Speed optimal implementation of a fully relativistic 3d particle push with charge conserving current accumulation on modern processors

Kevin J. Bowers[1]
[1]*Plasma Physics Group (X-1), Los Alamos National Lab, Los Alamos, NM 87545*

1. INTRODUCTION

On modern processors, the floating-point subsystem performance far exceeds that of the memory subsystem such that traditional vector oriented implementations of particle-in-cell methods can only attain a small fraction of the theoretical performance. Building on prior work [1], a highly optimized fully relativistic 3d particle push / accumulate has been implemented. The push uses the standard leap-frog particle advance with a 6th order accurate Boris rotation (the same as [2]). The field interpolation and the charge conserving current accumulation are implemented according to the prescription in [3]. The data structures used allow for very flexible partitioning strategies and variably weighted particles.

The implementation has been carefully designed to minimize memory traffic between main memory and the processor, minimize cache thrashing, maximize the locality of memory accesses and to allow SIMD extensions on modern processors to be utilized portably. At the time of this writing, a particle advance of 7.2 MPA/S (million particles pushed and accumulated per second) has been achieved in the common case on an Intel Pentium 4 2.533GHz workstation with RDRAM memory and 6.1 MPA/S on an AMD Athlon 1.733GHz workstation with DDR266 DRAM memory subsystem. This can be shown to be near the theoretical workstation performance.

The high performance advance has been integrated into a 3d PIC code developed by the author. Presently the code is being used to do first principles astrophysical simulations of magnetic reconnection [4]. The new code decreased the turnaround time for these simulations compared with the traditionally implemented PIC code used previously over an order of magnitude. The author is routinely running hundreds of millions of particles for tens of thousands of time steps overnight on a modest 16-processor cluster. The code is being further developed for simulations of laser plasma interactions and radiographic beam devices.

2. CHARACTERISTICS OF MODERN COMMODITY PROCESSORS

2.1. Memory is "light years" away

Consider a workstation with 4GB of ECC DRAM. Taking the area of a single DRAM cell to be ~0.4μm^2 and the reduction of the speed-of-light for propagation in the circuit boards to be n_{eff} ~ 3, an optimistic order-of-magnitude estimate of the round trip signal propagation to an arbitrary memory address due to speed-of-light considerations alone is:

$$t_{round} \sim \frac{n_{eff}}{c} 4 \sqrt{\left(\frac{0.4\ \mu\mathrm{m}^2}{1\ \mathrm{cell}}\right)\left(\frac{9\ \mathrm{cells}}{1\ \mathrm{ECC\ byte}}\right)\left(\frac{2^{30}\mathrm{bytes}}{1\ \mathrm{GB}}\right) 4\ \mathrm{GB}} \sim 5\ \mathrm{ns}$$

For a 3GHz processor, memory is at least 15 processor clock cycles away. Further, issues like the memory access timings, chipset interactions, memory bus contention and so forth typically make the latency much worse (but these are more implementation issues than physical limitations).

2.2. Data locality is still important

Modern processors use a memory hierarchy to help reduce the above latency. At the top of the hierarchy is the register file. It has essentially no latency but it can only store a couple of hundreds of bytes. Next is the L1 cache. It typically has tens of kilobytes of low latency memory segmented into an instruction cache and a data cache. This is followed by the L2 cache, which contains hundreds of kilobytes of moderately fast memory. There sometimes are more cache levels. At the bottom of the hierarchy is main memory.

The processor transparently manages the contents of caches. For a code developer to take advantage of the caches requires some knowledge of typical cache behavior. An ideal cache keeps a copy of data needed for memory accesses in the not-too-distant future. Processors use simple heuristics to approximate this, such as replacing the least recently used cached data and by examining recent memory accesses to anticipate future accesses. Further complicating matters, hardware issues dictate that caches be built from “lines” (the granularity of memory stored in the cache) grouped into several “ways”. Ways simplify cache hit detection for a processor designer; a given memory transaction can only be stored in one place in each way.

The upshot of this is that data locality is still important. If multiple arrays are accessed simultaneously (in a traditional 3d PIC implementation this can be over 17 arrays in the particle advance and accumulate loops), many arrays will not be cached as there is no “way” to hold them. Likewise, if the caches are accessed in a random fashion, the processor cannot predict the accesses, further defeating the caches.

2.3. Dot products are much faster than AXPY’s

The rapidly increasing performance of commodity microprocessors has been largely driven by video game demand. With respect to scientific computing, this means that commodity processors are heavily optimized for single-precision short-vector calculations, particularly dot products. Both x86 and PowerPC based workstations provide SIMD instructions for these operations (SSE and AltiVec respectively).

Further, the memory bus control units are often optimized for dot-product calculations. A typical AXPY calculation (for example, $X_n = X_n + \delta_t V_n$) performs two memory loads and one memory store for every multiply and add. A dot product though does not need to perform any stores until the entire dot product is complete. The below benchmark of an AMD Athlon-MP 1.733GHz processor with DDR333 DRAM explicitly demonstrates this dot-product bias:

Rate of double precision loads from main memory:	330 million/second
Rate of double precision stores to main memory:	85.3 million/second

A dot product of two very large double precision vectors will run at ~660 MFLOP (load performance limited) while an AXPY of the same two vectors will never achieve more than 170 MFLOP (store performance limited). Given the above processor is theoretically capable of over 1.7 GFLOPS, an AXPY based calculation will only tap a small fraction of available performance.

