2015 IEEE 18th International Conference on Computational Science and Engineering

CPU+GPU Programming of Stencil Computations for Resource-Efficient Use of GPU Clusters

Mohammed Sourouri*†, Johannes Langguth*, Filippo Spiga‡, Scott B. Baden§ and Xing Cai*†
*Simula Research Laboratory, Norway
†University of Oslo, Norway
‡University of Cambridge, UK
§University of California, San Diego, USA

mohamso@simula.no, langguth@simula.no, fs395@cam.ac.uk, baden@ucsd.edu, xingca@simula.no

***Abstract*—On modern GPU clusters, the role of the CPUs is often restricted to controlling the GPUs and handling MPI communication. The unused computing power of the CPUs, however, can be considerable for computations whose performance is bounded by memory traffic. This paper investigates the challenges of simultaneous usage of CPUs and GPUs for computation. Our emphasis is on deriving a heterogeneous CPU+GPU programming approach that combines MPI, OpenMP and CUDA. To effectively hide the overhead of various inter- and intra-node communications, a new level of task parallelism is introduced on top of the conventional data parallelism. Combined with a suitable workload division between the CPUs and GPUs, our CPU+GPU programming approach is able to fully utilize the different processing units. The programming details and achievable performance are exemplified by a widely used 3D 7-point stencil computation, which shows high performance and scaling in experiments using up to 64 CPU-GPU nodes.**

***Keywords*-GPU, CUDA, Stencil, MPI, CPU+GPU computing**

I. Introduction

GPU clusters are becoming increasingly popular, because they deliver excellent performance and high power efficiency in many types of scientific applications [1], [2]. However, the current generation of GPUs are not general-purpose, which means they cannot act as standalone devices. GPUs are therefore dependent on general-purpose CPUs that can act as hosts. Even if future generations of GPUs could operate as standalone devices, a virtually CPU-free cluster is not advisable since GPUs are not suitable for certain HPC workloads. Future GPU clusters such as LLNL Sierra and ORNL Summit [3] suggest that heterogeneous CPU-GPU clusters will remain widespread, which prompts the development of CPU+GPU computing.

In heterogeneous CPU+GPU applications, several programming models, such as MPI, OpenMP and CUDA are combined to exploit the strengths of CPUs and GPUs to achieve high performance. However, three principal problems arise when going from homogeneous to heterogeneous CPU+GPU computations.

First, an additional work partitioning between the CPUs and GPUs needs to be introduced. Unlike the standard partitioning among the compute nodes of a homogeneous cluster, this new workload division is asymmetric, which requires a division ratio that is proportional to the relative compute speed of the two processing units. Second, inter- and intra-node data exchanges involving both the CPUs and GPUs, must be properly programmed in order to attain high communication efficiency. Third, a scheme for controlling intra-node communication, synchronization and computation must be implemented.

The challenge here lies in establishing an effective overlap between the computation and the inter- and intra-node communication. This requires introducing proper task parallelism in addition to the usual data parallelism.

To tackle the first problem, we make use of the fact that many scientific computations are memory-bound. Thus, we can roughly predict the CPU-only and GPU-only performance based on realistic memory bandwidth values obtained by e.g. the STREAM benchmark [4]. Such a simple modeling can suggest a reasonable workload division between the CPU and the GPU(s) on each compute node. At the same time, care should be taken to make a heterogeneous implementation flexible enough for fine-tuning the workload division.

We address the second problem by using a single MPI process to control the CPU and the GPU, instead of two separate MPI processes. The resulting CPU-GPU data exchanges are performed via `cudaMemcpyAsync`. For GPU-GPU data exchanges, we use a portable approach that lets the respective CPU relay the messages. In other words, MPI messages between CPUs cover both CPU $\leftrightarrow$ CPU and GPU $\leftrightarrow$ GPU interactions. The key to efficient data exchanges lies in obtaining a good overlap with various computing activities, which are controlled by the CPU. Direct GPU $\leftrightarrow$ GPU data exchanges across nodes can easily be adopted to improve our approach, but this requires that CUDA-aware MPI with GPUDirect v3 [5] is available.

To overlap computations with the various intra-node and inter-node data exchanges, we adopt a programming style that involves MPI, OpenMP and CUDA. In such an approach, task parallelism is key. Some of the OpenMP threads are dedicated to the main part of the computation, while the remaining OpenMP threads handle other tasks, such as data movement and boundary computations.

