Big Astrophysical Data Visualisation on the MIC architecture

Claudio Gheller[1], Timothy Dykes[2], Mel Krokos[2] and Marzia Rivi[3]

[1] CSCS-ETHZ, Lugano, Switzerland
cgheller@cscs.ch
[2] University of Portsmouth, Portsmouth, U.K.
timothy.dykes@myport.ac.uk, mel.krokos@port.ac.uk
[3] University of Oxford, Oxford, U.K.
rivi@physics.ox.ac.uk

Abstract

....

1 Introduction

The scientific data volume produced by experiments, observations and numerical simulations is increasing exponentially with time. This is true for any scientific domain, but in particular for astrophysics. Next generations of telescopes and antennas are expected to produce enormous amount of data. The Square Kilometer Array project (REF), for instance, will generate an exabyte of data every day, twice the information currently exchanged on the internet on a daily basis and 100 times more information then the CERN LHC (REF) experiment produces. At the same time, computer simulations represent an invaluable instrument for astrophysicists to validate theories and compare to observations through numerical experiments. The ultimate cosmological simulations (REF), performed using sophisticated N-body codes (REF), could describe the details of the evolution of the universe up to the present time, following the behaviour of gravitating matter represented by a hundred billion particles. These runs produce output files whose size is of the order of tens of terabytes each. The fast technological progresss of supercomputing systems will soon lead to simulations producing outputs with size of the order of the petabyte or more.

Size does not represent the only challenge posed by scientific data. Is also essential to effectively extract all the information hidden in the sea of bytes represented by each single data file. Software for data mining and analysis is often highly computationally demanding and ultimately unusable on large datasets.

Visual exploration and discovery represents an outstanding aid to big data processing, e.g. by providing scientists with prompt and intuitive insights enabling them to identify interesting characteristics and thus define regions of interest within which to apply time-consuming methods. Additionally, they can be a very effective way in discovering and understanding correlations, associations and data patterns, or in identifying unexpected behaviours or even errors. Visualization is also an effective means for communicating scientific results not only to researchers but also to members of the general public. However, also visualization tools require high performance computing (hereafter HPC) resources, to fulfil the requirements posed both from data size and from the need of having fast (if not real-time) rendering.

The Splotch software (REF), our ray-casting algorithm for effectively visualizing large-scale, particle-based datasets, addresses these issues, providing high quality graphic outputs, processing data of, ideally, any size, already efficiently exploiting a broad variety of HPC systems:

multi-core processors and multi-node supercomputing systems (REF: Procedia Computer Science, 1(1) pp.1775-1784, 2010), and GPUs (REF1: Astronomical Society of the Pacific Conference Series, 475 (ADASS XXII) pp.103-106, 2013; REF2: in preparation). This paper will describe the work accomplished to enable Splotch to run on the new Intel PHI (REF) accelerator, taking advantage of the Many Integrated Core (hereafter MIC) architecture, which is expected to provide, on suitable classes of algorithms, outstanding performance with power consumption being comparable to standard CPUs. We will present the MIC implementation and optimisations, performance tuning, benchmarks carried out, and the resulting performance measurements, comparing that of an OpenMP based implementation running on multiple cores of a single CPU. A brief overview of this implementation will also be given, referring to [REF] for further details.

2 Splotch Overview

This section gives a brief overview to the Splotch algorithm, along with a description of the MPI and OpenMP related functionality for reference as these are key to the development of the MIC implementation.

Splotch is a pure C++ algorithm for generating 2 dimensional images from particle based datasets. It has no reliance on external libraries, minus those required for parallelism or particular file formats e.g. HDF5 [ref], and can be compiled from a downloadable self-contained tar-ball with a suitable makefile, use of preprocessor definitions and makefile switches allows compilation for a variety of hardware environments. The main stages of the Splotch workflow can be summarised as:

Data Load
: Source data is read using an appropriate reader and stored in the Splotch particle structure, various readers are implemented for data types such as HDF5, RAMSES, and GADGET. [ref]

Processing and Rasterization
: Data is preprocessed as per user requirements, performing tasks such as ranging, normalization, and applying logarithms to particle attributes, amongst others. Particles are roto-translated with reference to supplied camera and look-at positions along with an up-vector. Active particles, those within the view frustum, are identified and assigned an RGB color value dependant on particle properties and an external colour map, particles outside of the frustum are labelled inactive and not considered further.

