Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
1–13

DOI: 10.1177/1094342016652713
hpc.sagepub.com
SAGE

Splotch: porting and optimizing for the Xeon Phi

Timothy Dykes[1], Claudio Gheller[2], Marzia Rivi[3,4] and Mel Krokos[1]

Abstract

With the increasing size and complexity of data produced by large-scale numerical simulations, it is of primary importance for scientists to be able to exploit all available hardware in heterogenous high-performance computing environments for increased throughput and efficiency. We focus on the porting and optimization of Splotch, a scalable visualization algorithm, to utilize 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 using multiple Xeon Phi. Finally we compare performance against results achieved with the Graphics Processing Unit (GPU) based implementation of Splotch.

Keywords

Xeon Phi, high-performance computing, visualization, optimization, GPU

Introduction

Nowadays dealing with large data effectively is a mandatory activity for a rapidly increasing number of scientific communities, for example, 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 time by following the behavior of gravitating matter represented by many billions of particles. Performing such simulations often produces data outputs (or "time snapshots") in the order of multiple terabytes. This situation can only be exacerbated as advances in supercomputing are opening possibilities for simulations producing snapshots of sizes in the order of petabytes, or even exabytes to look toward the exa-scale era.

Large size is not the only challenge posed, it is also essential to effectively extract information from the typically complex data sets. Algorithms for data mining and analysis are often highly computationally demanding; visualization can be an outstanding analytical aid for further exploration and discovery, for example 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 intensive and time-consuming methods, thereby minimizing unnecessary expensive analysis. Furthermore, it is a very effective way of qualitatively discovering and understanding correlations, associations and data patterns, or in identifying unexpected behaviors or even errors. However, visualization algorithms typically require high-performance computing (HPC) resources to overcome issues related to rendering large and complex data sets in acceptable timeframes.

Splotch (Dolag et al., 2008) is an algorithm for visualizing large particle-based data sets, providing high quality imagery while exploiting a broad variety of HPC systems such as multi-core processors, multi-node supercomputing systems (Jin et al., 2010), and also GPUs (Rivi et al., 2014). The variety of implementations is due to the fact that many HPC systems of today are exploiting not only standard CPUs but accelerators to achieve maximum computational power with low energy usage. As of November 2015, 4 of the top 10 supercomputers in the world exploit GPUs or Xeon Phi coprocessors, including the number one supercomputer Tianhe-2 (developed by China's National University of

[1]School of Creative Technologies, University of Portsmouth, Portsmouth, UK
[2]CSCS-ETHZ, Lugano, Switzerland
[3]Department of Physics and Astronomy, UCL, London, UK
[4]Department of Physics, University of Oxford, Oxford, UK

Corresponding author:
Timothy Dykes, School of Creative Technologies, University of Portsmouth, Eldon Building, Winston Churchill Avenue, Portsmouth, PO1 2DJ, UK.
Email: timothy.dykes@port.ac.uk

Defense Technology) which runs on a combination of 12-core Intel Xeon E5 CPUs and Xeon Phi 31S1P devices (Top500org, 2015). The ability to fully exploit modern heterogenous HPC systems is of paramount importance toward achieving optimal overall performance. To this end, this article reports on recent developments enabling Splotch to exploit the capability of the Intel Xeon Phi (Chrysos, 2012) coprocessor, taking advantage of the Many Integrated Core (MIC) architecture (Intel, 2013a), which is envisaged to provide, on suitable classes of algorithms, outstanding performance with power consumption being comparable to standard CPUs.

Many developers have been exploring the possibility of using the MIC-based products to accelerate their software, as can be seen in the Intel Xeon Phi Applications and Solutions catalogue (Intel, 2016); high-performance visualization software developers are also on board with efforts being made to extend VisIt[1], an open source scientific visualization tool, to exploit the MIC architecture at the Intel Parallel Computing Center at the Joint Institute for Computational Sciences between the University of Tennessee and Oak Ridge National Laboratory. Developers are enticed not only by the large potential for compute power but also a key advertised feature of the Xeon Phi and MIC architecture: the ability for developers to work with regular programming tools and methods they would use for a standard Xeon (or indeed, other processors). This includes use of parallel application programming interfaces (APIs) and runtimes such as MPI and OpenMP from standard C++ and Fortran source code, while also offering the possibility of using more specialized options for parallelism such as Intel's Cilk Plus.[2]

Current experiences implementing algorithms or porting preexisting parallel codes to the Xeon Phi show many successes optimizing specific kernels and improving performance as compared to unoptimized kernels on Xeon Phi (e.g. Borovska and Ivanova, 2014; Gaburov and Cavecchi, 2014). Despite this, when comparing overall program performance against that achievable on a node with standard Xeon CPUs, authors may achieve similar or lower performance to the CPU (e.g. Elena and Rungger, 2014; Reid and Bethune, 2014), and it is not surprising considering the recent introduction of the architecture and the reported difficulties in achieving performance (e.g. Fang et al., 2014). This, however, is in the process of changing as more efforts are made to utilize the architecture with improvements showing in all areas as the technology matures. One particular success story of note is the exploitation of the Xeon Phi enabled Tianhe-2 to achieve peta-scale performance with an earthquake modeling simulation (Heinecke et al., 2014). It is expected that further success stories will appear as more machines are built exploiting this architecture and more developers investigate this area of technology, especially considering the commitment made to the architecture with new supercomputers featuring Xeon Phi *Knights Landing* chips commissioned by both the US Department of Energy's NERSC Centre and the NNSA's advanced simulation and computing program, delivering in 2016.

In view of this, throughout this article, we adapt the Splotch visualization algorithm to be suitable for the architecture of the Xeon Phi, in order to be ready to take advantage of large many-core systems. We share our experiences implementing an offloaded processing model and applying optimizations as outlined in the various Intel guides available, along with ideas of our own in regards to memory management and vectorization. With the rapidly increasing number of cores available in Xeon processors, many of these optimizations can also be applied on a Xeon CPU, and so we can also look to improve the OpenMP and MPI implementations of Splotch based on lessons learned throughout this experience.

The structure of the article is as follows: we provide a brief background to Splotch and the Xeon Phi (Section 2) and describe our MIC implementation (Section 3) focusing on optimization issues related to memory usage, data transfers, and vectorization along with discussion and examples of performance analysis methods. We then discuss the performance details (Section 4) of our implementation using a benchmark data set produced by a Gadget[3] N-Body simulation and a cluster of up to sixteen Xeon Phi devices. We include a comparison of performances achieved implementing the Splotch algorithm separately for Xeon Phi and GPU (Section 4.3), and finally in Section 5, we summarize our experiences and present pointers to future developments.

Background

The Splotch code

Splotch[4] is implemented in pure C++, with no dependencies upon external libraries except where necessary for external file formats (e.g. HDF5[5]), and includes several readers supporting a number of popular formats for astrophysics. Figure 1 shows a set of Splotch visualizations of the sample data set used for performance analysis, and Figure 2 the current execution model of the Splotch algorithm; data sets are converted into the Splotch internal format as they are loaded from files, followed by three key phases: preprocessing, rasterization, and finally rendering with an object-order ray-casting approach.

