Evaluating multi-core and many-core architectures through accelerating the three-dimensional Lax–Wendroff correction stencil

Yang You, Haohuan Fu, Shuaiwen Leon Song, Maryam Mehri Dehnavi, Lin Gan, Xiaomeng Huang and Guangwen Yang

International Journal of High Performance Computing Applications published online 5 March 2014
DOI: 10.1177/1094342014524807

Original Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
1–18

DOI: 10.1177/1094342014524807
hpc.sagepub.com
SAGE

Evaluating multi-core and many-core architectures through accelerating the three-dimensional Lax–Wendroff correction stencil

Yang You[1], Haohuan Fu[1,4], Shuaiwen Leon Song[2], Maryam Mehri Dehnavi[3], Lin Gan[1,4], Xiaomeng Huang[1,4] and Guangwen Yang[1,4]

Abstract

Wave propagation forward modeling is a widely used computational method in oil and gas exploration. The iterative stencil loops in such problems have broad applications in scientific computing. However, executing such loops can be highly time-consuming, which greatly limits their performance and power efficiency. In this paper, we accelerate the forward-modeling technique on the latest multi-core and many-core architectures such as Intel® Sandy Bridge CPUs, NVIDIA Fermi C2070 GPUs, NVIDIA Kepler K20× GPUs, and the Intel® Xeon Phi co-processor. For the GPU platforms, we propose two parallel strategies to explore the performance optimization opportunities for our stencil kernels. For Sandy Bridge CPUs and MIC, we also employ various optimization techniques in order to achieve the best performance. Although our stencil with 114 component variables poses several great challenges for performance optimization, and the low stencil ratio between computation and memory access is too inefficient to fully take advantage of our evaluated architectures, we manage to achieve performance efficiencies ranging from 4.730% to 20.02% of the theoretical peak. We also conduct cross-platform performance and power analysis (focusing on Kepler GPU and MIC) and the results could serve as insights for users selecting the most suitable accelerators for their targeted applications.

Keywords

Complex stencil, 3D wave forward modeling, Kepler GPU, Intel Xeon Phi, optimization techniques, performance power analysis

1 Introduction

The forward modeling of wave propagation is a widely used computational method in oil and gas exploration. Its iterative stencil loops (Li and Song, 2004) also have broad applications in image processing, data mining and various physical simulations (Meng and Skadron, 2009). However, the time-consuming iterative stencil loops greatly limit the application's performance efficiency.

In the past decade, with the design constraints on heat dissipation and power consumption, the development trend of processor architectures has moved from increasing the clock rate to increasing the number of identical cores (Mudge, 2001). While programmers in the past could depend on the ready-made performance improvements from faster clock speeds, nowadays they have to face the challenges of scaling applications over hundreds of cores within a single chip (Satish et al., 2012).

Graphics Processing Units (GPUs) have become a popular accelerator for general purpose computation since the introduction of the Compute Unified Device Architecture (CUDA) programming model by

[1]Department of Computer Science and Technology, Tsinghua University, China
[2]Pacific Northwest National Laboratory, USA
[3]Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, USA
[4]Ministry of Education Key Laboratory for Earth System Modeling, Tsinghua University, China

Corresponding author:
Yang You, FIT 3-126, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China.
Email: you-y12@mails.tsinghua.edu.cn

NVIDIA in 2006. CUDA has been used in the fields of machine learning (Raina et al., 2009), oil exploration (Liu et al., 2009), genomics (Manavski and Valle, 2008), finance (Surkov, 2010) etc. due to its superior computing capability and high parallelism. Recently, Intel introduced the Many Integrated Core (MIC) architecture, which claims to provide more than 1 Tflop double-precision performance (http://www.xbitlabs.com/). In addition, the $\times 86$ core architecture of MIC provides a friendly environment with high programmability and easy software porting.

In this paper, we present various efficient strategies based on four evaluated platforms (Sandy Bridge CPUs, Fermi GPUs, Kepler GPUs, and the Intel® Xeon Phi) to accelerate the complex iterative stencil loops, which dominate the execution time of our three-dimensional (3D) wave forward-modeling application.

Each accelerator architecture has its own unique advantages over the others for certain types of applications (Asano et al., 2009; Clapp et al., 2010; Nickolls and Dally, 2010). Therefore, choosing the best accelerator for a specific application becomes a key issue in improving the application's performance and power efficiency. In order to provide support for such selection, we also conduct cross-platform performance and power analysis (focusing on Kepler GPUs and the Intel® MIC) to provide insight to help users select the most suitable accelerators for their applications (see Section 8.3).

The contributions of this work include:

(1) The design of "one-thread-one-point" and "one-thread-one-line" schemes to maximize the performance and resource utilization of our stencil kernel on Fermi and Kepler GPUs. We then compare the performance achieved by these two schemes on Fermi and Kepler GPUs respectively, and suggest what optimization techniques should be applied when the architecture changes from Fermi to Kepler.
(2) The exploration of a series of optimization techniques for our stencil kernel on Sandy Bridge CPUs and the Intel® MIC. We also explain how the architectural differences between Sandy Bridge and MIC relate to their significant performance differences.
(3) A detailed cross-platform comparison analysis between Kepler GPUs and MIC, which can provide insight to help users who need to choose the best accelerators for their specific applications.

The rest of the paper is organized as follows: In Section 2, we provide an overview of the application's background and related work; a detailed Lax–Wendroff Correction (LWC) stencil analysis is given in Section 3; a comprehensive description and analysis of the evaluated architectures used in this paper is provided in Section 4; optimization techniques on two types of GPUs, Sandy Bridge, and MIC are demonstrated in Sections 5 and 6; a summary of the experimental results is give in Section 7; detailed cross-platform performance/power comparison analysis is provided in Section 8; and conclusions for this work are discussed in Section 9.

2 Background and related work

Explicit finite-difference (EFD) is a classical strategy for solving the wave-propagation problem. One of the most efficient and commonly used methods for EFD is the LWC, which was proposed by Lax and Wendroff in 1964. During recent years, LWC has become a mature approach and its efficiency has been improved significantly by various works from Dablain (1986), Sei and Symes (1995), Blanch and Robertsson (2007) and other scholars. For instance, in Dablain's work, published in 1986, high-order approximation for derivatives was proven to be very efficient because of its low time cost and lesser memory requirements, and such methods have been widely used ever since.

Although the performance of numerical algorithms has improved significantly during the past 30 years, they still cannot satisfy the growing demands of industries such as oil and gas exploration. Therefore, accelerating these algorithms through utilizing the current most advanced parallel architectures such as GPGPU and Intel® Xeon Phi has become an urgent task.

In 2009, Paulius Micikevicius accelerated a simple finite-difference algorithm using a shared memory data reuse approach on a Tesla-10 GPU. The results showed that a single GPU could achieve an order of magnitude speedup over a four-core CPU. Thousands of threads were utilized to traverse the 3D volume slice-by-slice in order to maximize the data reuse. Okamoto et al. (2010) and Michéa and Komatitsch (2010) proposed similar techniques and achieved $20\times$ to $60\times$ speedups over a single CPU. They also achieved higher overall cluster performance using GPUs as accelerators in a HPC environment. Additionally, there have been other works (Unat et al., 2011; Zhang and Mueller, 2012) focusing on automatically generating GPU-stencil code, which is able to deliver competitive performance against painstakingly handcrafted work.

However, in order to further accelerate the LWC stencil using modern accelerators (i.e. GPUs and MIC) or accelerator-based large-scale parallel systems, we still face some challenges:

(1) In LWC, a higher order of difference can be efficient because it requires a smaller data size and less memory cost for simulating the same wave field. However, a high order of difference can

make the stencil algorithm more complicated, which is undesirable for programmability. Its performance could also be constrained by the limited number of registers on accelerators. In order to maintain balance between algorithm efficiency and complexity, we need to choose the proper order of difference for accelerating LWC on modern heterogeneous systems (shown in Section 3.1).

(2) In addition to the single-variable partial derivatives in LWC such as $\frac{\partial^2 U}{\partial y^2}$ and $\frac{\partial^2 U}{\partial z^2}$ (solved using the Micikevicius (2009) method in this paper), we also have to handle the multiple-variable partial derivatives such as $\frac{\partial^2 V}{\partial x \partial y}$ and $\frac{\partial^2 W}{\partial x \partial z}$, which brings a further challenge for the limited number of registers for each thread on accelerators such as GPUs (shown in Section 5.2).

There have also been a few works on using Intel® MIC to accelerate parallel applications such as sorting (Satish et al., 2010), data mining (Heinecke et al., 2012), and ray tracing (Benthin et al., 2012). However, they are based on the previous Knights Ferry MIC (KNF), rather than on the more advanced Knights Corner MIC (KNC). There have been several important improvements (Duran and Klemm, 2012; http://www.hpcwire.com/hpcwire/) to the KNC architecture, including: (a) the increase in GDDR5 memory size from 2 GB to 8 GB; (b) the increase in the number of cores from 32 to 61; (c) the increase in the theoretical bandwidth of the memory from 125 GB/s to 320 GB/s; and (d) the processing power of single/double precision has also improved significantly. Therefore, optimizing complex iterative stencil loops of LWC on the KNC architecture is more meaningful, especially when exploring the architectural differences between the host Sandy Bridge CPUs and the device Xeon Phi accelerators.

3 LWC stencil analysis

In order to balance the efficiency and cost of the optimization as well as address the challenge (1) described in the previous section, we chose to use the central-difference scheme (four-order spatial accuracy: $O(\Delta t^2, \Delta x^4, \Delta y^4, \Delta z^4)$) to speedup LWC on modern system accelerators (terms explained in Table 1).

3.1 Mathematical derivation

We use U, V and W to denote the three displacement components in the x-, y- and z-directions. U, V and W are the functions that depend on x, y, z and t. The wave equations in isotropic media are shown as equations (1)–(3).

Table 1. Explanation of the mathematical terms used in this paper.