978-1-4673-8297-7/15 $31.00 © 2015 IEEE
DOI 10.1109/CSE.2015.33

IEEE computer society

To illustrate the programming details, we choose a widely used 3D 7-point stencil. Our programming approach is by no means limited to the motivating example, and is applicable to many other numerical computations. This paper makes the following contributions:

- We develop an optimized heterogeneous CPU+GPU implementation for computing the stencil over a uniform 3D grid with detailed illustrations of advanced programming techniques (Section III).
- We show that fine-grained use of CPU threads increases parallelism and thus application performance.
- Insight into a simple performance model for heterogeneous CPU+GPU applications (Section IV).
- Experiments showing that concurrent CPU+GPU computations can outperform a highly optimized GPU-only implementation, even on GPU clusters where each compute node is equipped with two GPUs per node (Section V).

II. GPU-ONLY IMPLEMENTATIONS

In this section, we give a detailed description of the GPU-only, MPI+CUDA implementation used.

For this paper, we have chosen a representative numerical kernel that sweeps over a uniform 3D grid. The size of the grid is defined as $N_x \times N_y \times N_z$, and a 7-point stencil is used to alternatingly update the two arrays, `u_new` and `u_old`, which are stored in row-major order. The corresponding GPU versions of the two arrays are `d_unew` and `d_uold`.

A. Single-GPU Implementation

To secure good single-GPU performance, we use the pipelined wavefront technique, as described in [6] and [7]. This technique consists of introducing a for-loop to compute values column-wise along the z axis. In addition, we make use of the Kepler GPUs' fast read-only cache. In our example application, data from `d_uold` is only read, making it an ideal candidate for the read-only cache. Moreover, a small portion of the constant memory is used to store constants in order to free registers.

B. Multi-GPU Implementation

It suffices to use a conventional 3D domain decomposition to break the global domain into smaller 3D subdomains, one per GPU. In this implementation, all the computational work is done by the GPUs, while the required GPU $\leftrightarrow$ GPU interactions are realized as MPI messages that are relayed through the CPU.

To enable overlap between computation and communication, the entire computation is split into interior points and boundary points, and further, each subdomain is extended with a layer of ghost cells [8].

Although CUDA-aware MPI libraries with GPUDirect v3 such as MVAPICH2-GDR [9] can simplify multi-GPU programming, we have assumed for the sake of generality and portability that GPU-GPU interactions are relayed through the CPU. In other words, CPU $\leftrightarrow$ GPU intra-node data movement is always required, in addition to inter-node MPI communication. Since each subdomain's boundary points are computed first, the overhead related to the intra-node data transfer and inter-node MPI can be overlapped with the remaining computation on the interior grid points.

In addition to a CUDA kernel that computes the interior points, we use supplementary CUDA kernels to compute the boundary points. We create one CUDA stream per side of the subdomains in order to allow simultaneous execution of the compute kernels for the boundary points and the interior points. Furthermore, asynchronous data transfer functions such as `cudaMemcpyAsync` are used to perform concurrent CPU $\leftrightarrow$ GPU data transfers. As a result, computation and communication can be overlapped.

```
for (int t = 0; t < iterations; t++) {
  for (auto i: direction_vector)
    MPI_Irecv(recv_buf);
    ComputePackBoundary<<<dir_stream[i]>>>;
    cudaMemcpyAsync(cudaMemcpyDeviceToHost, dir_stream[i]);

  ComputeInteriorPoints<<<inner_stream>>>;

  for (auto i: direction_vector)
    cudaStreamSynchronize(dir_stream[i]);
    MPI_Isend(send_buf);

  MPI_Waitall();

  for (auto i: direction_vector)
    cudaMemcpyAsync(cudaMemcpyHostToDevice, dir_stream[i]);
    UnpackBoundary<<<dir_stream[i]>>>;

  // cudaDeviceSynchronize and swap pointers
}
```

Listing 1. An MPI+CUDA implementation using only GPUs for computation.

Listing 1 outlines the pseudocode of our multi-GPU implementation. The different sides are stored as enumerated types in a C++ vector, iterated using the C++11 `auto` keyword. Before every boundary exchange, points in the boundary region need to be computed and then stored in a buffer. The process of storing values in a buffer is referred to as *packing*, and the reverse is called *unpacking*. In our implementation, the `ComputePackBoundary` kernel performs computation of boundary points as well as packing, thus eliminating redundant global memory accesses.

Each `ComputePackBoundary` kernel is placed in its own CUDA stream. The same streams are reused for unpacking, similar to the approach presented in [10]. In Listing 1, `cudaStreamSynchronize` ensures that data from the GPU has been successfully copied to the CPU before `MPI_Isend` is called.

In the multi-GPU version presented in this section, computation and communication are overlapped since the computation of interior points is decoupled from the computation of boundary points through the use of separate CUDA streams, and thus occurs simultaneously with the MPI communication.