Preprocessing performs ranging, normalization, and optionally applies common functions to input fields (e.g. logarithm). At the rasterization stage, particle

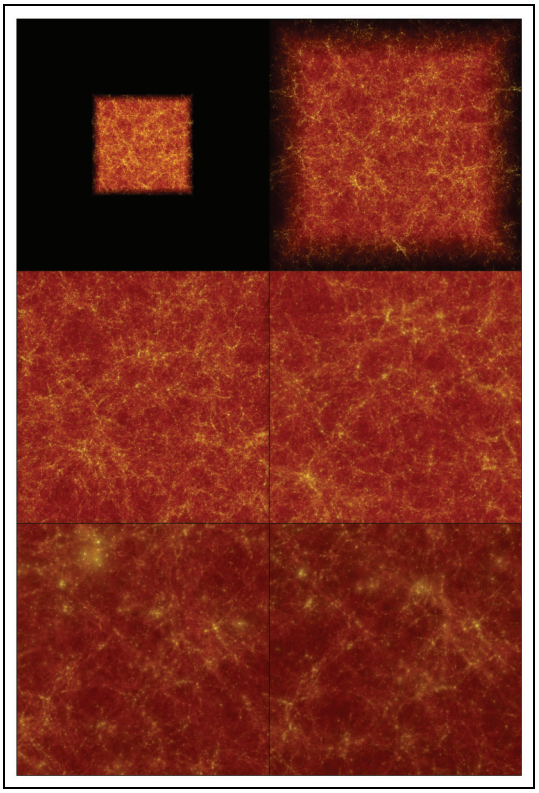

Figure 1. Six Splotch images of the data set used for performance testing, seen from the most remote distance (*top left*) and closest distance (*bottom right*). The images depict a data cube describing large-scale universe structure produced by an N-Body SPH simulation using the Gadget code.

positions represented as three-dimensional (3-D) Cartesian coordinates are transformed (roto-translated), and a perspective projection is applied with reference to supplied camera and look-at positions, followed by clipping and coloring. Colors stored as a triplet of single precision (S.P.) floating point values representing red, green, and blue components (RGB) are generated either directly from a vector input field or as a function of a scalar input field using a color map loaded at runtime. For rendering, rays are cast along lines of sight and contributions of all encountered particles are calculated based on the radiative transfer equation (for a more detailed description see Dolag et al., 2008).

The area of the image influenced by a particle is dependent on the radius, an intrinsic scalar component of each particle, and the distribution of affected pixels is obtained through a Gaussian kernel. A larger average radius throughout the data set results in longer rendering times, as particles affect larger portions of the image. This is caused for example by moving the camera close to or within the data set.

One notable feature of Splotch is the option to supply a *scene file*, which provides the ability to modify parameters, for example the brightness or color palette thresholds, over a progression of images which can then be stitched together to form a movie, allowing the user to visualize evolutionary properties of the simulated data set.

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 line, the Xeon Phi, contains roughly 60 cores on a single chip, dependent on the model, and acts as an accelerator for a standard Intel Xeon processor. Programs can be executed natively by logging via SSH into the device, which

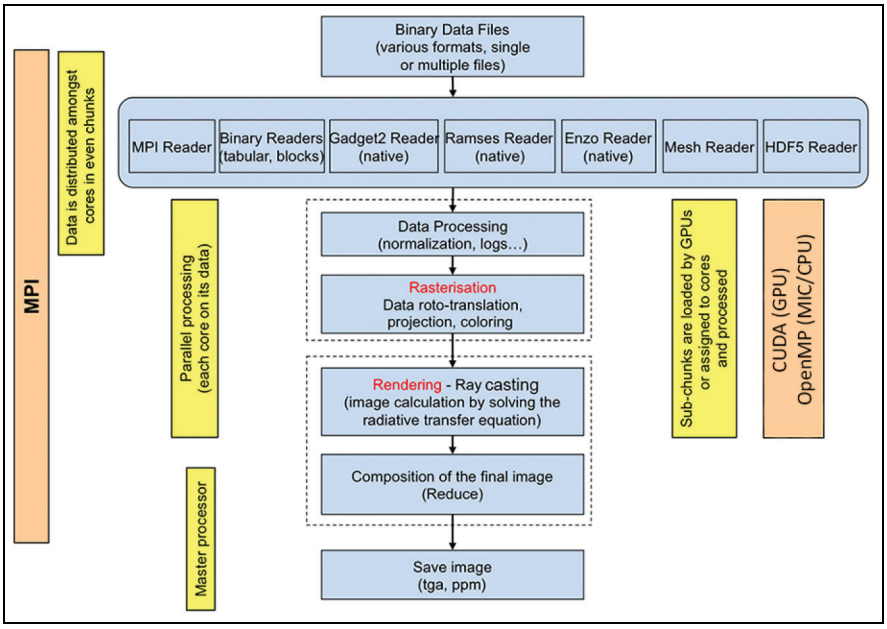

Figure 2. Standard execution model of the Splotch code.

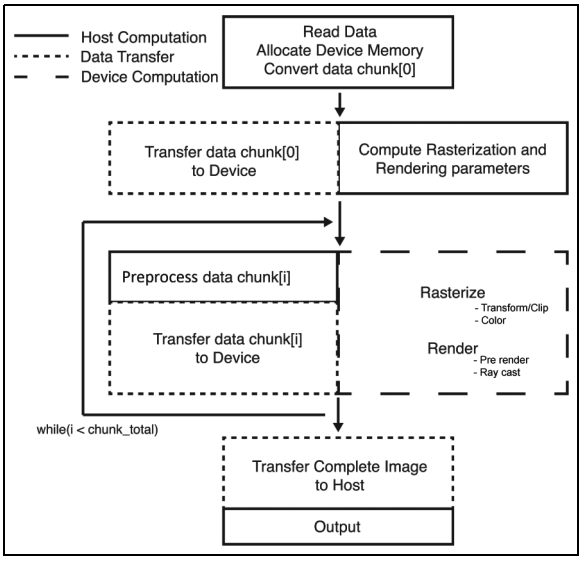

Figure 3. Model illustrating execution flow of the offloading implementation.

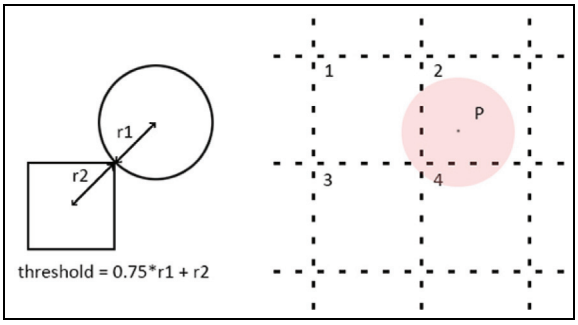

Figure 4. *Left*: Distance threshold definition below which a particle is considered to affect a tile. *Right*: Example of a particle projected onto a 2-D-tiled image. In this case, the particle is considered to affect tiles 1, 2, and 4, as the overlap between particle P and tile 3 is negligible.

hosts a Linux micro-OS, or by using the device through one or more MPI processes in tandem with those running on the Xeon host (symmetric mode). Alternatively users can offload data and portions of code to the coprocessor via Intel's language extensions for offload (LEO), 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 white paper (Chrysos, 2012). Here, we give a very 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 bidirectional ring interconnect brings these components together. The cores are in order and up to four hardware threads are supported to mitigate the latencies inherent with in-order execution. The vector processor unit (VPU) is worth mentioning due to the innovative 512-bit wide single instruction multiple data (SIMD) capability, allowing 16 S.P. or 8 double precision (D.P.) floating point operations per cycle, with support for fused multiply-add operations increasing this to 32 S.P. or 16 D.P. floating point operations.