Term	Explanation
T	The whole physical simulation time
Δt	Temporal step
$\Delta x, \Delta y, \Delta z$	Spatial steps in the x-, y-, z-directions
U, V and W	Displacement components in the x-, y-, z-directions
ρ, λ, μ	Coefficients
h_1, h_2, h_3	Coefficients

$$\rho \frac{\partial^2 U}{\partial t^2} = (\lambda + 2\mu)\frac{\partial^2 U}{\partial x^2} + \mu \frac{\partial^2 U}{\partial y^2} + \mu \frac{\partial^2 U}{\partial z^2} + (\lambda + \mu)\frac{\partial^2 V}{\partial x \partial y} + (\lambda + \mu)\frac{\partial^2 W}{\partial x \partial z} \quad (1)$$

$$\rho \frac{\partial^2 V}{\partial t^2} = (\lambda + \mu)\frac{\partial^2 U}{\partial x \partial y} + \mu \frac{\partial^2 V}{\partial x^2} + (\lambda + 2\mu)\frac{\partial^2 V}{\partial y^2} + \mu \frac{\partial^2 V}{\partial z^2} + (\lambda + \mu)\frac{\partial^2 W}{\partial y \partial z} \quad (2)$$

$$\rho \frac{\partial^2 W}{\partial t^2} = (\lambda + \mu)\frac{\partial^2 U}{\partial x \partial z} + (\lambda + \mu)\frac{\partial^2 V}{\partial y \partial z} + \mu \frac{\partial^2 W}{\partial x^2} + \mu \frac{\partial^2 W}{\partial y^2} + (\lambda + 2\mu)\frac{\partial^2 W}{\partial z^2} \quad (3)$$

For instance, in order to get the expression $\frac{\partial^2 U}{\partial t^2}$ in equation (1), we need to first compute the spatial partial derivatives $\left(\frac{\partial^2 U}{\partial x^2}, \frac{\partial^2 U}{\partial y^2}, \frac{\partial^2 U}{\partial z^2}, \frac{\partial^2 U}{\partial x \partial y} \text{and} \frac{\partial^2 U}{\partial x \partial z}\right)$ through the Taylor expansions (Dablain, 1986). We can then obtain $\frac{\partial^2 U}{\partial t^2}$ by adding the spatial partial derivatives to equation (1), where λ and μ are the coefficients. $\frac{\partial^2 V}{\partial t^2}$ and $\frac{\partial^2 W}{\partial t^2}$ can also be obtained using the same procedure.

The deductions shown above are based on spatial variables. We can also get the temporal variables (i.e. U_{t+1} and U_{t-1} at time t) using Taylor expansions. In this way, we are able to update U_{t+1}, V_{t+1},W_{t+1} by the values of their previous moments ($t-1$ and t). This process is shown in equations (4)–(6). We then loop them for $\frac{T}{\Delta t}$ times, where T and Δt represent the whole physical simulation time and temporal step, respectively (Table 1).

$$U_{t+1} = 2U_t - U_{t-1} + \frac{\partial^2 U}{\partial t^2}(\Delta t)^2 + O((\Delta t)^3) \quad (4)$$

$$V_{t+1} = 2V_t - V_{t-1} + \frac{\partial^2 V}{\partial t^2}(\Delta t)^2 + O((\Delta t)^3) \quad (5)$$

Algorithm 1 The LWC algorithm

Step 0: Start
Step 1: $Times \leftarrow 0$, set all the elements of U, V and W as zero
Step 2: Use Taylor expansions to calculate the spatial partial derivatives
Step 3: Get $\frac{\partial^2 U}{\partial t^2}$, $\frac{\partial^2 V}{\partial t^2}$ and $\frac{\partial^2 W}{\partial t^2}$ through substituting the spatial partial derivatives into equations (1)–(3)
Step 4: If the point in wave source, $\frac{\partial^2 W}{\partial t^2} \leftarrow \frac{\partial^2 W}{\partial t^2} + \frac{ffs}{\rho h_1 h_2 h_3}$
Step 5: Update U, V and W through substituting $\frac{\partial^2 U}{\partial t^2}$, $\frac{\partial^2 V}{\partial t^2}$ and $\frac{\partial^2 W}{\partial t^2}$ into equations (4)–(6)
Step 6: $Times \leftarrow times + 1$
Step 7: If $times \leq \frac{T}{\Delta t}$, go Step 2
Step 8: End

$$W_{t+1} = 2W_t - W_{t-1} + \frac{\partial^2 W}{\partial t^2}(\Delta t)^2 + O((\Delta t)^3) \quad (6)$$

3.2 Algorithm overview

Algorithm 1 represents our numerical algorithm; the flow chart is shown in Figure 1. Refer to our source code at https://github.com/You1991/ManyCoreForwardModeling for the implementation details.

3.3 Ratio of computation to memory access

We define the Ratio of Computation to Memory Access (RCMA) in equation (7).

The LWC stencil (shown in Figure 2) is a complex 3D kernel, which dominates the computation time of our forward-modeling algorithm. As mentioned above, to compute all three components (U, V and W) of one central point, 36 neighboring points of this central point (shown in Figure 2) are required to be accessed. Therefore, plus one extra point, to update a single point at each step, we need to read 114 elements (38×3) and 14 coefficients in total, with 228 floating point operations. For the best case, where we have the perfect data reuse (U, V and W are only fetched once), the single-precision RCMA is 19 (equation (8)). However, for the worst case, where we have to read U, V and W for each point, the single-precision RCMA is reduced to 0.5 (equation (9)). Therefore, the range of single-precision RCMA for the LWC stencil is between 0.5 and 19. Similarly, the range of double-precision RCMA for LWC stencil is between 0.25 and 9.5.

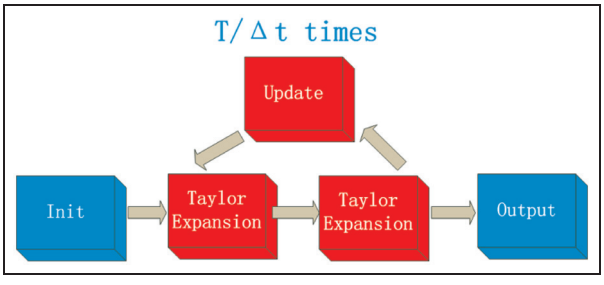

Figure 1. Flow chart of the algorithm. The middle loops (marked in red) dominate the execution time for acceleration.

$$RCMA = \frac{num\_flops}{num\_bytes} \quad (7)$$

$$\frac{228 \times data\_size}{3 \times data\_size \times 4\text{B}} = 19 \quad (8)$$

$$\frac{228 \times data\_size}{114 \times data\_size \times 4\text{B}} = 0.5 \quad (9)$$

4 Architectures for accelerating the forward-modeling algorithm

We have accelerated our algorithm on several of the currently most advanced multi-core and many-core architectures including the NVIDIA Fermi C2070 GPU, the NVIDIA Kepler 20× GPU, Intel® Sandy Bridge CPUs and the Intel® Xeon Phi co-processor. The important parameters of these architectures are listed in Table 2.

4.1 Memory hierarchy

4.1.1 The ×86 cores. The Sandy Bridge architecture used for the comparison analysis in this paper is shown in Figure 3. It is composed of 2 Xeon E5-2560 CPU sockets and 16 ×86 cores (8 cores per socket) in total. Each core has a 64 kB private L1 cache and a 512 kB private L2 cache. Additionally, each CPU is equipped with an extra 20 MB coherent L3 cache which is shared by all 8 cores. These two 20 MB L3 caches are connected by an 8 GT/s Intel® Quickpath Interconnect, providing real-time data exchanges between different cores. In addition to the limited high-speed three-level cache, each CPU possesses 4 memory channels, providing a 68 GB/s bandwidth (tested by STREAM benchmark) for the whole system.

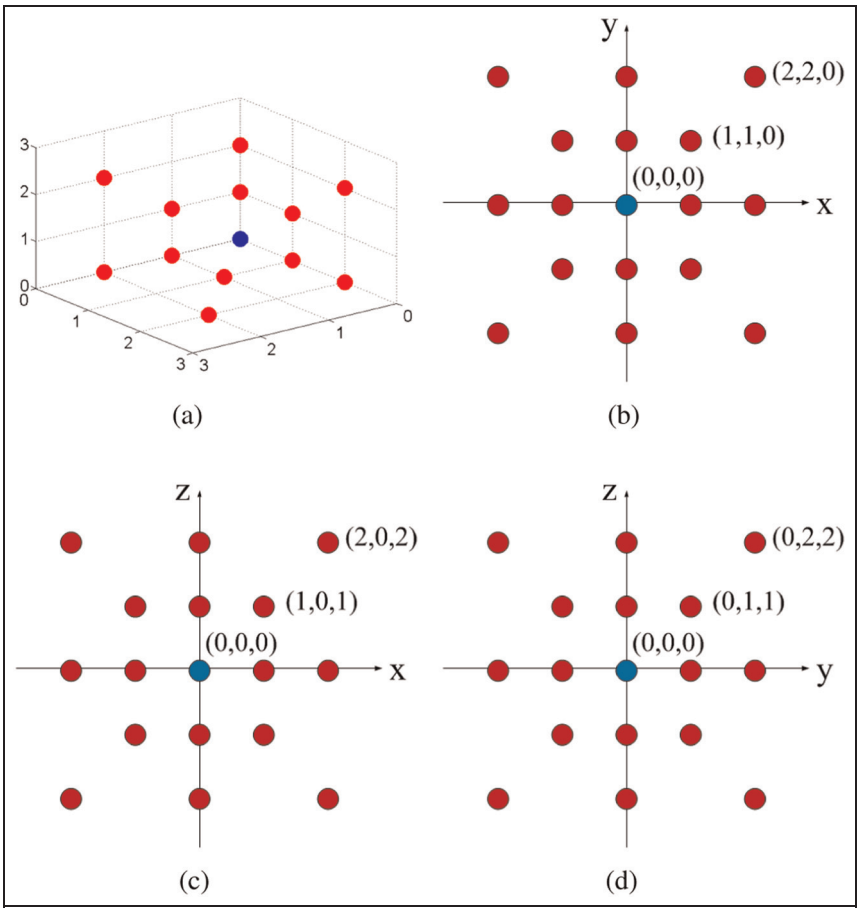

Figure 2. Stencil: each point in the graph depends on its 36 neighboring points. These 36 points are evenly distributed in the XOY, YOZ and XOZ planes. There are 37 points in total after counting the central point.

