Big Data Visualization on the Xeon Phi

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

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

Abstract. *With the increasing size and complexity of data produced by large scale astrophysical simulations, it is important to be able to exploit all available hardware in High Performance Computing environments for increased throughput and efficiency. We focus on the modification and optimisation of Splotch, a scalable data visualization algorithm, to utilise the Xeon Phi, Intel's coprocessor based upon the new Many Integrated Core architecture. We discuss steps taken to offload data to the coprocessor and algorithmic modifications to aid faster processing on the many-core architecture and make use of the uniquely wide vector capabilities of the device, with accompanying performance results.*

Keywords: Big Data, Visualization, Xeon Phi, High Performance Computing, Astrophysics

1 Introduction

Nowadays dealing with big data effectively is a mandatory activity for a rapidly increasing number of scientific communities, e.g. in environmental, life and health sciences, and in particular in astrophysics. Some of the largest cosmological N-body simulations can describe the evolution of our universe up to present times by following the behaviour of gravitating matter represented by hundreds of billions of particles. Performing such simulations often produces time snapshots in the order of tens of terabytes. This situation can only be exacerbated as supercomputing advances are opening possibilities for simulations producing snapshots of sizes in the order of petabytes, or even exabytes towards the exascale era.

Large size is not the only challenge posed, it is also essential to effectively extract information from the typically complex datasets. Algorithms for data mining and analysis are often highly computationally demanding. Exploration and discovery through visualization can then be an outstanding aid, e.g. by providing scientists with prompt and intuitive insights enabling them to identify relevant characteristics and thus define regions of interest within which to apply further 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. However, visualization algorithms typically require High Performance Computing (HPC) resources to overcome issues related to rendering large and complex datasets in acceptable timeframes.

Splotch [1] is an algorithm for visualizing big particle-based datasets, providing high quality imagery while exploiting a broad variety of HPC systems such as multi-core processors, multi-node supercomputing systems [2], and also GPUs [3]. The ability to exploit all devices that may be available within a modern HPC system is of paramount importance towards achieving optimal overall performances. This paper reports on recent developments enabling Splotch to exploit the capability of the Intel Xeon PHI [5] coprocessor, taking advantage of the Many Integrated Core (MIC) architecture [6], which is envisaged to provide, on suitable classes of algorithms, outstanding performance with power consumption being comparable to standard CPUs. We describe our MIC implementation (Sect. 3) focusing on optimisation issues related to memory usage and data transfers and vectorization. We then discuss the performance details (Sect. 4) of our implementation using a benchmark dataset produced by a Gadget [4] N-Body simulation. Finally we summarise our experiences and present pointers to future developments.

2 Background

2.1 The Splotch Code

Splotch is implemented in pure C++ (i.e. no dependencies upon external libraries) and includes several readers supporting a number of popular formats for astrophysics. Datasets are converted into the Splotch internal format as they are loaded from files. This is followed by preprocessing and rasterization, and finally rendering. Preprocessing can perform ranging, normalization, and apply logarithms to particle attributes if necessary. Particles are then roto-translated with reference to supplied camera and look-at positions and assigned RGB color values appropriately using externally supplied color maps. For rendering pixels, rays are cast along lines of sight, and contributions of all encountered particles are accumulated. The contribution of particles is determined by solving the radiative transfer equation [3].

2.2 Overview of the Xeon Phi

The idea behind MIC is obtaining a massive level of parallelism for increasing 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. The Xeon Phi acts as a coprocessor for a standard Intel Xeon processor. Programs can be executed natively by logging into the device itself, which hosts a Linux micro-OS, or by using the device through one or more MPI processes amongst further processes running on the Xeon host. Alternatively users can offload data and portions of code to the coprocessor via a series of pragma based extensions available in C++ or Fortran. For a detailed technical description of the processor's architecture, the reader is referred to the Xeon Phi whitepaper [5]. Here, we give a short overview of its main features.

Each core has access to a 512 KB private fully coherent L2 cache and memory controllers and the PCIe client logic can access up to 8 GB of GDDR5 memory. A bi-directional ring interconnect brings these components together. The cores are in-order and up to 4 hardware threads are supported to mitigate the latencies inherent with in-order execution. The Vector Processor Unit (hereafter VPU) is worth mentioning 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. Finally support for fused-multiply-add operations increases this to 32 SP or 16 DP floating point operations.