Splotch on the MIC

Implementation

The Splotch algorithm ported to the Xeon Phi uses the offload model, as illustrated in Figure 3. While the executable runs on the Xeon host, data and processing are offloaded to the device via Intel's LEO. The ability to run already OpenMP- and MPI-based programs on the Xeon Phi means the swiftest approach to enable Splotch to run effectively on the device is to modify the current implementation, as opposed to moving to another software paradigm such as Intel's Cilk Plus[2] which may provide additional features but would involve a more thorough rewrite of the algorithm, future work is envisioned to also explore this paradigm.

The rasterization phase consists of a highly parallel 3-D transform, projection, clipping, and color assignment on a per-particle basis. These are split into two kernels, the transformation (including projection and clipping) and the coloring. Transform parameters are precomputed asynchronously, and work is distributed among threads via OpenMP parallel for-loops. The already highly parallel nature of these loops meant that no significant algorithmic modifications were needed in order for the code to run, however both are optimized through use of manual and automatic vectorization to provide a performance boost for this phase (see Section 3.2.2).

The rendering phase consists of a pre-render stage, and a ray-casting render stage combined into a single kernel in which image pixels are subdivided in a two-dimensional grid of tiles that are distributed among threads and rendered. The size of these square tiles is defined by a runtime parameter (see Section 3.3 for more). During the pre-render stage, a list of particle indices is created for each tile, indicating all particles affecting that tile (see Figures 4 and 5). Following this, each thread renders a full list by solving the radiative transfer equation along line-of-sight rays and retrieves another in round-robin fashion; in this way, pixel access is kept local to each tile and not shared between threads, avoiding concurrent access and race conditions. Finally when all chunks of data have been processed and accumulated, the resultant device image is copied back to the host for output. The original OpenMP rendering process described in Rivi et al.'s (2014) study has not been conceptually modified, rather the implementation has been optimized for MIC.

```
begin parallel block
  get_particles_for_thread(threadID, start, end)
  for each p in particles[start] : particles[end]
    get_affected_tiles(range, tile_size, p.x, p.y, p.radius)
    threshold = 0.75*p.radius + sqrt(2)*tile_size/2
    for each x in tiles[range.min_x] : tiles[range.max_x]
      cx = get_center(x)
      for each y in tiles[range.min_y] : tiles[range.max_y]
        cy = get_center(y)
        distance = length((p.x, p.y), (cx, cy))
        if(distance < threshold)
          indices_vector[thread_id][x][y].push_back(p.index)
        end if
      end for
    end for
  end for
end parallel block
```

Figure 5. Pseudocode illustrating the construction of vectors of particle indices which affect the tiles of a decomposed image.

To facilitate the visualization of data sets potentially much larger than the memory capacity available, while minimizing overhead due to data transfers, a double buffered scheme to overlap computation and data transfer has been implemented. We exploit the ability to asynchronously transfer data via a series of *signal* and *wait* clauses provided as part of the Intel C++ LEO extensions. This scheme creates two storage buffers; data from the host is copied to the first buffer, and while this data is processed the second buffer is asynchronously populated. The second buffer is then processed while the first buffer is asynchronously replaced with a new set of data. This loop can continue indefinitely while data is available in host memory, thereby solving the problem of limited device memory without costly delays in processing due to waiting for a full device buffer to be repopulated. The efficacy of this approach is dependent on the ratio of computation to Input/Output (I/O), i.e. we must spend more time rendering particles on the device than we do transferring data. The minimum computational cost a particle can incur in our scene is in the case where it affects a single pixel, a point-like particle. Referring to the results in Section 4.2, a rough estimate for the minimum computational time per-particle (smallest average radius) is 2.2E-8 seconds. Our tests show that on average we achieve 9.2E-9 seconds per-particle for data transfer. As such, the approach is beneficial even in the least computationally expensive type of scene.

Optimization

The three main kernels of our code, transformation, coloring, and rendering consume the majority of processing time. We focus on optimizing these by targeting a set of key problems known to cause performance issues with the Xeon Phi, as laid out in various resources for programming such devices.[6] Memory usage, data transfer, vectorization, and general tuning are discussed in the following subsections.

We used two key tools to analyze performance and identify target areas for optimization. The first is Intel VTune Amplifier XE, a profiling tool capable of measuring, analyzing, and visualizing performance of processing, both offloaded and native, on the Xeon Phi. Among other features, it simplifies the process of evaluating the benefit of manually inserted intrinsics in comparison to those generated by the compiler.

The second tool we adopted for tuning is the Performance API (PAPI) (Mucci et al., 1999). This API assists direct measurement of hardware events on the device. In particular, the ability to target a select set of statements with in-code hooks can be used for fine grained capture of hardware events in a large codebase. Moreover PAPI provides a high level of hardware event control in a lightweight form, without the autoanalysis and visualization options of a more fully featured tool such as VTune Amplifier. We used a small wrapper to facilitate use of PAPI in Intel's offload mode, which is available on Github with a sample benchmark and setup instructions.[7]

Memory usage. Cost of dynamic memory allocation on the Phi is relatively high (Intel, 2013b), so in order to minimize unnecessary allocations, buffers are created at the beginning of the program cycle and reused throughout. We found that we were unable to asynchronously allocate memory using offload clauses (allocating directly with a LEO clause as opposed to offloading a call to *malloc*), and so overheads incurred allocating these buffers cannot be mitigated by overlapping allocation with host activity. Use of the MIC_USE_2MB_BUFFERS environment variable forces buffers over a particular size to be allocated with 2 MB pages rather than the default 4 KB, which improves data allocation and transfer rates and can benefit performance by potentially reducing page faults and translation look-aside buffer (TLB) misses (Intel, 2012a). To our experience, a single process offloading to the device, and reserving large buffers, can allocate memory roughly 2–2.5× faster having set this environment variable appropriately. This reduces the cost for initial memory allocation (column 2 vs. 3 of Figure 8) and decreases L1 cache misses, however detrimentally affects kernel execution speed by increasing the L1 TLB miss ratio (Table 1). In this case, the environment variable is appropriate for single-image visualization, where initial memory allocation is a large proportion of the overall time. For movie generation, where initial allocation occurs only once, the benefit of faster allocation is outweighed by the increased kernel execution time and it is advantageous to allocate with 4 KB buffers as the default.

A notable issue for the many-core architecture is scalable memory allocation. When working with dynamic memory, for example through use of standard template library (STL) containers, parallel calls to *malloc* can be serialized (Intel, 2012b). In our case, this can occur in the pre-render phase, where the target image is geometrically decomposed into n tiles, and for each tile, a vector of particle indices is generated (as described in Figure 5). This requires each OpenMP thread to create n arrays, which are then filled with particle indices to be rendered. Each vector requests more memory as it reaches capacity, resulting in significant stalls caused by simultaneous allocations from different threads.