Table 2. Related parameters of architectures.

Architecture	Sandy Bridge	KNC	Fermi	Kepler
Frequency (GHz)	2.00	1.09	1.50	0.73
Peak performance double (Gflops)	256	1010	515	1320
Peak performance single (Gflops)	512	2020	1030	3950
L1 Cache (kB)	32/core	32/core	64/SM	64/SM
L2 Cache (kB)	256/core	512/core	768/card	1536/card
L3 Cache (MB)	20/socket	0	0	0
Coherent cache	L3	L2	L2	L2
Memory type	DDR3	GDDR5	GDDR5	GDDR5
Memory size (GB)	128	8	6	6
Theoretical memory bandwidth (GB/s)	100	352	192	250
Measured memory bandwidth (GB/s)	68	159	97	188
Computation bandwidth ratio single (Gflops/GB)	7.52	12.70	10.62	21.01
Computation bandwidth ratio double (Gflops/GB)	3.76	6.35	5.31	7.02

Similar to the Sandy Bridge architecture, our Intel® KNC (shown in Figure 4) also consists of many $\times 86$ cores, Quickpath Interconnect (5.5 GT/s) and several memory controllers. However, there are several distinct differences between these two architectures. As for the number of cores, there is a significant improvement from KNC MIC (61 in total, 60 for computation and one for OS management) over Sandy Bridge. For memory bandwidth, KNC MIC can provide a 352 GB/s theoretical bandwidth (5.5 GTransfers/s $\times$ 16 channels $\times$ 4 B/Transfer) and 159 GB/s practical bandwidth (STREAM benchmark), which are almost three times

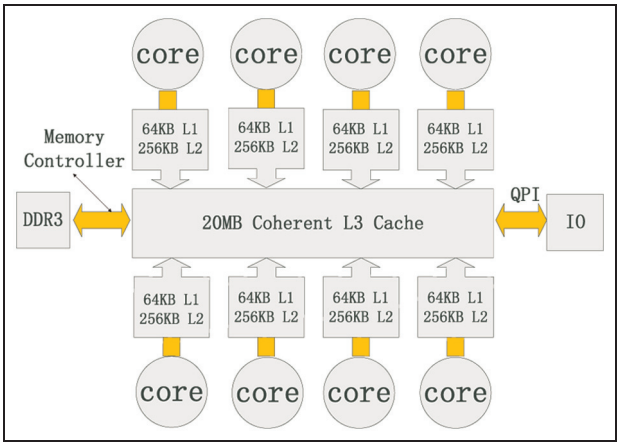

Figure 3. Sandy Bridge architecture. Each CPU is composed of 8 ×86 cores and a 20 MB coherent L3 cache. Each core has a 32 kB L1 data cache, a 32 kB L1 instruction cache, and a 256 kB L2 cache.

that for Sandy Bridge. Moreover, MIC does not have an L3 cache but has a 31 MB coherent L2 cache (512 kB for each core).

4.1.2 GPGPU. Each Fermi GPU card is equipped with 14 streaming multi-processors (SM). In each SM (Figure 5), there is a 64 kB high-speed on-chip buffer that can be configured as either 48 kB of shared memory + 16 kB L1 cache (default), or the other way round. Each SM also has a 12 kB read-only texture cache. Additionally, there is a unified 768 kB L2 cache that is shared among all the SMs, and a 6 GB on-board GDDR5 memory with a bandwidth of 192 GB/s.

There are some notable architectural improvements in terms of memory from Kepler over Fermi (shown in Figure 6): (1) users have an additional option to divide the 64 kB on-chip buffer into 32 + 32 kB; (2) the size of the read-only texture cache increases from 12 kB to 48 kB; (3) the size of the L2 cache doubles; and (4) the bandwidth of global memory grows up to 250 GB/s.

4.2 Processing power

The growing concern of power dissipation (Mudge, 2001) has moved the focus of modern processors from the increasing clock rate to increasing parallelism to improve peak performance without drastic power increment. All our experimental architectures employ two-level parallelisms: task and data parallelism.

For Sandy Bridge and Xeon Phi, the task parallelism is achieved by utilizing multiple or many hardware threads. The data parallelism benefits from an on-core Vector Processing Unit (VPU) and Single Instruction Multiple Data (SIMD). In each CPU core, the 256-bit instruction can process 4 double-precision operations or 8 single-precision operations at a time. In each MIC core, the 512-bit instruction doubles the data parallelism.

For Fermi and Kepler, the task parallelism comes from the independent warps that are executed by different SMs. In each warp, the data-parallelism is achieved by the computations performed by the different CUDA cores within the SM. Although the number of SMs does not increase from Fermi to Kepler, the 192 cores per SM in Kepler is more powerful than the 32 cores per SM in Fermi.

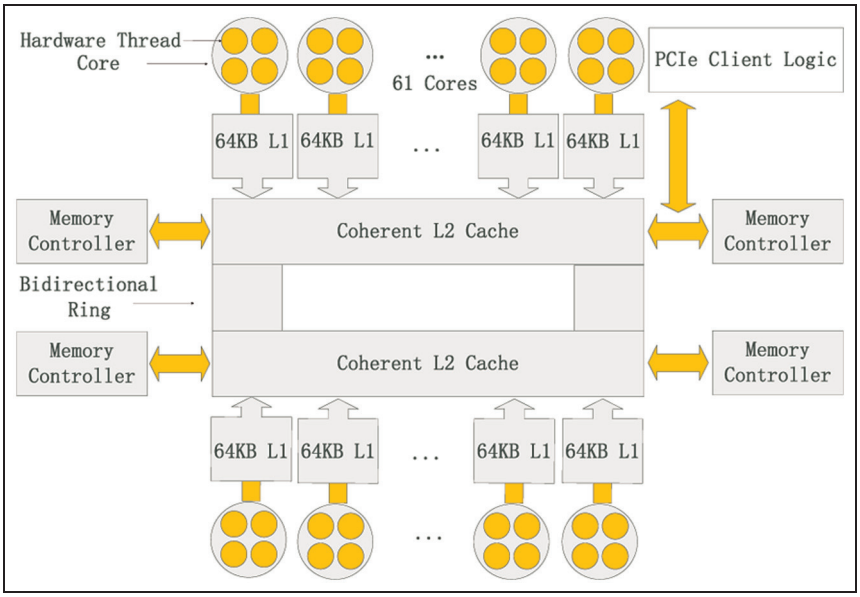

Figure 4. KNC Architecture. Each MIC consists of 61 cores and a 31 MB coherent L2 cache. Each core has a 32 kB L1 data cache and a 32 kB L1 instruction cache (http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner).

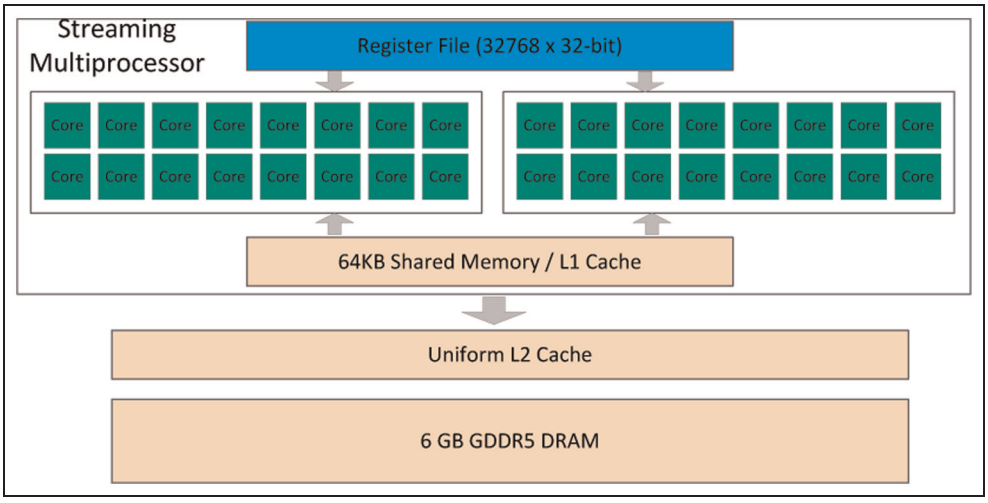

Figure 5. NVIDIA Fermi architecture. Each streaming multi-processor contains 32 CUDA cores, a 64 kB on-chip L1 cache + shared memory and a 768 kB unified L2 cache.

