ompute and Memory to clearly identify the highest contributor. High-level overview of the utilization fo	e throughput reports the achieved or compute and memory resources	percentage of utilization with respect to the theoretical maximum. Brea of the GPU presented as a roofline chart.	akdowns show the throughput for each individual sub-metric of	
Compute (SM) Throughput [%]	18.22	Duration [usecond]		.50
/emory Throughput [%]	37.37	Elapsed Cycles [cycle]	11,3	332
.1/TEX Cache Throughput [%]	28.18	SM Active Cycles [cycle]	7,168.	.86
.2 Cache Throughput [%]	49.50	SM Frequency [cycle/nsecond]		
RAM Throughput [%]	0.00	DRAM Frequency [cycle/nsecond]		.54

exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues

HistoryBased

~	Latency issue	Look at <u>Scheduler S</u>	tatistics and <u>• Warp Stat</u>	e Statistics for potential	reasons.							
1	Roofline Anal	ysis The ratio of peak f analysis.	loat (fp32) to double (fp	64) performance on this	device is 2:1. The kernel a	chieved 0% of this device'	s fp32 peak performance	and 0% of its fp64 peak p	erformance. See the ᡚ	Kernel Profiling Guide for n	nore details on roofline	
	GPU Throughput											
Com	- oute (SM) [%]											
	Memory [%]						· · · · · · · · · · · · · · · · · · ·					
		0 10	1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2	10.0	20.0	0.0.5				900 0	0.0 10	
	0	.0 10	5.0 Z	.0.0		Speed Of L	aht (SOL) [%]		0.0	50.0 90		

 GPU Speed Of Light 	t Throughput			All
iigh-level overview of tl ompute and Memory t	the throughput for compute and memory resources of the GPU. For to clearly identify the highest contributor. High-level overview of the	each unit, the throughput reports the achieved percentage of utilization for compute and memory resources of the GPU pr	utilization with respect to the theoretical maximum. Breakdowns show the through esented as a roofline chart.	put for each individual sub-metric of
Compute (SM) Through	hput [%]	31.93 Duration [m	second]	
/lemory Throughput [%	6	59.03 Elapsed Cyc	cles [cycle]	2,923
I/TEX Cache Through	hput [%]	36.22 SM Active C	tycles [cycle]	2,893,240
2 Cache Throughput [[%]	74.85 SM Frequen	cy [cycle/nsecond]	
RAM Throughput [%]		59.03 DRAM Freq	uency [cycle/nsecond]	
	Look at <u>> Scheduler Statistics</u> and <u>> Warp State Statistics</u> for pot The following table lists the metrics that are key performance inc Metric Name	ntial reasons. cators: Value Guidance		
	gpu_compute_memory_throughput.avg.pct_of_peak_sustained	elapsed 59.0295 59.030 < 80.000		
	sm_throughput.avg.pct_of_peak_sustained_elapsed	31.9341 31.934 < 80.000		
← FP64/32 Utilizat ← Est. Speedup: 1	tion The ratio of peak float (fp32) to double (fp64) performan 6.09% determines that this kernel is fp64 bound, consider using	on this device is 2.1. The kernel achieved close to 0% of this 2-bit precision floating point operations to improve its perfor GPU Throughput	s device's fp32 peak performance and 23% of its fp64 peak performance. If <u>a Comp</u> mance. See the Revenet Profiling Guide for more details on roofline analysis.	
F	The second se			
Compute (SM) [%]				
t t				
Memory [%]				

Speed Of Light (SOL) [%]

EventBased

EPFL Benchmarks





- 10M markers Benchmark:
 - Collisional full-orbit simulation of prompt-losses of fusion alpha particles
 - 2D wall; ITER-like but circular equilibrium interpolated with cubic splines
 - 2D wall rectangular, coulomb collisions, gyro orbit, simulation time = 0.0001s, fixed time step
 - **Jed**: 2x Platinum 8360Y, intel/2021.6.0
 - Leonardo: A100, nvhpc/23.1
 - NVIDIA Grace Hopper Superchip engineering sample early access courtesy of NVIDIA
 - \circ Intel Ponte-Vecchio 600W (2 tiles) engineering sample early access courtesy of INTEL



EPFL Profiling





EventBased version:

- kernels mostly memory-bound
- multiple branch divergences in end_condition kernel involving lower Memory SOL due to thread divergence

Main kernels	%
move_particle	64.8
diagnostics	9.6
end_condition	6.5
collisions	5.8
copy_particles_structures	5.5
sorting	< 0.1
packing	< 0.1

TABLE I.RELATIVE WEIGHTS OF THE DIFFERENT STEPS OF THE SIMULATION ON A100. % Values Are Averaged Simulating 1 Million Particles With The Ascot5 Event-Based-Packing Algorithm

Main kernels	Memory SOL (%)	Compute SOL (%)
move_particle	68	30
diagnostics	80	26
end_condition	36	12
collisions	40	56

 $\begin{array}{c} {\tt table il.Test_loadBalanced, Speed Of Light - 1 Million Particles With \\ The Ascot5 Event-Based-Packing Algorithm \\ \end{array}$





Roofline







- **Successful GPU Transition**: ASCOT5 was efficiently ported from CPU to GPU using a directive-based strategy, ensuring code consistency.
- **Optimized Algorithms**: Three strategies were tested, with event-based-packing achieving the best performance due to improved load balancing and reduced thread divergence.
- **Significant Speedup**: Event-based-packing on H100-96GB shows up to 6x speedup over a dual Intel Xeon CPU node.
- **Future Work**: Conduct new tests incorporating enhanced physical models.