For the Xeon host, no attention is paid to potential memory allocation stalls, due to the relatively low number of threads, typically 16 or less. The number of threads active on the many-core architecture however is often much higher, resulting in thousands of dynamic arrays across many threads, necessitating a scalable solution to memory allocation. It is already possible to achieve sufficiently scalable memory allocation in parallel environments through use of external libraries, for example *ptmalloc* (Gloger, 2006) or Intel Threading Building Blocks *scalable_allocator* (Intel, 2014). However, in order to solve this problem without introducing dependancies on external libraries, we implemented a custom memory management solution consisting of a template array (*array_t*) and a memory allocator. Each thread is provided with an instance of the allocator, which asynchronously allocates a user-defined subset of storage from the device memory in the initial setup of the program. This "pooled" memory is then suballocated on request to local arrays, which greatly reduces risk of clashing calls to *malloc* from differing threads. The allocator implements bidirectional coalescence in order to minimize fragmentation, maintains the requested alignment for allocations, and if unable to provide the requested amount of memory, it will request this via a standard aligned allocation.

We compared performance of an STL vector container with and without Intel's scalable_allocator, against our custom vector implementation *array_t* with and without the pooled memory allocator. We can see in Figure 6 that performance for the STL container, and custom array implementation using standard aligned allocation methods (i.e. *_mm_malloc()*), decreases significantly as the particle count rises above two 2^{20} or roughly one million. The *array_t&allocator* retains high performance as all containers local to a thread are allocated memory from a dedicated per-thread memory pool. In the largest cases, where allocations begin to exceed local memory pools and are only limited by device memory, the custom allocator performs slightly better; this is in part due to faster 2 MB page allocation, which does not appear to benefit Intel's scalable allocator. The effect of using this allocator in Splotch can be seen in Figure 8, reducing the dominating pre-render stage in column 1 to be negligible in column 2.

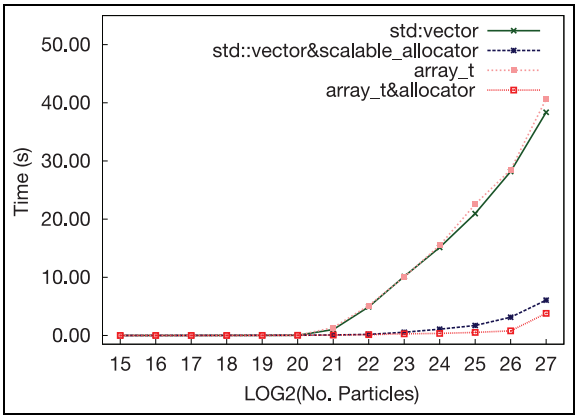

Figure 6. Comparing performance of an std:: vector with and without Intel's scalable_allocator against a custom array implementation (array_t) with and without pooled memory allocation.

Overheads in dynamic allocation and data transfer can incur a penalty when running a single host process offloading to the device. In order to minimize these penalties, we made 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 among the host MPI processes allowing to minimize overheads in data transfer, allocation and processing providing a noticeable performance increase, further details of which are given in Section 4 and similar experiences can be seen in Borovska and Ivanova's (2014) study.

Vectorization. The large 512-bit wide SIMD capability of the MIC architecture is exploited through vectorization carried out both automatically by the compiler, and manually using Intel intrinsics which map directly to Intel initial many-core instructions (IMCI) (Intel, 2012c). We discuss the vectorization process and performance gains for our three main kernels.

The roto-translation and filtering stage of rasterization is a fairly simple and highly parallelizable geometric transformation, there are no data interdependencies and minimal branching. It was modified to enable auto-vectorization through a series of steps described in Intel's Vectorization guide (Intel, 2012d) and feedback from the compiler vectorization report. While the auto-vectorization is not often instantly applicable, in simple cases such as this the modifications are fairly trivial compiler pragmas and alignment corrections.

Conversely, the coloring stage of rasterization involves a per-particle inner loop through an external color look-up table. This causes the compiler not to

Table 1. The effect of 2 MB versus 4 KB page allocation on overall cache and TLB usage, with subsequent effect on execution speed for the offloaded transformation kernel, obtained via Intel VTune.[a]

Derived metric	4 KB pages	2 MB pages	% Diff.	Effect
Per-Thread CPI	4.83	4.95	2.48	Negative
L1 Misses	54.0E + 06	42.0E + 6	−20.3	Positive
Estimated Latency Impact	10.4E + 03	14.5E + 03	39.4	Negative
L1 TLB Miss Ratio	0.001	0.014	1300	Negative
L1 TLB Misses per L2 TLB Miss	20.0	105	425	Negative
Transform kernel time/s	0.263	0.367	39.5	Negative

TBL: translation look-aside buffer.
[a]Both positive and negative values of the percentage differences can indicate a performance improvement, depending on the parameter. Beneficial factors are indicated as "positive" in the last column.

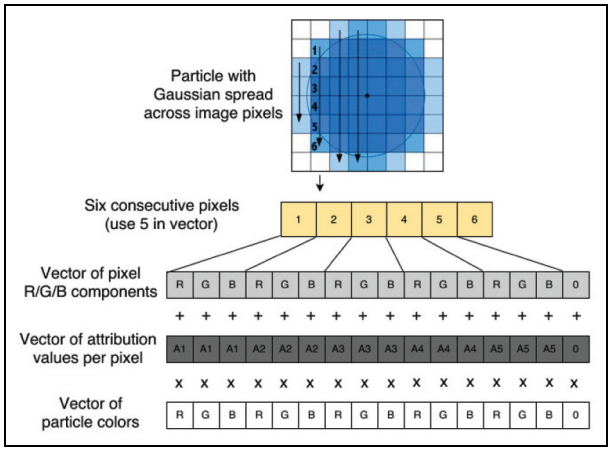

Figure 7. Vectorized update of up to five consecutive image pixels via fused multiply-add instruction.

auto-vectorize the most suitable loop that in our case is the outer one through particles. As a solution, we iterate through the outer loop in increments of 16 (i.e. the device S.P. SIMD vector capability) and manually vectorize the iteration; the remainder loop (if any) is processed in a non-vectorized fashion. We analyzed the performance of this modification with PAPI; in Table 2, a set of native hardware events plus derived metrics have been measured before and after the insertion of manual intrinsics. It can be seen from this example that the manual intrinsics doubled the speed of the kernel, while the measured FLOP/s increased by 60%.

Looking more closely at Table 2, contributors to the performance gain include the 20% decrease in instructions retired per thread (event: *instructions_executed*) coupled with a 30% drop in CPU cycles overall (event: *cpu_clk_unhalted*). The 7% rise in vector intensity (derived metric: *vpu_elements_active*/*vpu_instructions_executed*) indicates more of the vector elements in a VPU register were active on average during vectorized execution, leading to less instructions necessary. It is likely that the reformat of instructions involved in

Table 2. Comparing various hardware events and metrics measured with PAPI before and after manual vectorization of the colorize kernel.