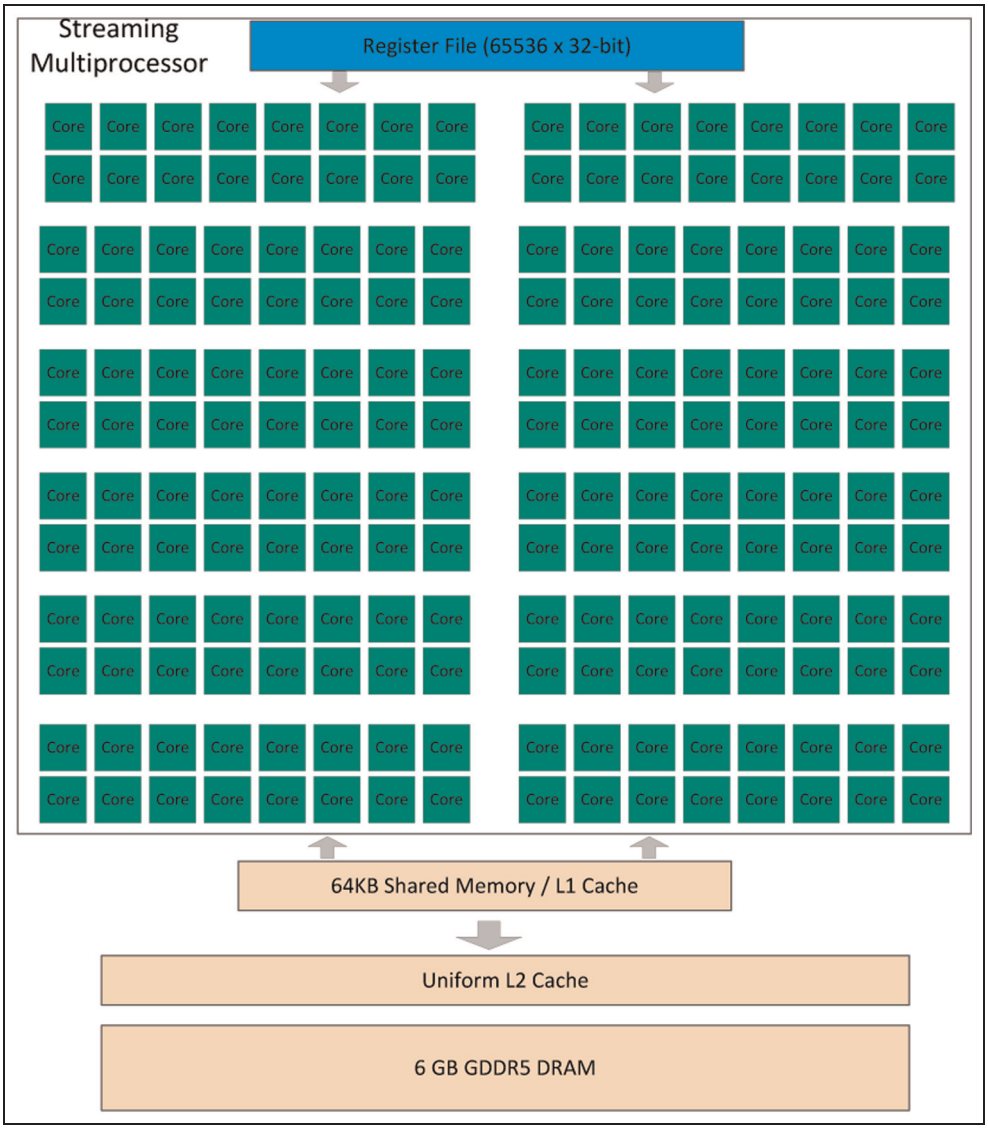

Figure 6. NVIDIA Kepler architecture. Each streaming multi-processor contains 192 cores, a 64 kB on-chip L1 cache + shared memory and a 1536 kB unified L2 cache.

In order to get a satisfactory performance, fully utilizing the two-level parallelism, and improving the occupancy rate of the computing resources are the crucial issues.

$$RCMB = \frac{max\_performance}{max\_measured\_bandwidth} \quad (10)$$

4.3 Ratio of computation to memory bandwidth

Similar to the RCMA (equation (7)), the Ratio of Computation to Memory Bandwidth (RCMB) of a specific architecture is defined in equation (10). Compared to the RCMBs of the evaluated architectures (shown in Table 2), the algorithmic RCMA (3.3) is much lower, even in the worst case. Take the single-precision case for example, the RCMB of Kepler is 21.01 while the RCMA of LWC is 0.5, which means the limited memory bandwidth may not match the high processing power required for accelerating the LWC stencil. Therefore, improving data reuse in cache and shared memory is necessary in order to reduce the impact of the constrained bandwidth.

5 Optimizations on the GPUs

We propose two schemes for accelerating the stencil kernel on the GPUs: Scheme (1) one thread handles one point; and Scheme (2) one thread handles one line, which means we put the 2D neighboring slices in shared memory and each thread traverses the 3D stencil slice-by-slice on its own line. In Scheme (2), the slices are allocated into shared memory at the beginning, and each point can reuse the neighboring points in the process of computation. Scheme (1) can provide extremely high parallelism while Scheme (2) can maximize shared memory data reuse and reduce off-chip traffic.

5.1 Optimization scheme (1): One thread handles one point

In order to maximize parallelism and make full use of millions of lightweight threads on the GPUs, we propose this one-thread-one-point scheme. The challenges of this technique are: (a) getting the corresponding relation between the thread-ID and stencil coordination; and (b) calculating the boundary condition correctly. Both of them increase the complexity of programming when handling non-trivial stencil computation. The details of Scheme (1) are shown in Algorithm 2.

Algorithm 2 The one-thread-one-point scheme

Step 0: Start
Step 1: Calculate the threadId
Step 2: Set all the elements as zero
Step 3:
$x \leftarrow threadId$ DIV ($dimy$ MUL $dimz$)
$y \leftarrow$ ($threadId$ MOD ($dimy$ MUL $dimz$)) DIV $dimz$
$z \leftarrow threadId$ MOD $dimz$
Step 4: If x in bounds and y in bounds and z in bounds, perform Algorithm 1 (each point is only controlled by one thread)
Step 5: Synchronization
Step 6: End

5.2 Optimization scheme (2): One thread handles one line

As mentioned in Section 4.3, improving data reuse in the on-chip buffer is indispensable since our stencil is very likely constrained by the limited memory bandwidth. Thus, using the two-dimensional (2D) slice shared memory scheme may be a good choice. However, our stencil is much more complex (shown in Figure 2) compared with previous work because we not only need to compute the single-variable partial derivatives such as $\frac{\partial^2 U}{\partial y^2}$ and $\frac{\partial^2 U}{\partial z^2}$, but also have to handle the multiple-variable partial derivatives such as $\frac{\partial^2 V}{\partial x \partial y}$ and $\frac{\partial^2 W}{\partial x \partial z}$.

First, we divide the 3D stencil into multiple 2D slices and then further divide each 2D slice into multiple 2D tiles. After that, we apply the following two strategies:

Strategy (1): putting only one slice into the shared memory and putting the other neighboring points into registers (shown in Figure 7(a)). The round points denote all the neighboring points and the triangular points denote the additional points for updating the neighboring points. There are 12 additional round points and 4 triangular points in the XOZ plane, with the same number in the YOZ plane. Therefore, we must put 32 floats in registers for each displacement. In other words, we have to put 96 points into registers for the three displacements (U, V, W).

Strategy (2): putting five slices that contain all the neighboring points (Figure 7(b)) into the shared memory, therefore no additional points or neighboring points will be stored in registers.

For simplicity, we assume that our block size is 32×16 and we use a single-precision floating-point type here. We evaluate these two strategies by considering the size of the shared memory and registers in order to decrease the data-access latency.

5.2.1 Register usage. The capacity of registers for each block is 64 kB for Kepler and 32 kB for Fermi. Therefore, the registers can only hold 32 points on Kepler and 16 points on Fermi for each thread (calculated using equation (11)). Additionally, each point in

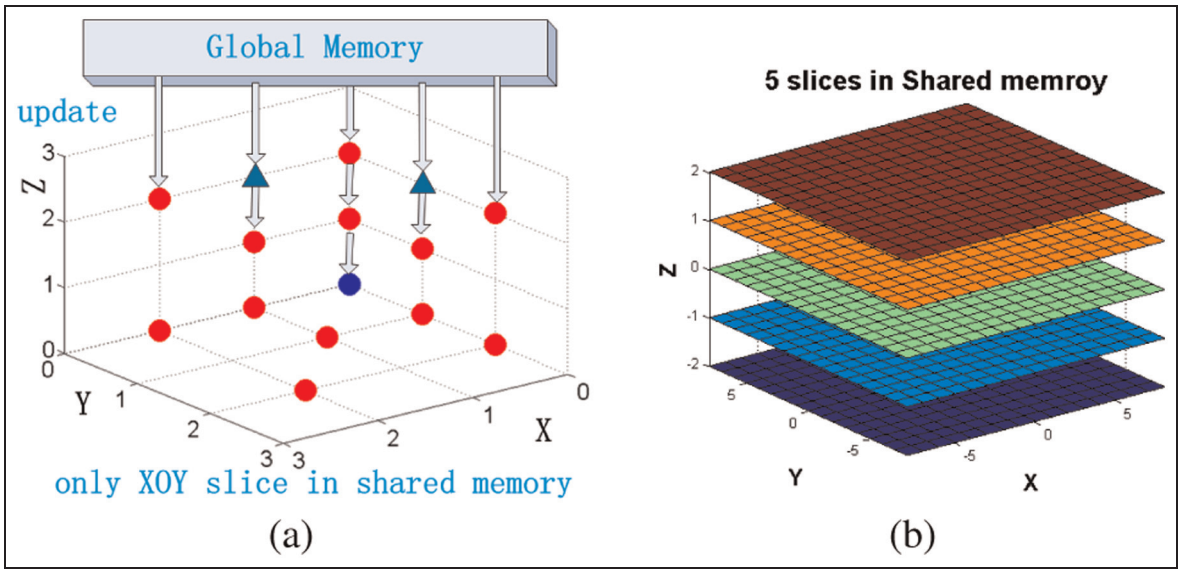

Figure 7. Two strategies for managing slices. Strategy (1): putting the XOY slice into shared memory and other neighboring points into registers. The round dots denote the neighboring points. In order to update the data, we have to add the triangular points. Strategy (2): putting five slices into shared memory, which contain all the neighboring points.

our stencil depends on several other parameters. Therefore, we have to put these additional parameters into the registers because they are accessed very frequently. For Strategy (1), we have to store 96 points in the registers, which not only exceeds the capacity of registers but also leaves no space for storing the additional necessary parameters. In contrast, for Strategy (2), we do not need to store any neighboring point in the registers.

$$\frac{block\_register\_capacity}{block\_size \times floating\_point\_size} = max\_points \quad (11)$$

5.2.2 Shared memory usage. If we take the halos into consideration, the size of the tile will be $(32 + 4) \times (16 + 4)$. In fact, the shared memory could hold more than 15 tiles (U, V and W three components) if we use the 48 kB shared memory (calculated using equation (12)).

$$max\_num\_tiles = \frac{shared\_memory\_capacity}{tile\_size \times floating\_point\_size} \quad (12)$$

Based on the analysis of the registers and shared memory above, we decided to apply our five-slice scheme (Figure 7(b)) to accelerate the stencil kernel. The five-slice scheme is illustrated in Algorithm 3. Even though accessing registers is more efficient than shared memory, this trade-off is worth it for the overall performance improvement since we can store all the necessary parameters in registers for speedup.