3 Splotch on the MIC

3.1 Implementation

The Splotch algorithm can be broken down into a series of distinct phases illustrated by the execution model shown in Fig. 1. While the executable runs on the Xeon host, data and processing is offloaded to the device via Intel offload clauses. 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 data transfers, and to facilitate the visualization of datasets potentially much larger than the memory capacity available.

For rasterization, a highly parallel 3D transform and colorize is performed on a per-particle basis. Transform parameters are precomputed, as they are identical for each particle, and computation is distributed amongst threads via an OpenMP parallel for-loop. Further tuning enabled this loop to be auto-vectorized by the Intel compiler providing a performance boost for this phase.

The rendering phase begins with splitting the number of available OpenMP threads into groups, the number of which is provided by the user, allocating a buffer per group for the resulting image. The dataset is evenly distributed amongst the groups, each rendering to the assigned buffers which are finally accumulated serially.

Each group image is split into a two dimensional grid of tiles which are distributed amongst threads to avoid race conditions where multiple threads attempt to draw to a single pixel simultaneously. For each tile a list is generated containing the indices of all particles affecting that tile.

Each thread processes a set number of tiles, rendering the list of particles for each tile. In this way pixels are not shared between threads and concurrent accesses are avoided. Finally when all chunks of data have been processed and accumulated, the resultant device image is copied back to the host for output.

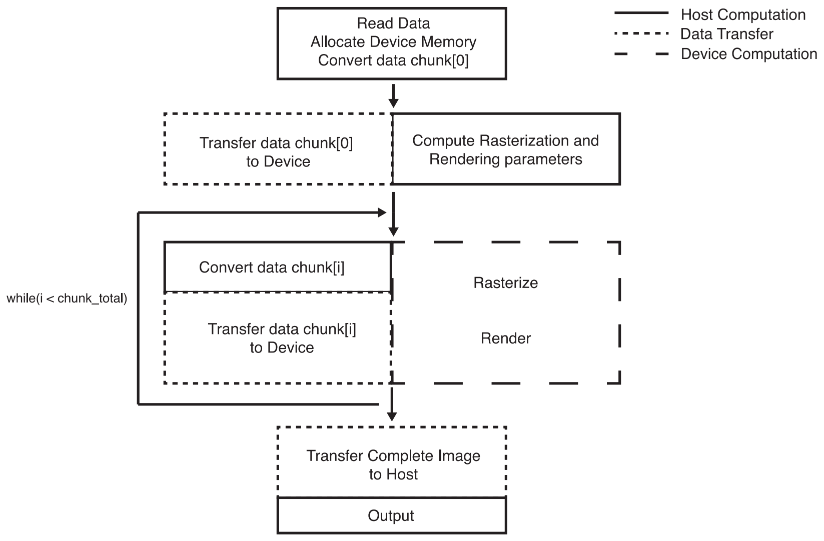

Fig. 1. Model illustrating execution flow of the new implementation.

3.2 Optimisation

Memory Usage and Data Transfer Cost of dynamic memory allocation on the Phi is relatively high [7], so in order to minimise unnecessary allocations buffers are created at the beginning of the program cycle and reused throughout. Use of the MIC_USE_2MB_BUFFERS environment variable forces buffers over a particular size to be allocated with 2MB pages rather than the default 4KB, which improves data allocation and transfer rates and can benefit performance by potentially reducing page faults and TLB (translation look-aside buffer) misses [8]. We have performed tests using offload clauses for compiler managed memory allocation on the device. To our experience a single process offloading to the device, and reserving large buffers, can potentially allocate memory roughly 2-2.5x faster having set this environment variable to 64K. The inability to asynchronously allocate offload memory means that overheads incurred allocating these buffers cannot be mitigated by overlapping allocation with host activity.

Additionally, overheads in dynamic allocation and data transfer incur a penalty when running a single host process offloading to the device. In order to minimise these penalties, we make use of the MPI implementation of Splotch. Multiple MPI processes on the host are each allocated a subset of the device threads to exploit. In this way, the device is subdivided amongst the host MPI processes allowing for memory and data transfer to occur in parallel providing a noticeable performance increase, further details of which are given in Sect. 4.

Vectorization The large 512 bit width SIMD capability of the MIC architecture is exploited through vectorization carried out both automatically by the compiler, and manually using Intel Initial Many-Core Instructions (IMCI) [9]. Firstly the particle data structure used in Splotch, a 36 byte structure consisting of 9 fields, was re-examined and converted from an array of structures (AoS) to a structure of arrays (SoA). 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. Such optimisations are described in Intel's Vectorization guide [10] which, while providing examples for SSE, is also applicable to IMCI. Reformatting the data storage method led to a 10% performance increase in the rasterization phase, while the rendering phase was essentially unaffected, leading to a 3% performance increase of the overall computation. The efficacy of this modification may vary, as the rasterization phase affects all particles in the dataset whereas the rendering phase affects only those particles viewable within the scene. Moreover, in the rasterization phase particles are both read and stored after few operations, while in the rendering phase they are simply read and not accessed further for the majority of the computation, therefore this optimisation has a minor impact on the performance.