Core Event	Before	After	% Diff.	Effect
time	0.16	0.07	−53.88	Positive
vpu_instructions_executed	2.18E + 09	1.49E + 09	−31.53	Positive
vpu_elements_active	9.00E + 09	6.60E + 09	−26.69	Positive[a]
cpu_clk_unhalted	3.54E + 10	2.31E + 10	−34.72	Positive
instructions_executed	8.08E + 09	6.24E + 09	−22.82	Positive
long_data_page_walk	8.94E + 03	7.99E + 03	−10.61	Positive
L2_data_read_miss_mem_fill	6.69E + 06	1.11E + 07	66.51	Negative
L2_data_write_miss_mem_fill	1.15E + 07	1.50E + 05	−98.69	Positive
Derived metric				
Vector intensity	4.13	4.42	7.07	Positive
GFLOP/s	55.82	88.73	58.97	Positive
Per-Thread CPI	4.38	3.7	−15.42	Positive
Per-Core CPI	1.09	0.93	−15.42	Positive
Read Bandwidth (bytes/clock)	0.13	0.13	−4.73	Negative
Write Bandwidth (bytes/clock)	0.07	0.09	28.89	Positive
Bandwidth (GB/s)	12.57	13.46	7.07	Positive

PAPI: Performance API.
[a]Positive when the percentage reduction in vpu_instructions_executed is larger.

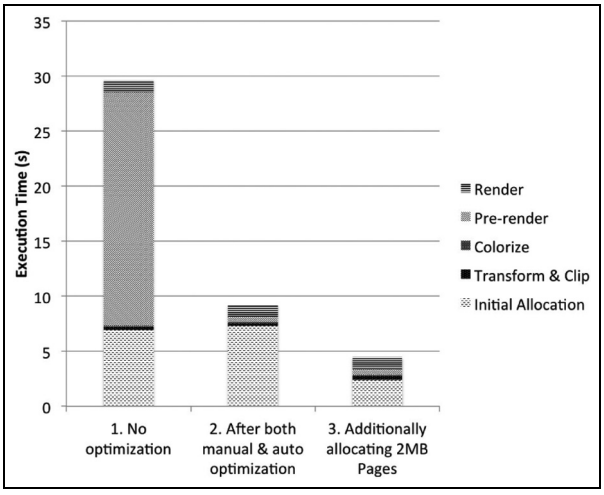

Figure 8. The effect of optimization: column 1 shows the code with no optimization, column 2 includes all optimizations discussed in Section 3.2, while column 3 includes 2 MB page allocation which greatly reduces initial allocation cost while subtly increasing kernel execution times due to cache usage.

manual vectorization had a higher impact on the reduced cycle count (and therefore reduced time to solution) than the mildly higher level of vectorization indicated by the vector intensity metric. We note that it can be difficult to interpret the meaning of hardware events when not accompanied by an increase or decrease in time, and so for more information on the capture, utilization and derivation of hardware metrics the reader is referred to the relevant Intel optimization guide (Intel, 2012e).

The rendering phase of the algorithm is difficult for the compiler to auto-vectorize due to a loop through non-consecutive particles followed by partially consecutive pixel updates. We additively combine an image pixel's current RGB values with the contribution from the current particle. This is calculated by multiplying the particle color by a scalar contribution value representing the Gaussian spread over the distance between the pixel and the particle centre on each axis. For each pixel on the horizontal axis, we calculate a scalar contribution value per-pixel on the vertical axis (illustrated in Figure 7). Only the inner vertical axis loop update contains consecutive memory accesses for vectorization. In this case, we manually optimized the kernel through extensive use of intrinsics: 5 S.P. particle RGB values (totaling 480 bits) and five scalar contribution values are packed into two respective 512-bit vector registers, V1 and V2; a third register, V3 contains five affected pixels, which are written simultaneously using a fused-multiply-add vector intrinsic, and written back to the image with an unaligned store intrinsic combination. This optimization reduced render kernel computing times up to 10% dependent on the scenes rendered, and our testing indicated the overhead of loading and unloading vector registers was only outweighed by processing five consecutive pixels simultaneously, rather than any smaller number.

The experience of trying to push the vectorization capabilities of the compiler and investigating different areas of the algorithm in an effort to optimize for vectorization has led to our recommendation that while the compiler can be very useful in automatically vectorizing code, it is still possible to gain significant performance boosts for complex kernels by manually inserting intrinsics.

Tuning

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 in a heuristic manner to optimal defaults for the test hardware based on results of scripted tests iterating through incremental sets of potential values. These can be modified for differing hardware via a parameter file passed in at runtime.

For relatively small data sets where processing time is low, that is, a matter of seconds, initialization of the device and OpenMP threads can cause a noticeable overhead. The impact of this can be minimized by placing an empty offload clause with an 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 input data. Alternatively the environment variable OFFLOAD_INIT can be set to pre-initialize all available MIC devices before the program begins execution.

Thread count and affinity are important factors in the tuning process. We examined the effect of varying the number of threads per core and thread affinity for a series of Splotch renderings. We ran a set of tests varying the camera position in order to have a fair comparison of the effect as a function of the average particle radius (see Figure 9). We tested with one to four threads per core, four being the maximum number of hardware contexts available per core. A series of preliminary tests indicated the *scatter* affinity[8] is ideal for our use case and so we set this configuration for the final tests, although we noted that in the case of 4 threads per core the difference between affinity settings (in particular *scatter* and *balanced*) was negligible most of the time, as also mentioned in Reid and Bethune (2014).

From Figure 9, we see that the best performance is obtained for four threads per core. It is important to run these tests, which can be trivially scripted, for any multi-threaded application as it is likely the suitable configuration will be different from ours. It can also be noted that, as expected, the gap between one to two threads per core is noticeably larger than between two to three and three to four; this is likely due to the

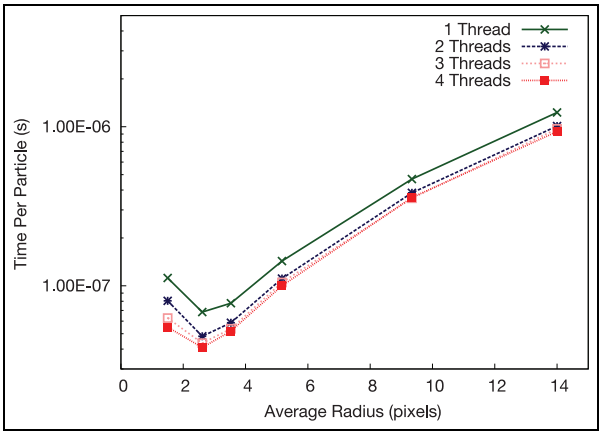

Figure 9. Comparison of performance with one to four threads per core, using optimal *scatter* affinity settings.

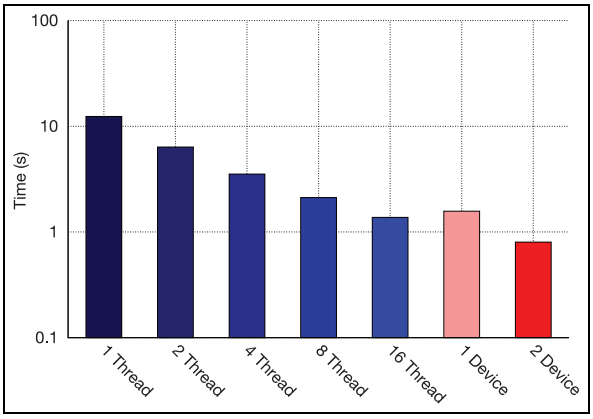

Figure 10. Per-frame total processing time: Xeon OpenMP 1-16 threads versus single and dual Xeon Phi devices.