5.2.3 Time division multiplex access. When applying the five-slice scheme, we can only set the maximum block size as 16×16 for double precision in order to put 15 slices into the shared memory. However, the general optimization principle on GPUs is to maximize the number of threads in a block, when possible, to make full use of the quick-switching threads. To maximize the number of threads in a block for better performance, we only create one buffer in shared memory and let the three components (U, V and W) access it in sequence. Although we still cannot create the maximum 1024 threads in a block (plus the halos), there is an obvious improvement in performance from 256 threads to 512 threads.

Algorithm 3 The one-thread-one-line scheme

Step 0: Start
Step 1: Set all the elements as zero
Step 2: Put the five slices into shared memory
Step 3: Jump to the third slice
Step 4: Perform the calculation for the in-bound points
Step 5: Perform the calculation for the points in the halos
Step 6: Synchronization
Step 7: Jump to the next slice.
Step 8: If there are more than two slices left, go to Step 4
Step 9: Synchronization
Step 10: Update U, V and W
Step 11: End

5.3 Other optimizations

In addition to the two optimization schemes above, we also attempted to use other strategies for improving the

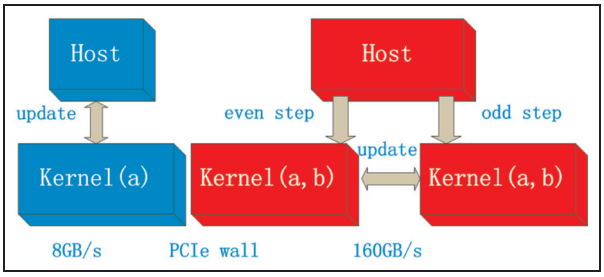

Figure 8. The left figure shows that without optimization the updated data have to be transferred between the host and GPU. The right figure shows our memory space swapping scheme which does not require the data shuffle through PCIe. The PCIe wall means the bandwidth of PCIe cannot match the bandwidth of GDDR5.

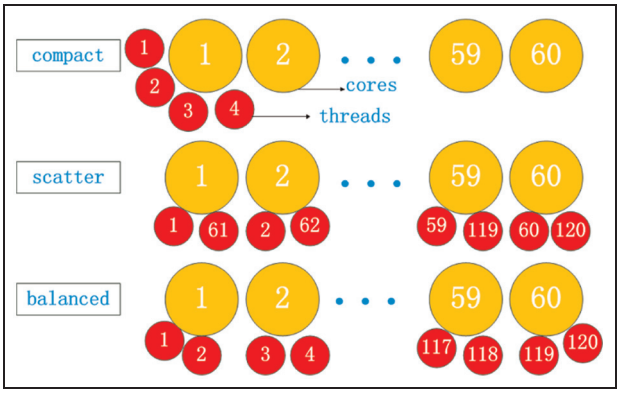

Figure 9. The three models of Affinity for load balancing. In the compact mode, the new thread will firstly be allocated to one core until the core reaches its maximum load. In scatter mode, the new thread will firstly be allocated to the core that has the lightest load. The Balanced model is a comprehensive one. It not only assigns the new thread to the core that has the lightest load but also tries to assign the neighboring threads to the same core.

performance further, including swapping memory space within the GPU card to minimize the communication overhead through PCIe (shown in Figure 8), setting the best block size according to the warp size and shared memory bank size, and adjusting the proportion between share-memory and L1 cache of the 64 kB on-chip memory appropriately to get the best configuration.

6 Optimizations on Sandy Bridge and MIC

The Sandy Bridge processor (CPU) and Xeon Phi (MIC) share many similarities in architecture (a brief analysis is given in Section 4). Therefore, we employ similar optimization techniques for CPU and MIC. These techniques generally fall into three categories: task parallelism, data parallelism, and improving data reuse in the on-chip buffer.

6.1 Task parallelism

In this paper, we choose OpenMP as the programming model for task parallelism. For OpenMP, the efficient techniques of task parallelism contain the configuration of the proper thread number and the utilization of cores and threads in balance.

6.1.1 The appropriate number of threads. For both CPU and MIC, each independent hardware device owns a specific amount of physical resources, which provides an inherent task parallelism. Hence, for taking full advantage of this mechanism, a practical way is to set the number of threads according to the number of physical resources in each device. For Sandy Bridge CPUs, there are 16 physical cores. For Intel® MIC, there are 60 physical cores. Additionally, each MIC core is equipped with 4 hardware threads, thus there are 240 virtual cores in a Xeon Phi processor. Therefore, we set the number of threads as 16 and 240 for the Sandy Bridge CPU and MIC, respectively.

6.1.2 Load balancing. Another technique used for taking full advantage of the existing hardware resources is load balancing. In our case, both the coherent L2 cache on MIC and the shared L3 cache on Sandy Bridge CPU support the Non-Uniform Memory Access (NUMA) effect. Therefore, allocating the virtual threads to the proper cores is highly necessary. We use the Affinity model (shown in Figure 9) to accomplish this. The most effective mode is 'Compact' for CPUs and 'Balanced' for MIC. In this way, we decompose the task into several portions and distribute them across all the physical devices evenly.

6.2 Data parallelism

After accomplishing efficient task parallelism, another concern is how to achieve data parallelism in each physical device. A practical solution to this issue involves referring to the SIMD mechanism, which is supported by both Sandy Bridge CPU and Intel® MIC. In addition, MIC also provides a VPU for more efficient vectorization. MIC supports 512-bit instruction, which means 16 single-precision or 8 double-precision operations can be executed at one time. Together with the vectorization scheme, we apply memory alignment so that the vectorization can use aligned load instructions to achieve a more efficient data parallelism.

6.3 Data reuse

In most cases, the two-level parallelism (task and data) contributes a significant improvement in performance.

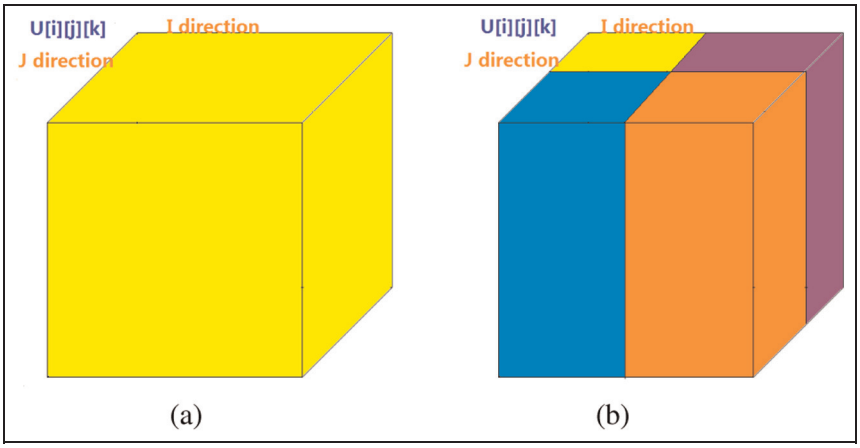

Figure 10. We decompose the domain in the first two dimensions (the I and J directions) because the data in the third dimension (the K direction) is stored in a continuous address. Our experiments also show that it is meaningful to block in the third dimension (the K direction). Figure (a) shows the original domain, and (b) represents the sub-domains. This figure serves as an sample illustration. We do not necessarily divide the problem into four parts.

However, the high parallelism also requires a large amount of data during a short period of time, which may be beyond the capacity of the peak memory bandwidth. This bottleneck limits the further improvement in performance under certain circumstances. Therefore, we employ the blocking scheme to decompose each 3D domain into several smaller 3D sub-domains according to the different cache levels (Figure 10). In order to obtain the best configuration of block and cache size, we make an automated search over all the possible blocking schemes in one, two and three dimensions, respectively.

6.4 The choice between off-load mode and native mode

Similar to the GPGPU, the bandwidth of the PCI-express bus in the Intel® MIC is still not satisfactory for conveying the data exchanges between the host CPU and the MIC cards, making it one of the major performance bottlenecks for MIC. Fortunately, the native mode of MIC allows us to execute the entire program on the co-processor directly without code modification to avoid any cost for the data transmission between host and device. However, native mode does not apply to all the cases. For instance, porting the serial code to MIC generally brings no benefit (see Section 8.1).

6.5 Other optimization techniques

In addition to the techniques above, we also tried other methods such as: (1) prefetching the data in the cache to reduce the pressure of memory bandwidth; (2) exploring the schedule model in OpenMP for the proper relationship between thread and iteration; (3) employing Cilk array notation rather than the implicit semi-automatic compiler hints to vectorize the data-intensive part explicitly; (4) using Cilk Plus to replace OpenMP for task parallelism; and (5) rewriting the source codes by applying the highly parallel Intel® Math Kernel Library. However, these techniques did not improve the overall performance significantly for our case.

7 Experimental results and summary

In addition to the specific optimization methodologies for GPGPU and MIC respectively, we also utilize some general techniques that are suitable for all the evaluated architectures including: (1) converting the high-latency operations (e.g. division) to efficient operations (e.g. multiplication); (2) getting rid of branches to avoid discontinuities in SIMD and Single Instruction Multiple Thread (SIMT) execution; and (3) preferring Structure of Array (SOA) rather than Array of Structure (AOS) for continuous memory access.

In terms of performance measurement, we use Performance Application Programming Interface (PAPI) (Browne et al., 2000) to measure the serial code on Sandy Bridge CPU (simultaneous multi-threading and SIMD disabled) for the total number of floating-point operations. Then we use the number of floating-point operations per second (Flops) to represent the performance. The peak performances achieved by the architectures are shown in Tables 3 and 4 and Figure 11.