Rendering
: For each pixel of the image, a ray is cast along the line of sight, and contributions of all encountered particles are additively accumulated. The contribution a particle may have to a particular pixel color is determined by solving the radiative transfer equation:

  [Insert radiative transfer equation]

  [Description of equations components and how they affect rendering]

2.1 Parallel Implementations

The OpenMP additions to Splotch [ref?] that provide the foundation of the MIC algorithm are outlined in this section. Following this is a brief description of the MPI features of Splotch, for reference in section [MPI offload sect].

2.1.1 OpenMP

OpenMP is employed to parallise core sections of Splotch algorithm if working in an OpenMP enabled environment. The rototranslation and coloring stages consist of applying a 3D transform to the particle coordinates and assigning an RGB value from a color lookup table. Each thread is simply assigned a portion of the particles with which to perform these stages.

The rendering is slightly more complex, due to the inevitability that multiple particles will fall on the same pixels. Threads cannot simply draw all assigned particles anywhere in the image without the possibility of race conditions where multiple threads attempt to draw to a pixel at the same time. To solve this, the image is split into 100x100 pixel tiles. The entire particle array is preprocessed in parallel to generate a list of particle indices per tile per thread, which is then condensed down to a single list of particle indices per tile. Each thread is then assigned a tile, on a first come first serve basis, and renders all particles in the tiles designated index list. To render a particle, the portion of the tile that is affected is calculated, and then processed in columns of pixels. Each pixel additively accumulates a contribution from all particles affecting it, the contribution a particle will have on the pixel is defined by the equation given in section [previous section rendering]

2.1.2 MPI

MPI is incorporated into both the file readers and the core algorithm for exploiting a distributed memory system. Each process simply loads a subsection of the dataset, and the serial or OpenMP method is used to render to a partial image, which is then accumulated to a final image by the root processor. [do we actually do openmp+mpi anywhere other than the mic algorithm?]

3 Splotch on the MIC

The MIC based implementation of Splotch uses both OpenMP and MPI in order to fully exploit the many core programming paradigm necessary to take advantage of this architecture. This section gives a brief overview of the Xeon Phi, and the steps taken to modify and optimise Splotch to make effective use of this hardware.

3.1 Overview of the MIC Architecture

The core ideal behind the MIC micro-architecture is obtaining a massive level of parallelism for high throughput performance in power restricted cluster environments. To this end Intel's flagship MIC product, the Xeon Phi, contains roughly 60 cores on a single chip, dependent on the model. Each core has access to a 512 KB private fully coherent L2 cache, memory controllers and the PCIe client logic provide access to up to 8 GB of GDDR5 memory, and a bi-directional ring interconnect brings these components together. The cores are in-order, however up to 4 hardware threads are supported to mitigate the latencies inherent with in-order execution. The Vector Processor Unit (hereafter VPU) is worthy of note due to the utilisation of an innovative 512 bit wide SIMD capability, allowing 16 single precision (SP) or 8 double precision (DP) floating point operations per cycle, support for fused-multiply-add operations increased this to 32 SP or 16 DP floating point operations per cycle.

The Xeon Phi acts as a coprocessor for a standard Intel Xeon processor connected via PCIe, and there are various modes of execution on a system utilising one or more Xeon Phi coprocessors. It runs the Linux operating system, and so can be seen as a networked node through a virtualised TCP/IP stack over the PCIe bus. This allows a user to log into the node, transfer a program over and run natively, or to use the coprocessor as an MPI process alongside the Xeon. The ability to partition subgroups of processors allows the Phi to run multiple MPI processes at the same time, a technique exploited in [insert subsection number]. In addition, a heterogenous approach is possible, using the coprocessor to accelerate a standard CPU based algorithm by offloading sections of computation to the device using a small series of directives available to both C++ and Fortran.