As the rendering phase of the algorithm is complex, and thus unsuitable for automatic vectorization, this is manually optimised through use of the Intel intrinsics, which map directly to IMCIs. Drawing consists of additively combining a pixel's current RGB values with the contribution from the current particle, which is calculated by multiplying the particle color by a scalar contribution value. In order to expedite this process, up to five single precision particle RGB values (totalling 480 bits) and five scalar 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.

Tuning For relatively datasets where processing time is low, initialisation of the device and OpenMP threads can cause a noticeable overhead. The impact of this can be minimised by placing an empty offload clause with empty OpenMP parallel section near to the beginning of the program, in order to overlap this overhead while other host activity is occurring, in this case while reading from file. Alternatively the environment variable OFFLOAD_INIT can be set to on_start to pre-initialise all available MIC devices before the program begins execution.

Various parameters of the algorithm can be tuned to find best performance. Render parameters such as the number of thread groups and tile size are set to optimal defaults for the test hardware based on results of scripted tests iterating through sets of incremental potential values. These can be modified via a parameter file passed in at runtime for differing hardware.

4 Results

Performance analysis consisted of running a variety of tests on the Dommic facility of the Swiss National Supercomputing Centre, Lugano. In this system, each node is based upon a dual socket eight-core Intel Xeon 2670 processor architecture running at 2.6 GHz with 32 GB of main system memory. Two Xeon Phi 5110 MIC coprocessors are available per node.

Tests are carried out using an N-Body simulation performed using Gadget, consisting of roughly 21 million particles; 10 million dark matter particles, 10 million baryonic matter particles and 1 million star particles. A 100 frame animation orbiting the dataset is used to measure per-frame timings to produce an image of 800^2 pixels. For performance comparisons, both the host and device code use a tile size parameter of 100 pixels.

In the tests measuring the device performance, we run 8 MPI processes on the host all offloading to the same device, as discussed in Sect. 3.2 and set thread groups of 15 threads each, which turned out to be the optimal distribution for the test system. We use four OpenMP threads per available core to match the four available hardware threads, equating to roughly 240 threads. OpenMP threads on the host are mapped to one per core.

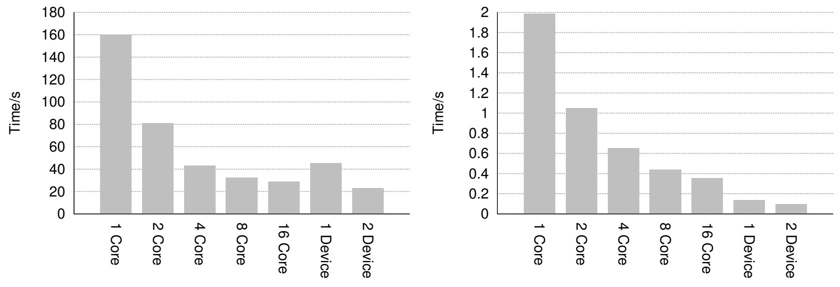

Fig. 2. *Left:* Per-Frame time for all phases: host OpenMP 1-16 cores vs single and dual Xeon Phi devices. *Right:* Per-Frame time for rasterization: host OpenMP 1-16 cores vs single and dual Xeon Phi devices.

Fig. 2 (*left*), describing per frame processing times of the OpenMP host implementation vs dual and single devices, shows that use of a single device provides comparable results to 4 cores on the host, while two devices outperform 16 cores by 20%, due to the non-linear scaling of the OpenMP implementation. The additional use of a second device provides a 2x performance improvement for the MIC algorithm. Fig. 2 (*right*) shows the strongest area of improvement, the rasterization phase, with a single device outperforming 16 cores roughly 2.5x, with roughly 3x improvement provided by using dual devices.