Another important issue is the performance efficiency (illustrated in equation (13)), which indicates how deep we have explored for the computing potential of a given architecture. From Figure 12 and Table 4, we can find that although CPUs do not necessarily give

Table 3. The baseline performances. For CPU and MIC they are measured for the serial version of the LWC stencil using one thread; for the two types of GPUs, they are measured for the LWC stencil using the one-thread-one-point scheme.

Architectures	CPU	MIC	Fermi	Kepler
Single precision (Gflops)	2.651	0.1874	83.09	173.7
Double precision (Gflops)	3.373	0.1563	40.89	86.10

Table 4. The peak performances obtained on the LWC stencil.

Architectures	CPU	MIC	Fermi	Kepler
Single precision (Gflops)	70.25	129.1	131.2	186.8
Single-precision efficiency	13.72%	6.390%	12.74%	4.730%
Double precision (Gflops)	51.25	58.70	75.00	87.47
Double-precision efficiency	20.02%	5.810%	14.56%	6.630%

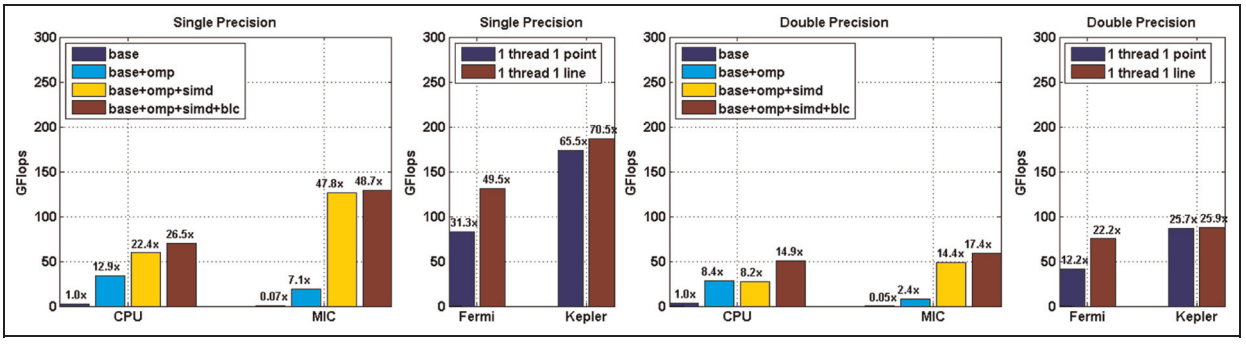

Figure 11. This figure shows the peak performances achieved by Sandy Bridge CPUs, KNC, the Fermi GPU and the Kepler 20× GPU for single precision and double precision for the stencil kernel. The number on top of each bar represents the number of times performance improvement that scheme achieved compared to the base (serial code) Sandy Bridge CPU case. 'base' means the performance of the serial code. 'omp' means using OpenMP for task parallelism. 'simd' means using vectorization for data parallelism. 'blc' means using cache blocking to improve data reuse in the on-chip buffer. '1 thread 1 point' denotes Scheme (1) for GPU optimization (using one thread to handle one computing point), and '1 thread 1 line' denotes Scheme (2) for GPU optlimization (putting five neighboring slices into shared memory).

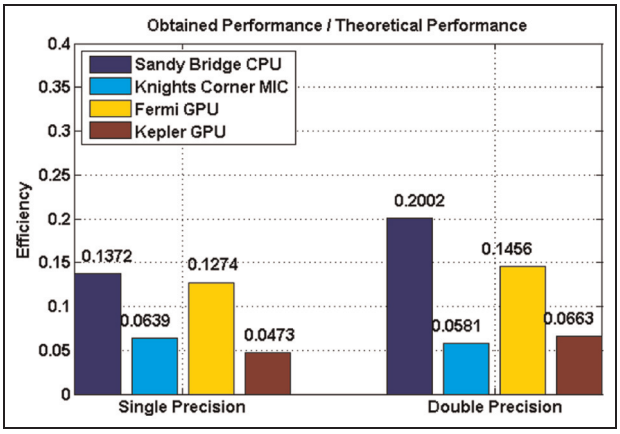

Figure 12. This figure sums up the efficiencies achieved by the four architectures. The efficiency is calculated according to equation (13). The efficiency denotes the effectiveness of our optimizations for the computing potential of a given architecture.

the highest performance, they does achieve the best performance efficiency.

From Table 2, we can see that the RCMA of our stencil kernel is much lower than the RCMB of the evaluated architectures. Even though we could fully take advantage of the bandwidth, the performance for our stencil kernel is still not satisfactory. For example, the maximum expected performance (equation (15)) of our stencil on Kepler for the non-data-reuse situation is 94 Gflops (single precision) or 47 Gflops (double precision). Therefore, to get the peak performance on the evaluated architectures for our stencil kernel, the required memory bandwidth from the evaluated architectures (equation (14)) for the non-data-reuse case is 7900 GB/s (single precision) or 5280 GB/s (double precision), which is way higher than what our current evaluated architectures can provide (shown in Table 2).

Thus, constrained by the mis-marriage between the stencil RCMA and architecture RCMB, the existing optimization techniques cannot improve the performance further unless hardware modifications are applied.

8 Comparisons and analysis

$$Performance\_Efficiency = \frac{Stencil\_Peak\_Performance}{Architecture\_Theoretical\_Peak\_Performance} \quad (13)$$

$$Required\_Memory\_Bandwidth = \frac{Architecture\_Peak\_Performance}{Stencil\_Flop\_Byte\_Ratio} \quad (14)$$

$$Maximum\_Expected\_Performance = Stencil\_Flop\_Byte\_Ratio \times Architecture\_Memory\_Bandwidth \quad (15)$$

8.1 Comparison between the Sandy Bridge CPU and MIC

From Figure 11, we see that significant improvements in performance over the base case (serial code) are achieved for both CPU and MIC after a series of optimizations. Although MIC achieved a higher peak performance than Sandy Bridge, its performance efficiency was worse (shown in Figure 12). This is mainly because the base performance of MIC is much lower than the Sandy Bridge CPU (shown in Table 3). The base performance of MIC is 14.2 times and 21.6 times lower than Sandy Bridge for single precision and double precision respectively.

There are several factors that cause the inferior base performance of MIC (serial version) compared to Sandy Bridge: (1) the difference in clock rate results in a performance discrepancy of two times; (2) a given instruction of MIC cannot be executed in the consecutive two cycles, which may account for an additional performance discrepancy of two times; and (3) the Intel® MIC does not provide out-of-order execution and L3 cache, which may account for the remaining performance discrepancy of three to five times.

On the other hand, from the CPU and MIC peak performances shown in Figure 11, we can observe that the Intel® MIC outperforms Sandy Bridge (one socket) by 4.1 $\left(\frac{129.1\times 2}{62.70}\right)$ times for single precision and by 2.3 $\left(\frac{58.69\times 2}{51.24}\right)$ times for double precision. We can actually use the performance difference of the base version from MIC and Sandy Bridge to estimate the peak performance difference. For instance, the base performance of MIC is 14.2 times or 21.6 times lower than Sandy Bridge for single precision and double precision, respectively. In addition, each MIC card has 240 hardware threads while each CPU socket has only 8 hardware threads. Thus we can estimate the performance speedup for MIC over Sandy Bridge using only task parallelism: $\frac{240}{14.2\times 8} = 2.11$ times (single precision) and $\frac{240}{21.6\times 8} = 1.39$ times (double precision). With the additional 2 times performance difference in SIMD width for MIC over Sandy Bridge, we can theoretically estimate that MIC outperforms Sandy Bridge for our stencil kernel by 4.22 times (single precision) and 2.78 times (double precision), which is very close to the 4.10 and 2.30 obtained from the measurements.

Based on these results, we conclude that, in terms of performance for our specific stencil, one Sandy Bridge hardware thread is equivalent to 14.2 (single precision) or 21.6 (double precision) Knights Corner hardware threads and one Knights Corner card is equivalent to 4.1 (single precision) or 2.3 (double precision) Sandy Bridge sockets.

8.2 Comparison between the Fermi and Kepler GPUs

From the GPU performance shown in Figure 11, we can see that the one-thread-one-point scheme (maximum parallelism) on the Kepler GPU exhibits a 108.9% improvement in single-precision performance over that of the Fermi GPU. In addition, the one-thread-one-line scheme (on-chip data reuse) on the Kepler GPU exhibits a 42.42% improvement in single-precision performance over that of the Fermi GPU. Several important architectural improvements from the Fermi to the Kepler GPU (http://www.nvidia.com/content/PDF) may account for these performance improvements:

(1) There is no difference between Fermi C2070 and Kepler 20× in the number of SMs (both have 14). For each SM, however, the Kepler GPU has 192 single-precision cores and 64 double-precision units, while the Fermi GPU only has 32 cores. Additionally, from the Fermi GPU to the Kepler GPU, the number of special function units (SFUs) increases from 4 to 32, and the number of LD/ST increases from 16 to 32.
(2) From the Fermi GPU to the Kepler GPU, the number of warps available for scheduling concurrently on each SM increases from two to four. Moreover, each Kepler warp scheduler controls double-instruction dispatch units, which means two independent instructions can be dispatched to each warp per cycle (shown in Figure 13). Combined with improvement (1), the parallelism of the Kepler GPU is four times higher than the Fermi GPU.
(3) The size of the block registers changes from 32 to 64 kB. This feature ensures each thread is allocated more registers.