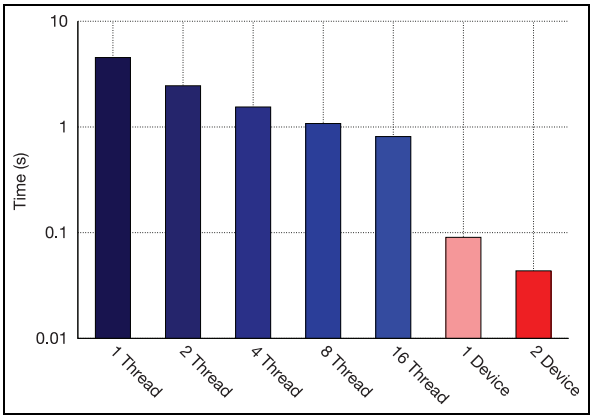

Figure 11. Per-frame rasterization time: Xeon OpenMP 1-16 threads versus single and dual Xeon Phi devices.

inability of the core to issue two instructions from a single hardware thread context in back to back cycles (Intel, 2012f).

Results

Hardware and test scenario

All tests were performed using the Dommic facility at the Swiss National Supercomputing Centre, Lugano. In this eight node cluster, each individual 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, making up to sixteen *Knights Corner* coprocessors available.

The sample data set used for measurements is a snapshot of an N-Body SPH simulation performed using the Gadget code. For the single node animation tests of Section 4.2.1, we filter to 50 million gas particles and five million star particles (~1.8 GB) in order to process a single chunk of data and more accurately measure individual kernels over many animation frames, whereas for all other tests we use the full size of 200 million gas particles and 20 million star particles (~7.2 GB).

MIC Performance

A 100 frame animation with the camera orbiting the data is used to measure average per-frame timings producing images of 1024^2 pixels. For performance comparisons, the device uses a tile size parameter of 40 pixels, a heuristically chosen optimal value; due to cache sharing with particle data, varying this parameter within reasonable range (i.e. more 'natural' choices such as 32 or 64 pixels) has little effect on performance.

For clarity, in all OpenMP tests on the Xeon, we use one thread per core. For MPI offloading to device, each task is allocated an even share of the 236 hardware thread contexts on the device which are then exploited with OpenMP, and in this case, thread binding is set explicitly. In the case of one task using the whole device, thread affinity is set as per the optimal settings for our use case (Section 3.3).

Single node speed-up. We compare performance of the OpenMP parallel version of Splotch on the Xeon and the Xeon Phi implementation exploiting both OpenMP and MPI. Figure 10, describing per frame processing times of the OpenMP Xeon implementation versus dual and single devices, shows that use of a single device provides results close to 16 threads on the Xeon. Figure 11 shows the strongest area of improvement, the rasterization phase, with a single device outperforming 16 threads (two CPUs) roughly 9×, with roughly 18× improvement provided by using dual devices. In both cases, the use of a second device provides a 2× performance improvement for the MIC algorithm. The best performance in terms of FLOP/s is achieved in the transform kernel, which roto-translates, projects and

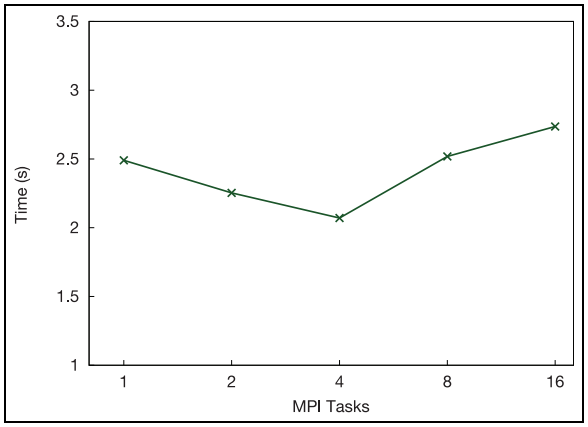

Figure 12. Per-frame processing time comparing multiple MPI tasks on the host sharing a single Xeon Phi; each task is further parallelized through OpenMP to use the full number of hardware thread contexts available.

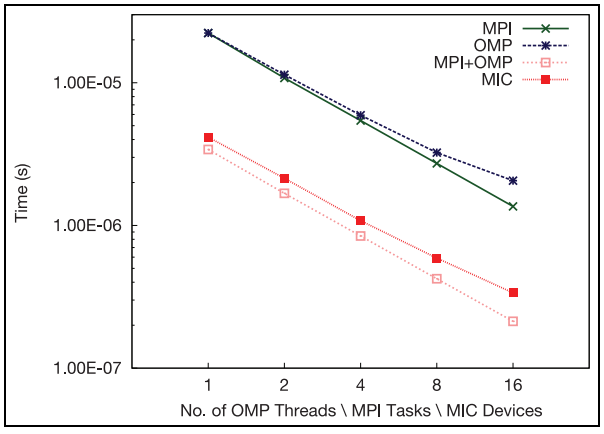

Figure 13. Scalability: per-particle processing time for varying models from serial to highly parallel and Xeon Phi.

filters particles as a subset of the rasterization phase. PAPI measurements indicate this kernel in a single device achieves approximately 300 GFlop/s, or 15% of peak S.P. performance.

Figure 12 shows comparison of per-frame processing times using varying numbers of MPI processes on the host, offloading OpenMP parallelized code to a single Xeon Phi. Subdividing the available device threads among MPI processes allows to more effectively spread the workload across the device to ensure all threads are working equally. These tests show best performance with 4 MPI processes, above this it appears that the overhead of additional MPI processes causes performance to deteriorate.

Multiple node scalability. We employ three sets of tests running from serial to highly parallel with multiple paradigms using a dual socket 16 core node; 1–16 OpenMP threads, 1–16 single threaded MPI tasks with one task per core, and 1–16 MPI tasks with one task per CPU and 8 OpenMP threads per task (see Figure 16). These are compared with a final set with up to 16 Xeon Phi (Figure 13).

It can be seen that for the full Splotch code, we currently achieve performance with one Xeon Phi roughly similar to one CPU parallelized with OpenMP. The use of a data set larger than device memory causes a decrease in performance in comparison to that shown in Section 4.2.1. The non-linear scaling for the Xeon OpenMP implementation is due to locality issues during rendering, as threads access particles according to their position when projected onto an image rather than their location in memory. This is not an issue for the MPI implementation as each task renders particles independently of the other tasks, and there is no risk of non-local memory access. Scalability of the MIC is non-linear in the 8–16 range, this is due to the data set not being large enough to fully exploit the power of the device when subdivided, we expect to see more linear scaling using larger data sets with device counts ranging above 8.

Splotch: MIC versus GPU

Our experience of implementing the Splotch code for both GPUs and Xeon Phi allows to make a comparison of the performance we have achieved through similar expenditures of time and effort. In the case of the GPU, we implemented our algorithm via CUDA (Rivi et al., 2014), while in the case of the Xeon Phi, we use our optimized parallel model with OpenMP, and in both cases, MPI can also be exploited. We use the same full data set and host processors for performance tests. We render a single image of 1024^2 pixels for six different camera positions (Figure 1, starting from very far and reaching progressively closer to the center of the simulation). In order to make a comparison, we measure on a per-particle basis the total compute time (i.e. full algorithm execution minus data read and image output), the rendering kernel time, and the rasterization kernel time. For completeness, we also compare against eight OpenMP threads exploiting an 8-core Intel Xeon E5-2670.