An advantageous factor of programming for the Xeon Phi, as opposed to other accelerators commonly used in HPC environments such as GPUs, is the similarity of techniques used to exploit parallelism on both the Xeon Phi and regular Xeon processors. Algorithms that already utilise parallel paradigms involving OpenMP, MPI, Intel TBB, or Intel Cilk Plus can often be run on the Xeon Phi with little modification. While further tuning is necessary to fully take advantage of the new hardware, it is not a necessity to reimplement the entire algorithm.

3.2 MIC Implementation

The implementation targeting the MIC is based on the original C++ and OpenMP version of Splotch (see section 2.1). While the core algorithm executes on the Xeon (referred to as the host), data is transferred to the Xeon Phi (referred to as the device) and the majority of the computation is offloaded using the C++ pragma extensions provided by Intel in a syntactic style similar to OpenMP, attention has been paid to optimising parts of the algorithm to better take advantage of the wide SIMD capability of the Phi.

3.2.1 Algorithm

The rendering algorithm can be broken down into a series of phases illustrated by the execution model shown in [figure x]. A double buffered scheme has been implemented using the ability to asynchronously transfer data via a series of signal and wait clauses provided by the Intel extensions. This allows to minimise overhead due to transferring data to the device for processing, and to facilitate the rendering of datasets potentially much larger than the memory capacity available, discussed further in section [section number].

[figure illustrating execution model]

A necessary overhead is incurred from the outset performing various initialisations discussed further in section 4.1. In addition to this, the first chunk of data must be transferred to the device before the double buffering system can begin to compensate for data transfer times. This initial transfer is carried out asynchronously while precomputing render parameters such as those used in the 3d transformation, which will be static throughout the rendering process. The render parameters are copied to the device, which subsequently begins rendering while the next chunk of data is transferred.

The first stage of rendering is a highly parallel 3D transform and colorize performed on a per-particle basis using four OpenMP threads per available core, to match the available hardware threads, equating to roughly 240 threads. This stage is amenable to fast processing and requires only a small amount of tuning for MIC suitability.

The multistep OpenMP solution to the full rendering stage discussed in section [omp section] is modified to account for the larger number of threads available. Simply splitting the image into smaller tiles to allow for more threads results in particles affecting more tiles than previously, requiring more memory to store particle index lists per tile. To account for this, the following approach is adopted.

The number of available OpenMP threads are split into groups, allocating an image buffer per group large enough to hold the entire resulting image. Each group is assigned a subsection of particles such that the entire array is evenly distributed. At this point each thread group acts independently, drawing the allocated subsection of particles to the allocated image, finally reducing all images into a single buffer when all particles of the current data chunk have been processed, a system conceptually similar to the MPI Splotch implementation discussed in section [mpi section]. The number of groups to create and number of threads per group can be passed in as a runtime parameter, and the chosen values should reflect consideration of the available number of threads and thread:core ratio. This is discussed further in section 4.1.

Subsequent to group allocation, each thread group begins independently rendering an allocated subset of particles. This occurs in two phases, a pre render phase and a render phase. In order to allow each thread sole access to a particular set of pixels, and avoid race conditions discussed previously, the image is split into 2 dimensional grid of tiles the number of which is determined by a run time parameter tile_size. The pre-render phase generates a list of particle indices per tile, indicating all the particles whose area of influence overlaps with the tile. In this phase each thread is allocated a subset of the group's allocated particles, and generates a list for each tile resulting in (n_thread * n_tile) lists. A single thread accumulates the per-thread lists to attain a single list per tile. Once all lists have been accumulated, phase two begins. In this phase each thread is allocated a tile, or subset of pixels, and renders all particles in the list associated with that tile. In this way pixels are not shared between threads and concurrent accesses are avoided.

Following the accumulation of each group-specific image into a single buffer, which is retained throughout the entire rendering process, the next chunk of data is processed until all particles have been rendered. This constant buffer is then copied back to the host for output.

3.3 Optimisation

In order to appropriately exploit the MIC architecture and attain high performance gains, various optimisation methods were explored. Some more generic methods are applicable to other similar architectures, such as Intel 64 and IA-32 [REF], while others are specific to the Xeon Phi.