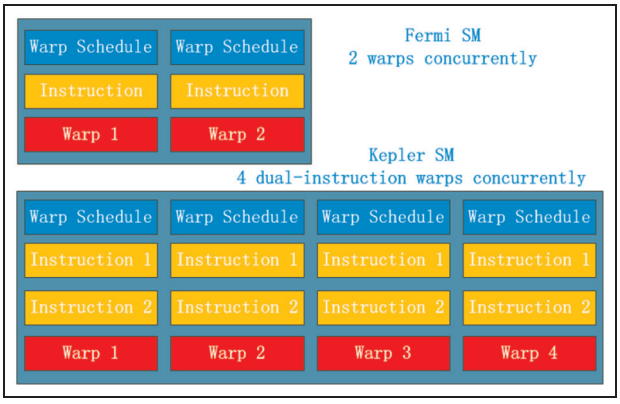

Figure 13. Each Fermi SM allows two warps to be issued and executed concurrently while a Kepler allows four; one instruction is dispatched to each warp for Fermi per cycle, while two independent instructions could be dispatched to each warp for Kepler per cycle (http://www.nvidia.com/content/PDF).

As described previously, our one-thread-one-point scheme requires high parallelism, which can benefit from the architectural improvements (1) and (2). Moreover, improvement (3) could further satisfy the high-register-access demand of our one-thread-one-point scheme. Therefore, we conclude that optimization techniques that require high parallelism and a large number of register accesses will likely benefit from the architectural improvements from Fermi to Kepler.

From the one-thread-one-point scheme to the on-chip data reuse scheme (one-thread-one-line), there are significant performance improvements ($1.6\times$ and $1.8\times$) on the Fermi architecture. However, the performance improvements between these two schemes on Kepler are significantly smaller ($1.1\times$ and $1.0\times$). There are several reasons behind this. First of all, as mentioned previously, the architectural improvements from Fermi to Kepler bear a certain degree of responsibility for this because the one-thread-one-point scheme can benefit more from Kepler than Fermi in terms of performance. Moreover, the limited 48 kB shared memory on Kepler highly constraints the block size we can choose for the on-chip data reuse scheme. Although we have applied a time-division multiplex access method for shared memory reuse (Section 5.2.3), we can create, at most, 512 threads rather than the maximum 1024 threads in a block for double precision. Therefore, the unimproved shared memory from Fermi to Kepler hinders the further improvement in performance.

8.3 Comparison of GPU and MIC

In this section, we will provide detailed comparisons between the GPU and the MIC architectures in several important aspects. Our objective is to provide scenarios for users to choose the proper accelerators for their applications.

8.3.1 Shared memory vs. L1 cache. From the single-precision results of Figure 11, we can observe that the performance of the omp-simd-blc MIC is slightly worse than that of one-thread-one-line on Fermi, and much worse than the performance of one-thread-one-line on Kepler. This is because the L1 cache of MIC is predominantly controlled by the compiler, while the shared memory of GPUs can be more flexibly managed by users. Even though we can conclude that user-controlled shared memory is more efficient than a compiler-controlled L1 cache for our stencil kernel optimization case, we cannot treat this conclusion as a general recommendation because the two different methods vary in the complexity of programming.

8.3.2 Performance and programmability. Programmability is one of the key issues for users who know little about the low-level programming method. Thus user-friendly program interfaces will be extremely important for the popularization of this programming model.

The programming models for MIC are based on several commonly used HPC methods such as OpenMP and Intel® Cilk, which are easier for developers to grasp in a short period of time. Moreover, since MIC is based on $\times 86$ architecture, it will require little, or even no, effort to port software from the conventional $\times 86$ platform to MIC. Additionally, MIC cannot only be run as an accelerator (off-load model), but can also be used as a host (Native model). This advantage may provide additional convenience for users as well. Last but not least, there have been many sophistically developed tools based on Intel architectures for compiling, debugging, and performance analysis, which can be easily ported to MIC.

As for GPU, with the introduction of OpenACC (Wienke et al., 2012), programming models may become more user friendly and the existing software will also be more compatible with GPUs than before. In fact, the OpenACC Application Programming Interfaces (APIs) may become part of the OpenMP specifications in the near future (http://www.openacc-standard.org/node/47). However, porting the large amount of existing software, such as MPI tools, to GPU still takes a lot of effort.

8.3.3 Power efficiency. Power consumption is another key constraint for developing applications and architectures (Balakrishnan, 2012), especially in future exascale system design. Power efficiency (the ratio between peak performance and power consumption) has already become equally as important as peak performance when designing or evaluating an architecture. For simplicity, we use the power consumption of the entire node (one CPU and two cards) to evaluate the power efficiency. From Table 5, we can observe that the

Table 5. Comparison of the NVIDIA Kepler GPU and the Intel® MIC.

Type	Kepler 20× GPU	Intel® KNC MIC
Peak performance single (Gflops)	186.8	129.1
Power consumption single (Watt)	419.1	466.1
Power efficiency single (Gflops/Watt)	0.4457	0.2770
Peak performance double (Gflops)	87.47	58.69
Power consumption double (Watt)	409.2	460.1
Power efficiency double (Gflops/Watt)	0.2138	0.1276

GPU-based node is 1.61 times (single precision) and 1.68 times (double precision) more efficient than the MIC-based node.

8.3.4 Suitable parallelism. The granularity of parallelism often influences the performance. In addition, differences in the best thread numbers of GPU and MIC require us to employ different optimizing schemes. Therefore, we attempt to find the relationship between architectural differences and suitable parallelism through comparisons between the GPU and MIC architectures.

GPU employs the SIMT architecture. The multiprocessor creates, schedules, and executes a group of threads concurrently. Similarly, MIC is based on the SIMD architecture, which is the foundation for the vectorization scheme on MIC. Although both SIMD and SIMT are useful for enhancing parallelism, there are essential differences between them in the granularity of parallelism (shown in Figure 14):

(1) SIMD: one MIC thread executes one 512-bit instruction. Therefore, one MIC thread corresponds to 16 single-precision operations and 32 single-precision floating points (or 8 double-precision operations and 16 double-precision floating points).
(2) SIMT: 32 threads (a warp) execute the same 32 instructions concurrently. Thus, one thread corresponds to one operation and two single-precision floating points.

In addition, in terms of core architecture (Figure 15), we find that the MIC core is composed of more complicated units. Therefore, the GPU thread is more lightweight and provides more fine-grain parallelization compared to the MIC thread. Based on their unique architectures, both of them have their own advantages over the other for different types of applications.

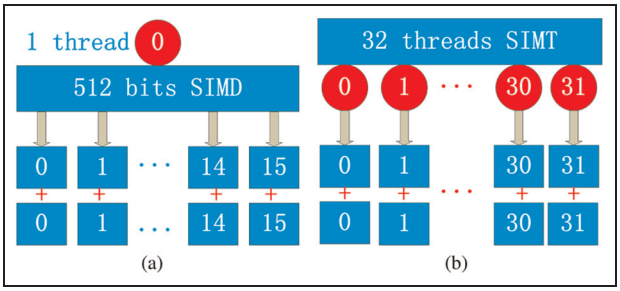

Figure 14. (a) The SIMD for MIC: one thread executes one 512-bit instruction, which handles 32 single-precision floating points at one time. (b) The SIMT method of GPU: 32 threads execute the same 32 instructions concurrently, one thread only handles 2 single-precision floating points.

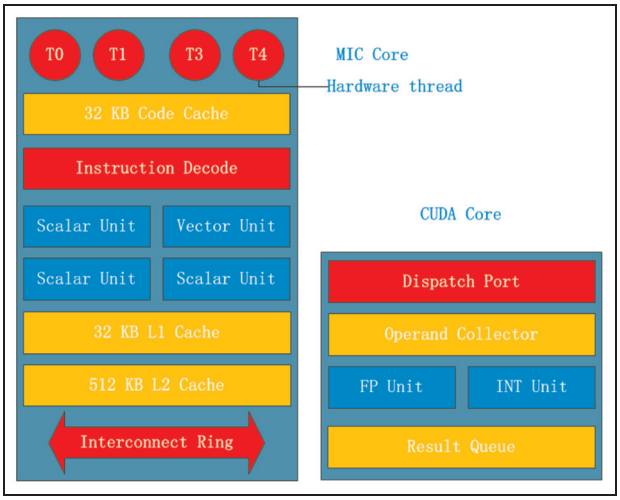

Figure 15. An MIC core (left) and a Fermi GPU core (right). It is obvious that the MIC core is much more complicated (http://www.nvidia.com/content/PDF, http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner).

For instance, GPU can provide millions of lightweight threads. A Kepler SM allows 4 dual-instruction warps to be issued and executed concurrently, which provides a 256-instruction ($32 \times 4 \times 2$) parallelism for each SM and a 3584-instruction (256×14) parallelism for each GPU card. Due to the fact that a given instruction of MIC cannot be executed in the consecutive two cycles, each MIC card can only provide a 122-instruction parallelism. Thus, the parallelism of a GPU card is 29.4 times that of an MIC card.

In contrast, as mentioned above, each MIC instruction corresponds to 16 single-precision or 8 double-precision operations. Therefore, the parallel granularity of MIC is 16 and 8 times that of GPU for single and double precision respectively.

With the additional performance discrepancies in thread switching speed and clock frequency, the analysis above is in line with our experimental results. Thus,

for optimizing applications on GPU, we could maximize parallelism in order to use enough lightweight threads (under resource limitation); as for the optimizations on MIC, in order to reach the peak performance of each powerful core, coarse-grained parallelism and a modest number of threads may be a better choice.

9 Conclusion

In this paper, we proposed methodologies to accelerate the widely used forward-modeling 3D wave propagation method based on Sandy Bridge CPU, KNC, Fermi GPU, and Kepler 20× GPU platforms, and achieved performance efficiencies (equation (13)) ranging from 4.730% to 20.02%.

We employed 2 schemes to optimize our 114-point stencil on GPU: (1) putting five neighboring slices in shared memory to maximize reuse of on-chip data and to solve the problems caused by insufficient registers; and (2) using one thread to handle one point in order to maximize the parallelism degree. Inspired by the experimental results of these two schemes on two types of GPUs, we concluded that the improvements from Fermi to Kepler will favor the optimization techniques which provide extremely high parallelism and require a large number of register accesses. In addition, we found that the unimproved shared memory from Fermi to Kepler may hinder the further improvement in performance for on-chip data reuse techniques.