Figure 14 shows total performance of a single NVIDIA K20X GPU versus a Xeon Phi, as function of the average particle radius. The average radius, while not being the only factor affecting rendering time (see Rivi et al., 2014, for more detail), is a useful metric for comparison; a larger radius means the particle will affect more of the image and computational cost will increase. It is clear that although the GPU implementation outperforms the Xeon Phi in all tests, for the larger radii the results are very similar.

Figures 15 and 16 show kernel specific performance on a per particle basis. The performance difference

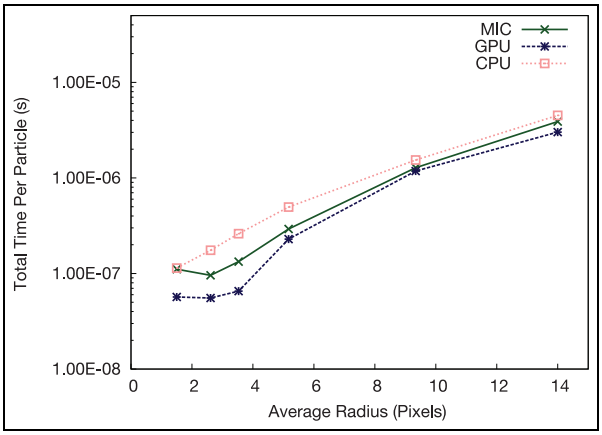

Figure 14. Xeon Phi versus NVIDIA K20X versus Xeon CPU: comparison of the total compute times on a per-particle basis.

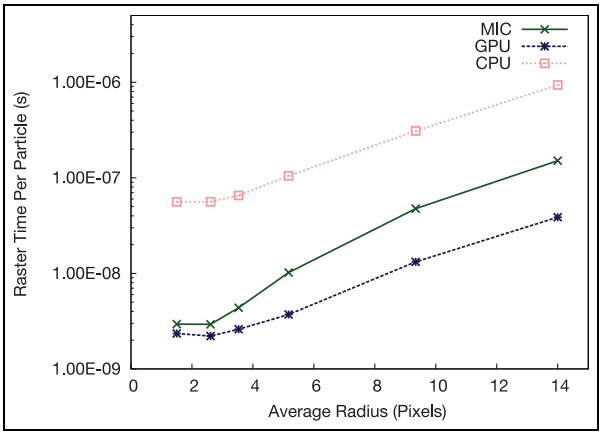

Figure 15. Xeon Phi versus NVIDIA K20X versus Xeon CPU: comparison of the rasterization per-particle times.

shown in Figure 15 is mostly attributed to the combination of the colorize and transform/filter kernels on the GPU. This means that for a large data set where a significant portion of particles are inactive (i.e. off screen), as is the case for the tests with larger average radius, the kernel ends before processing these particles. To retain automatic vectorization of the transform kernel on the MIC, the colorize kernel is run separately, and so all particles must be reread and the active status tested, causing the colorize kernel to be dependent on the total number of particles as opposed to solely the number of active particles as is the case on the GPU.

Figure 16 shows performance comparison for the rendering phase. For a larger average radius, above two to three pixels, the MIC outperforms the GPU for rendering. This is due to the MIC algorithm being more suited to scenarios where a particle may affect a large portion of the image, as particles can be rendered by multiple threads when affecting multiple tiles. For the GPU, this is not possible as each tile is rendered by different CUDA blocks, therefore when a particle affects more than one tile it must be transferred back to the host for rendering (see Rivi et al., 2014, for more on this).

The GPU performs very well in the lower radii range due to the fact that the large majority of particles are processed by the CUDA thread blocks and only a few of them are left to the CPU. Furthermore, a specific one particle per thread approach is used with point-like particles which is ideal for the hardware. The MIC performance decreases in the case where a considerable portion of the image is unused (e.g. with a point of view far from the computational box center, as in Figure 1 *top left*). The current decomposition method does not effectively load-balance this distribution of particles and requires improvements to account for such situations.

In most cases both the MIC coprocessor and GPU accelerator outperform, or perform very similar to, the Xeon 8-core comparison CPU.

Discussion

The results gathered so far demonstrate that in some areas of code the MIC architecture excels well beyond the host capabilities, although in others a fair amount of modification is necessary to gain acceptable performance levels, which is expected of a highly parallel architecture such as this. It is recommended to make extensive use of the optimization guides provided by Intel, and in order to achieve best performance, rely not only on automatic vectorization but manual insertion of intrinsics also. Memory management is also key to performance; use of MPI-based offload is shown to mitigate some overheads, similar to others' experience (e.g. Borovska and Ivanova, 2014). Issues regarding scalable memory allocation, which may not be apparent with an identical implementation on a Xeon CPU, can be greatly improved by use of a thread-aware allocation mechanism as demonstrated in Section 3.2.1.

This work has focused purely on the offload model of exploiting the Xeon Phi, however, the optimizations performed here can be effective not only to offload processing but also to native processing. In observation of the future plans of Intel, in particular the second generation Xeon Phi product codenamed *Knights Landing* (Hazra, 2013), we believe greater performance will be seen moving to a native model and utilizing the device as a processor in its own right. The improvements to the architecture in the second generation will remove many barriers to performance; the ability to function as a standalone processor with direct access to large memory will remove costly PCIe-based data transfer, and the move to atom-based cores will allow for more advanced architectural features to be exploited, for

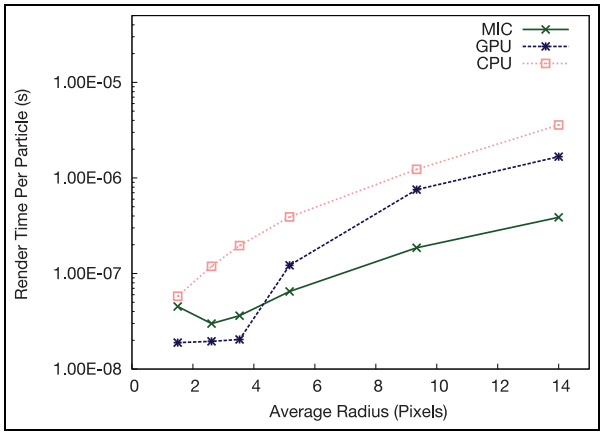

Figure 16. Xeon Phi versus NVIDIA K20X versus Xeon CPU: comparison of the rendering per-particle times.

example out of order execution, providing improved serial performance.

The comparison of GPU and Xeon Phi shows that our GPU implementation currently outperforms the Xeon Phi in most areas. However, Xeon Phi results are in some scenarios close or even better than those of the GPU, as, for instance, when particle distributions with a large average radius are processed. The GPU architecture, in fact, is ill-suited to the memory access patterns of particles that affect large fractions of the image. Consequently, such particles have to be moved back to and processed by the CPU, with a strong performance penalty. This penalty is avoided by the Xeon Phi, whose computing capabilities can be exploited for rendering particles of any size. It should be noted that the main barrier to performance for Xeon Phi is memory allocation. In each of the tests for MIC, memory allocation makes up at least half of overall time, and in the case where a Xeon Phi could allocate memory as fast as a regular Xeon, the Xeon Phi would outperform the GPU in most of our tests (allocation ratio shown in breakdown of Figure 8).