Fig. 3 shows comparison of per-frame processing times using varying numbers of MPI processes on the host, offloading to both single and dual devices, against the OpenMP and the MPI implementations of Splotch. Subdividing the available device threads amongst host MPI processes allows to transfer data and allocate memory in parallel while also more effectively spreading the workload across the device to ensure all threads are working equally. These tests show best performance with 8 and 16 MPI processes per device, possibly correspondent to the dual socket 8 core host CPU. It is interesting to note the current offloading model with a single device is comparable to 4 cores of the MPI host model, which scales linearly, with two devices comparable to 8 cores, leading in the direction of a linear scaling with number of devices.

These results suggest that an optimal usage of the Xeon Phi architecture consists of processing the data concurrently by both the multiple host cores and the device, coordinated through the MPI implementation, rather than using the host cores only for offload management. Data to be processed by the MIC can be offloaded asynchronously by the various host cores; while data from one core is copied to the device, the same core can continue its data rendering. At the same time, the MIC can process data already available that is previously sent by other cores.

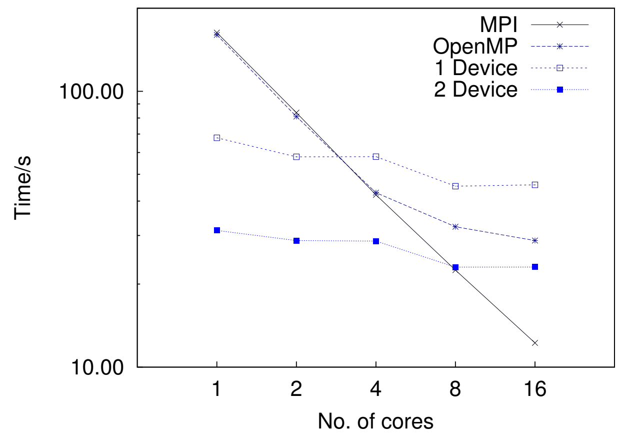

Fig. 3. Per-Frame processing time comparing MPI, OpenMP and MPI offloading to single and dual Xeon Phi devices.

5 Conclusions and Future Work

The results gathered so far demonstrate that in some areas of code the MIC architecture excels well beyond the host, although in others a fair amount of modification is necessary to gain acceptable performance levels. The use of MPI on the host provides a considerable performance increase especially for data heavy codes, however launching many MPI processes requires a lengthy execution script.

The high performance of the MPI host code indicates implementing an approach running MPI processes on both the host and device in order to fully exploit all available processing power may yield a higher overall performance, further work is planned to investigate this as described in the final paragraph of the previous section. In addition, work is planned to enable the code to run on multiple host nodes utilising all available devices, with further optimisation and testing to be done in order to optimise the double buffered data transfer and processing approach to effectively visualize much larger datasets. The ability to use multiple devices across multiple nodes will allow further testing of the scalability of the approach. Features provided in the new OpenMP 4.0 specification such as the *teams* construct will enable a OpenMP based approach to thread grouping as opposed to the manual method implemented here, if these constructs become supported by the Intel compiler in the future.

Intel released details of their second generation Xeon Phi product, codenamed Knights Landing, at the International Supercomputing Conference 2013 [11]. One important factor to note is the potential to use this as a standalone CPU rather than a coprocessor. This is worthy of further investigation as it invites the dismissal of complex and time consuming data transferral methods between a host and coprocessor, while retaining the ability to run directly on the host rather than copying executables to an external device.

References

1. Dolag, K., Reinecke, M., Gheller, C., Imboden, S.: Splotch: Visualizing Cosmological Simulations. New Journal of Physics, 10(12) id. 125006 (2008)
2. Jin, Z., Krokos, M., Rivi, M., Gheller, C., Dolag, K., Reinecke, M.: High-Performance Astrophysical Visualization using Splotch. Procedia Computer Science, 1(1) 1775–1784 (2010)
3. Rivi, M., Gheller, C., Dykes, T., Krokos, M., Dolag, K.: GPU Accelerated Particle Visualisation with Splotch. To appear in Astronomy and Computing (2014)
4. The Gadget Code, `http://www.mpa-garching.mpg.de/gadget/`
5. Chrysos, G.: Intel Xeon Phi coprocessor (codename Kights Corner). Hot Chips (2012)
6. Intel Many Integrated Core Architecture, `http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html`
7. Intel: Effective Use of the Intel Compiler's Offload Features (2013)
8. Intel: How to Use Huge Pages to Improve Application Performance on Intel Xeon Phi Coprocessor (2012)
9. Intel: Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual (2012)
10. Intel: A Guide to Vectorization with Intel C++ Compilers (2012)
11. Hazra, R.: Driving Industrial Innovation On the Path to Exascale: From Vision to Reality. International Supercomputing Conference (2013)