We also used various techniques to optimize our application on Sandy Bridge CPU and MIC, including task parallelism, data parallelism, on-chip data reuse, etc. We found that although KNC MIC quadruples Sandy Bridge CPU in the theoretical peak performance, the actual speedup is 1.2× (double precision) and 2.1× (single precision) due to lower clock frequency, lack of out-of-order execution, inferior ×86 core and insufficient cache.

We also presented cross-platform comparison analysis. Our focus is on the detailed comparison between MIC and GPU in terms of the performance–effort relationship, power efficiency, and parallel granularity. Our analysis indicated that although GPU requires a higher programming effort than MIC, it may achieve better improvements in peak performance. Specifically, one Kepler 20× GPU card is equivalent to 1.45 (single precision) and 1.49 (double precision) KNC MIC card in terms of peak performance for our stencil kernel. For power efficiency, our GPU-based node is 1.61 times (single precision) and 1.68 times (double precision) more efficient than our MIC-based node. Additionally, the parallelism of a GPU card is 29.4 times that of a MIC card, but the parallel granularity of MIC is 16 times (single precision) and 8 times (double precision) that of GPU.

The general findings for all the architectures are: (1) user-managed shared memory on GPGPU is more efficient than compiler-controlled L1 cache on ×86 cores for the on-chip data reuse strategy; and (2) on-chip data reuse is highly important when the architecture RCMB is higher than the stencil RCMA.

Acknowledgements

We would like to thank Zihong Lv for his advice in paper writing.

Funding

This work was supported in part by the National Natural Science Foundation of China (grant numbers 61303003 and 41374113) and the National High-tech R&D (863) Program of China (grant number 2013AA01A208).

References

Asano S, Maruyama T and Yamaguchi Y (2009) Performance comparison of FPGA, GPU and CPU in image processing. In: *2009 international conference on field programmable logic and applications*, Prague, Czech Republic, 31 August-2 September 2009, pp. 126–131. Piscataway: IEEE Press.

Balakrishnan M (2012) Power consumption in multi-core processors. *Contemporary Computing* 306: 3.

Benthin C, Wald I, Woop S, Ernst M and Mark W (2012) Combining single and packet-ray tracing for arbitrary ray distributions on the Intel MIC architecture. *IEEE Transactions on Visualization and Computer Graphics* 18(9): 1438–1448.

Blanch J and Robertsson J (2007) A modified Lax-Wendroff correction for wave propagation in media described by Zener elements. *Geophysical Journal International* 131(2): 381–386.

Browne S, Dongarra J, Garner N, Ho G and Mucci P (2000) A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications* 14(3): 189–204.

Clapp R, Fu H and Lindtjorn O (2010) Selecting the right hardware for reverse time migration. *The Leading Edge* 29(1): 48–58.

Dablain M (1986) The application of high-order differencing to the scalar wave equation. *Geophysics* 51(1): 54–66.

Duran A and Klemm M (2012) The Intel® many integrated core architecture. In: *2012 international conference on high performance computing and simulation*, Madrid, Spain, 2–6 July 2012 pp. 365–366. Piscataway: IEEE Press.

Heinecke A, Klemm M, Pflüger D, Bode A and Bungartz H (2012) Extending a highly parallel data mining algorithm to the Intel® many integrated core architecture. In: *Euro-Par 2011: Parallel processing workshops*, Bordeaux, France, 29 August–2 September 2011, pp. 375–384. Berlin: Springer.

Lax P and Wendroff B (1964) Difference schemes for hyperbolic equations with high order of accuracy. *Communications on Pure and Applied Mathematics* 17(3): 381–398.

Li Z and Song Y (2004) Automatic tiling of iterative stencil loops. *ACM Transactions on Programming Languages and Systems* 26(6): 975–1028.

Liu G, Liu Q, Li B, Tong X and Liu H (2009) GPU/CPU co-processing parallel computation for seismic data processing in oil and gas exploration. *Progress in Geophysics* 24(5): 1671–1678.

Manavski S and Valle G (2008) CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics* 9(Suppl 2): S10.

Meng J and Skadron K (2009) Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In: *23rd international conference on supercomputing*, Yorktown Heights, NY, USA, 8–12 June 2009 pp. 256–265. New York: ACM.

Michéa D and Komatitsch D (2010) Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. *Geophysical Journal International* 182(1): 389–402.

Micikevicius P (2009) 3D finite difference computation on GPUs using CUDA. In: *2nd workshop on general purpose processing on graphics processing units*, Washington, DC, USA 08 March 2009 pp. 79–84. New York: ACM.

Mudge T (2001) Power: A first-class architectural design constraint. *Computer* 34(4): 52–58.

Nickolls J and Dally W (2010) The GPU computing era. *IEEE Micro* 30(2): 56–69.

Okamoto T, Takenaka H, Nakamura T and Aoki T (2010) Accelerating large-scale simulation of seismic wave propagation by multi-GPUs and three-dimensional domain decomposition. *Earth Planets and Space* 62(12): 939–949.

Raina R, Madhavan A and Ng A (2009) Large-scale deep unsupervised learning using graphics processors. In: *26th annual international conference on machine learning*, Montreal, Canada, 14–18 June 2009 vol. 382, pp. 873–880. New York: ACM.

Satish N, Kim C, Chhugani J, Nguyen A, Lee V, Kim D, et al. (2010) Fast sort on CPUs, CPUs and Intel MIC architectures. Technical report, Intel, Santa Clara, CA, USA.

Satish N, Kim C, Chhugani J, Saito H, Krishnaiyer R, Smelyanskiy M, et al. (2012) Can traditional programming bridge the ninja performance gap for parallel computing applications? In: *39th international symposium on computer architecture*, Portland, Oregon, USA, 9–13 June 2012 pp. 440–451. Piscataway: IEEE Press.

Sei A and Symes W (1995) Dispersion analysis of numerical wave propagation and its computational consequences. *Journal of Scientific Computing* 10(1): 1–27.

Surkov V (2010) Parallel option pricing with Fourier space time-stepping method on graphics processing units. *Parallel Computing* 36(7): 372–380.

Unat D, Cai X and Baden SB (2011) Mint: realizing CUDA performance in 3D stencil methods with annotated C. In: *25th international conference on supercomputing*, Tucson, Arizona, USA,31 May–4 June 2011 pp. 214–224. New York: ACM.

Wienke S, Springer P, Terboven C and an Mey D (2012) OpenACC™ first experiences with real-world applications. In: *18th international conference on parallel processing*, Rhodes Island, Greece, 27–31 August 2012, pp. 859–870. Berlin: Springer.

Zhang Y and Mueller F (2012) Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In: *10th international symposium on code generation and optimization*, San Jose, California, USA, 31 March–4 April, pp. 155–164. New York: ACM.

Author biographies

Yang You is an MPhil candidate in the Department of Computer Science and Technology at Tsinghua University. His major research interest is parallel/distributed computing. Specifically, his research is focused on designing high-performance algorithms to optimize real-world applications such as massive graph traversal (e.g. BFS), scientific computing (e.g. stencils), machine learning (e.g. classification), and bioinformatics (e.g. genomes) on many-core and multi-core architectures like CPUs, GPUs and Intel® Xeon Phi co-processors. He is also interested in solving large-scale problems on multi-node clusters. He is a student member of the IEEE.

Haohuan Fu is an Associate Professor in the Ministry of Education Key Laboratory for Earth System Modeling, and the Center of Earth System Science at Tsinghua University. His research interests mainly focus on high-performance computing applications in earth and environmental sciences. Dr Fu has a PhD in computing from Imperial College, London. He is a member of the IEEE.

Shuaiwen Leon Song is currently a Research Staff Scientist for the Performance Analysis Laboratory at Pacific Northwest National Laboratory (PNNL). He graduated with a PhD from the Computer Science Department of Virginia Tech in May 2013. Before he joined PNNL, he was a member of the Scalable Performance Laboratory directed by Dr Kirk W Cameron at Virginia Tech. In the past, he has been an intern at the Center for Advanced Computing at Lawrence Livermore National Laboratory, at PNNL, and also an R&D intern for the Architecture Research Division at NEC Research American in Princeton, New Jersey. He was a 2011 ISCR scholar and recipient of the 2011 Paul E Torgersen Excellent Research Award. His research interests lie in several areas of high-performance computing.

Maryam Mehri Dehnavi received her PhD in Electrical and Computer Engineering from McGill University, Canada in 2012. She is currently a Postdoctoral Researcher in the Computer Science Department at MIT. Her research is focused on accelerating scientific computations on parallel architecture. The work involves developing domain-specific libraries for finite difference and finite element methods, designing auto-tuners to tune the performance of parallel code on

hybrid platforms and reformulating the communication patterns specific within the algorithms. During her PhD she worked in the Parallel Computing Laboratory at UC Berkeley where she worked on developing communication-avoiding Krylov solvers for many-core architecture. She has also worked in the Parallel Systems & Computer Architecture Laboratory at UC Irvine as a Visiting Researcher, and was a Senior R&D engineer at Qualcomm Inc. before joining MIT.

Lin Gan is a PhD candidate in the Department of Computer Science and Technology at Tsinghua University. His research interests include high-performance solutions to global atmospheric modeling and exploration geophysics, focusing on algorithmic development and performance optimizations based on hybrid platforms such as CPUs, FPGAs, and GPUs. He has a BE in Information and Communication Engineering from the Beijing University of Posts and Telecommunications. He is a student member of the IEEE.

Xiaomeng Huang is an Associate Professor in the Ministry of Education Key Laboratory for Earth System Modeling, and the Center of Earth System Science at Tsinghua University. He received his PhD from the Department of Computer Science and Technology at Tsinghua University in 2007. He is especially interested in high-performance computing, cloud computing, ocean modeling etc.

Guangwen Yang is Professor of Computer Science and Technology at Tsinghua University. He received his PhD from the Department of Computer Science at the Harbin Institute of Technology University in 1996. He is mainly engaged in the research of grid computing and parallel and distributed computing.