2.4. Guides for optimal implementation of PIC codes on commodity processors

- Maximize temporal data-locality
  - Maintain a particle list sorted by cell so field and source lookups occurs in a simple predictable order. Order-n techniques exist which achieve this in a PIC simulation at virtually no cost [1].
- Maximize spatial data-locality
  - Instead of maintaining many different arrays of particle quantities (an x-coordinate array, a y-coordinate array, etc), particle quantities should be grouped into a structure.
  - Likewise, instead of maintaining many different arrays of field quantities, field quantities should be grouped into a structure.
- Minimize memory bandwidth (particularly store bandwidth)
  - As the particles are usually the largest data structure, the particles should be processed in one pass. The push and accumulate should be done at the same time.
  - Use single precision arithmetic if possible; it halves the bandwidth compared to double precision and allows single-precision short vector operations to be utilized.
- Reorganize loops to take advantage of short-vector SIMD operations
- Minimize number of floating point operations provided it does not conflict with the previous guidelines

3. IMPLEMENTATION DETAILS

3.1. Particles

```
typedef struct { int32 cell; float dx,dy,dz;
                 float ux,uy,uz,w; } particle_t;
```

This structure has many desirable properties. It maximizes spatial data-locality of particle information and it is exactly 32-bytes in size (which fits nicely into caches). The structure can be treated as two 32-bit 4-vectors. The structure explicitly records the cell containing the particle, eliminating the need for double precision calculation and for doing cell calculations of the form trunc[x/δ_x]. It stores the particle position in the coordinate system of the cell, greatly simplifying the field interpolation. It supports variable weighted particles, needed for many types of PIC simulations. Lastly, it can be used on general unstructured meshes without alteration.

3.2. The Interpolator

```
typedef struct { float ex, dexdy, dexdz, d2exdydz;
                 float ey, deydz, deydx, d2eydzdx;
                 float ez, dezdx, dezdy, d2ezdxdy;
                 float bx, dbxdx, by, dbydy;
                 float bz, dbzdz, pad0, pad1;} interpolator_t;
```

The interpolator stores all information necessary to interpolate the fields within a cell. Some information is redundant with neighboring cells. However, this redundancy allows all memory accesses needed for field interpolation for a given particle to be contiguous, maximizing data spatial locality. Further, since particle sorting [1] is used, temporal data locality is maximized and the memory accesses are likely to be cache hits. The interpolator above is for energy conserving interpolations with hexahedral cells described in [2]. Like the particle type, 4-vector SIMD operations are compatible with this structure.

3.3. The Accumulator

```
typedef struct { float jx00, jx10, jx01, jx11;
                 float jy00, jy10, jy01, jy11;
                 float jz00, jz10, jz01, jz11; } accumulator_t;
```

The accumulator accumulates current streaks made by particles passing through each cell. Like the interpolator, some information in the accumulator is redundant. But, as before, this maximizes data locality during particle accumulation. This structure is compatible with 4-vector SIMD operations also.

3.4. In-cell processor, cell-cross handler and the guard list

In a 3d PIC simulation of a uniform thermal plasma simulation with $\omega_p\delta_t \sim 0.2$ and $\delta_x \sim \lambda_d$, it can be shown that ~90% of the time a particle will not leave the cell containing it during a time step. Thus, in a bundle of 4 particles ~63% of the time none of the particles will leave the cell containing it. It is worthwhile to optimize for this common case.

The particle pusher tries to push a bundle of 4 particles at a time (using 4-vector operations) assuming that no particles in the bundle leave the cell containing it. When a cell-crosser is detected, a special case handler does the full charge conserving current accumulate algorithm. If the special case handler detects an interaction with a boundary or mesh partition, the particle's address is stored on the guard list for later processing.

This allows 4-vector operations to be used while keeping complex operations outside a streamlined common case loop. It allows different partition schemes to be utilized without changing the optimized particle loop. Further it allows the particle list to be processed in one pass (except for the very small fraction of particle which hit the guard list). To keep the implementation portable, common 4-vector operations are encapsulated into a C++ class which uses operator overloading to transparently access SIMD operations.

REFERENCES

[1] Bowers. "Accelerating a Particle-In-Cell Simulation Using a Hybrid Counting Sort." *Journal of Computational Physics.* 173. 393-411. 2001.

[2] Blahovec *et al.* "3-D ICEPIC simulations of the relativistic klystron oscillator." *IEEE Transactions on Plasma Science.* 28. 821. 2000.

[3] Eastwood *et al.* "Body-fitted electromagnetic PIC software for use on parallel computers." *Computer Physics Communications.* 87. 155-178. 1995.

[4] Bowers and Li. "Helicity Dissipation in 3d PIC Simulations of Magnetic Reconnection of a Force Free Confguration." To be presented at APS-DPP. Albuquerque, NM. 2003.