Conclusions

In this article, we document our process of porting and optimizing Splotch, a visualization algorithm targeting large-scale astrophysical data, to the Xeon Phi coprocessor. We explain the background of Splotch, and the efforts to achieve performance in an offloading model on the Xeon Phi by modifying our original OpenMP and MPI parallelized code to exploit Intel's LEO. We discuss in detail the optimizations performed through use of environment variables, thread affinity, vectorization (automatic and manual), and memory management. We profile offloaded code using a small wrapper around the Performance API for directly measuring hardware events and share our experiences throughout the porting and optimizing process.

We run tests with multiple Xeon Phi versus a dual socket Xeon 2670 and record strong performance gains in individual kernels, that is $9\times$ dual socket performance for the transform and color kernels. The algorithm as a whole proves comparable to the dual socket Xeon for data sets that fit within device memory and a single socket for larger data sets. We contrast our achieved performance against that of our CUDA implementation on NVIDIA K20X graphics processors and find that our GPU implementation currently outperforms the MIC implementation, albeit only marginally in some areas, with the MIC being particularly suited to the rendering stage; both the coprocessor and accelerator outperform a single 8 core Xeon 2670 in all but one cases.

We also compare the experience of porting Splotch to Xeon Phi and GPUs. The MIC architecture, more "traditional" than that of the GPU, does not require extensive changes in the core algorithms, hence the design phase is simplified. Furthermore, the refactoring for the Xeon Phi is made easier by the ability to program with regular tools and familiar paradigms (e.g. OpenMP, MPI). On the other hand, performance tuning is, in most cases, highly demanding. Overall, the resulting time and effort needed to enable Splotch to run efficiently on a Xeon Phi is comparable to that needed for the GPU. With the introduction of further models of Xeon Phi, that is Knights Landing, this may change in the near future. However, it is likely developments in GPU architecture will also start to reduce critical barriers to performance (e.g. NVLink), and further developments may simplify code development for heterogenous systems exploiting both architectures (e.g. OpenACC), we believe both versions can also benefit from further development to reach optimal performance.

The Splotch code is now ready to effectively exploit new supercomputing systems making use of Xeon Phi devices. We intend to continue maintaining and developing the code and updating to exploit new hardware features when possible to continue to allow Splotch to utilize heterogenous systems in the best possible manner.

Acknowledgements

We would like to acknowledge CSCS-ETHZ, Lugano, Switzerland, for hosting author Timothy Dykes as part of their internship program, Klaus Dolag (University Observatory Munich) and Martin Reinecke (MPA Garching) for discussions and data for performance analysis and the Institute of Cosmology and Gravitation at the University of Portsmouth for providing access to, and assisting with use of, their SCIAMA cluster for testing and analysis.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) received no financial support for the research, authorship, and/or publication of this article.

Notes

1. https://visit.llnl.gov
2. http://www.cilkplus.org/
3. The Gadget code: http://www.mpa-garching.mpg.de/gadget/
4. https://github.com/splotchviz/splotch/wiki
5. http://www.hdfgroup.org/HDF5/
6. For example: https://software.intel.com/en-us/mic-develo per
7. https://github.com/TimDykes/mic_papi_wrapper
8. For more on thread affinity, see: https://software.intel.-com/en-us/articles/openmp-thread-affinity-control

References

Borovska P and Ivanova D (2014). Code optimization and scaling of the astrophysics software Gadget on Intel Xeon Phi. PRACE White Paper. Available at: http://www.prace-ri.eu/evaluation-intel-mic (accessed 2 January 2015).

Chrysos G. Intel Xeon Phi coprocessor (codename Knights Corner). In: *24th Annual IEEE Hot Chips Symposium*, California, USA, 27–29 August 2012.

Dolag K, Reinecke M, Gheller C, et al. (2008) Splotch: visualizing cosmological simulations. *New Journal of Physics* 10(12): 125006.

Elena A and Rungger I (2014) Enabling Smeagol on Xeon Phi: Lessons Learned. *Partnership for Advanced Computing in Europe (PRACE)* 134, Available at: http://www.prace-ri.eu/evaluation-intel-mic.

Fang J, Sips H, Zhang L, et al. (2014) Test-driving Intel Xeon Phi. In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, Dublin, Ireland, 22–26 2014. New York: ACM.

Gaburov E and Cavecchi Y (2014) Xeon Phi meets astrophysical fluid dynamics. Partnership for Advanced Computing in Europe (PRACE) 132, Available at: http://www.prace ri.eu/evaluation-intel-mic

Gloger W (2006) Available at: ptmalloc3. http://www.mall oc.de/en/ (accessed 10 December 2014).

Hazra R. (2013) Driving industrial innovation on the path to exascale: from vision to reality. In: *International Supercomputing Conference*, Leipzig, Germany, 16–20 June 2013.

Heinecke A, Breuer A, Rettenberger S, et al. (2014) Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, USA, 16–21 November 2014, pp. 3–14. New Jersey, USA: IEEE Press Piscataway.

Intel (2012a) How to use huge pages to improve application performance on Intel Xeon Phi coprocessor. Available at: https://software.intel.com/sites/default/files/Large_pages_mic_0.pdf (accessed 8 December 2014).

Intel (2012b) Optimization and performance tuning for Intel Xeon Phi coprocessors, Part 1: Optimization essentials. Available at: https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-1-optimization (accessed 10 December 2014).

Intel (2012c) Intel Xeon Phi coprocessor instruction set architecture reference manual. https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf (accessed 2 January 2015).

Intel (2012d) A guide to auto-vectorization with Intel C++ compilers. Available at: https://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers (accessed 18 December 2014).

Intel (2012e) Optimization and performance tuning for Intel Xeon Phi coprocessors, Part 2: Understanding and using hardware events. Available at: https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding (accessed 10 December 2014).

Intel (2012f) Xeon Phi coprocessor system software developers guide. https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-system-software-developers-guide, 2012. (accessed 18 December 2014).

Intel (2013a) Intel many-integrated-core architecture - advanced. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html (accessed 10 December 2014).

Intel (2013b) Effective use of the Intel compiler's offload features. Available at: https://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features (accessed 10 December 2014).

Intel (2014) Controlling memory consumption with Intel Threading Building Blocks (Intel TBB) scalable allocator. Available at: https://software.intel.com/en-us/articles/controlling-memory-consumption-with-intel-threading-building-blocks-intel-tbb-scalable (accessed January 2nd 2015)

Intel (2016) Intel Xeon Phi coprocessor applications and solutions catalogue. Available at: https://software.intel.com/sites/default/files/managed/eb/f7/intel-xeon-phi-catalog-jan-2016.pdf (accessed 8 June 2016).

Jin Z, Krokos M, Rivi M, et al. (2010) High-Performance astrophysical visualization using Splotch. *Procedia Computer Science* 1(1): 1775–1784.

Mucci PJ, Browne S, Deane C, et al. (1999) Papi: A portable interface to hardware performance counters.

Reid F and Bethune I (2014) Optimising CP2 K for the Intel Xeon Phi. PRACE White Paper. Avaialble at: http://www.prace-ri.eu/evaluation-intel-mic

Rivi M, Gheller C, Dykes T, et al. (2014) GPU accelerated particle visualisation with Splotch. *Astronomy and Computing* 5: 9–18.

Top500org (2015) Top500 list November 2015. Available at: http://www.top500.org/lists/2015/11/ (accessed 10 January 2016).