3.3.1 Memory Usage

Often scientific datasets being visualized are very large, it is inevitable that there will at times be more data than available device memory. This raises the issue of allocating and waiting for data transfer from main memory which, while reaching roughly 6 GB/s over the 16 channel PCIe 2.0 connection, is an additional overhead to be considered. This is not such a large concern in a standard Intel 64 environment where frequent memory allocations, such as might be invoked by repetitive resizing of dynamic arrays, have little comparative impact.

Cost of dynamic memory allocation on the Phi is relatively high [REF1], in order to minimise unnecessary allocations buffers are created at the beginning of the program cycle and reused throughout. A single process offloading to the Phi and reserving large buffers could allocate at a speed of roughly 800MB/s, plus the smaller overhead of 6GB/s for data transferral. Use of the offload syntax for asynchronous computation [REF?] allows to engage a double buffered approach, processing the data in smaller chunks and overlapping computation and data transfer to minimise this overhead.

Furthermore, use of the MIC_USE_2MB_BUFFERS environment variable forces any buffers over a particular size to be allocated with 2MB rather than the default 4KB pages, which improves data allocation rate, transfer rate and can benefit performance by potentially reducing page faults and TLB (translation look-aside buffer) misses. [REF2] [Add times to show memory allocation/transfer improvement with this variable]

3.3.2 Vectorization

The large 512 bit width SIMD capability of the MIC architecture is addressed through vectorization carried out both automatically by the compiler, and manually using Intel Initial Many-Core Instructions (IMCI) [REF3] . Firstly the core data structure used in Splotch rendering was re-examined, and converted from an array of structures (AoS) to a structure of arrays (SoA). The AoS method stores a large array of structures, each containing all of the information pertaining to a single particle. While this is useful for encapsulation, it is unsuitable for SIMD processing. [REF4] [Run test with aos vs soa and show results?] This aids the compiler in automatic vectorization, amongst other changes such as ensuring the correct data alignment and modifying loops to be more easily vectorized as described in Intels Vectorization guide [REF 5] which, while providing examples for SSE [ref?], is applicable to IMCI as well.

Other areas are not so simply vectorized, and must be manually optimised through use of the Intel intrinsics, which map directly to IMCIs. During rendering, each thread draws all particles affecting a particular subsection of the final image. Particles are processed by drawing columns of affected pixels, from left to right, until all affected pixels have been updated, illustrated in section [insert section or figure]. Drawing consists of additively combining a pixels current value with the RGB contribution from the current particle, which is calculated by multiplying the particle colour by a contribution value. In order to expedite this process up to five single precision particle RGB values and five contribution values are packed into two respective 512 bit vector containers. A third container contains 5 affected pixels, which are written simultaneously using a fused multiply add vector intrinsic, masked in order not to affect the final unused float value in the 16-float capable containers. A complex optimisation such as this is not likely to automatically occur during compiler optimisation, however provides a useful performance increase. [add actual performance increase result] [clarify this explanation].

3.3.3 MPI offload

The overheads in dynamic allocation and data transfer incur a significant penalty when running a single host process offloading to the device. Data heavy algorithms such as Splotch require a lot of transfers, and ideally would use as much of the device memory at a time as possible. One such solution to this problem is to run multiple host MPI processes, each being awarded a subsection of the device to offload to and work with. This allows each process to allocate a share of the memory, transfer subsections of data, compute and render the results asynchronously, thereby utilising as much device memory as possible while minimising overheads. This provides a noticeable performance increase discussed further in section [performance analysis sect], although must be implemented through a fairly unwieldy script to distribute hardware threads amongst processes. [final sentence too opinionated?]

4 Performance Analysis

- Contents:
- Performance analysis methods, test hardware etc

4.1 Tuning

- Necessary overhead (mic initilisation, omp thread initialisation - check thread stack size allocations effect on this?)
- Thread grouping, tile size, number of mpi processes used, results

4.2 Scalability

- Performance tests on various sizes of data compared to CPU

5 Discussion and Conclusions

... [Discuss knights landing]

5.1 Acknowledgments

...