III. HETEROGENEOUS CPU+GPU IMPLEMENTATIONS

In this section we describe two CPU+GPU implementations that extend the multi-GPU implementation presented in Section II-B. The drawback with the GPU-only implementation is that it leaves the CPU unused most of the time.

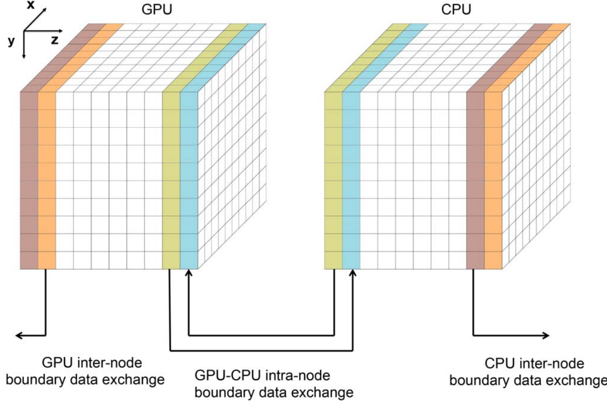


Fig. 1. Workload division between the CPU and GPU on a compute node.

Hence, both implementations described in this section aim to exploit the CPU for further performance improvements. The first implementation does so in a naive manner, while the second implementation adopts further optimizations including task parallelism, realized through OpenMP's nested parallel regions.

Both of our CPU+GPU implementations mix MPI+CUDA with OpenMP. We have already introduced the MPI+CUDA part, but not the OpenMP part. Thus, we start by describing a multi-core CPU implementation.

A. Multi-Core CPU Implementation

Our CPU kernel uses pencil shaped cache blocking along the y axis in combination with non-temporal store instructions, as described in [11]. This technique can be considered a special case of the original quadrilateral cache blocking technique presented in [12]. We have also implemented a simple auto-tuner, which ensures that the optimum cache block size is chosen when the code is moved from one cluster to another.

B. Naive Implementation

We decompose the global domain as described in Section II-B. Moreover, each subdomain is decomposed again, this time using a 1D decomposition along the z axis. This means that each subdomain is divided into two parts, as illustrated in Figure 1. The reason for decomposing each subdomain along the z axis is that a xy plane is contiguous in memory.

The total volume of inter-node data movement remains the same, but the number of MPI messages in the x and y directions are doubled, because CPUs are now also responsible for a part of the computation. For this reason, separate send and receive buffers are created for the CPU and GPU parts. These are noted as `gpu_send_buf`, `cpu_send_buf`, etc. in the following pseudocode.

In order to overlap computation with communication, we continue to use the same technique as in our multi-GPU implementation, where interior points and boundary points are treated separately. We reuse the CUDA kernels from our multi-GPU code, but write new functions for boundary point computation on the CPU side.

```
#pragma omp parallel default(shared)
for (int t = 0; t < iterations; t++) {
  #pragma omp master
  {
    for (auto i: direction_vector ex. intra boundary)
      MPI_Irecv(gpu_recv_buffer);
      MPI_Irecv(cpu_recv_buffer);
      ComputePackBoundary<<<dir_stream[i]>>>;
      cudaMemcpyAsync(DeviceToHost, dir_stream[i]);

    ComputeIntraBoundary<<<intra_stream>>>;
    cudaMemcpyAsync(DeviceToHost, intra_stream);

    ComputeInteriorPoints<<<inner_stream>>>;
  }

  for (auto i: direction_vector)
    HostComputePackInterBoundary();
    HostComputeIntraBounddary();

  #pragma omp barrier
  #pragma omp master
  {
    cudaMemcpyAsync(HostToDevice, data_stream);

    for (auto i: direction_vector ex. intra boundary)
      MPI_Isend(cpu_send_buffer);
      cudaStreamSynchronize(inter_stream);
      MPI_Isend(gpu_send_buffer);

    MPI_Waitall();

    for (auto i: direction_vector ex. intra boundary)
      cudaMemcpyAsync(HostToDevice, dir_stream[i]);
      UnpackBoundary<<<dir_stream[i]>>>;
  }

  HostComputeInteriorPoints();

  for (auto i: direction_vector ex. intra boundary)
    HostUnpackBoundary();

  #pragma omp barrier
  #pragma omp master
  {
    cudaDeviceSynchronize();
    std::swap(d_uold, d_unew);
    std::swap(u_uold, u_unew);
  }
  #pragma omp barrier
}}
```

Listing 2. A naive MPI + OpenMP + CUDA implementation.

To better mask the intra-node boundary data exchange between the GPU and CPU, we create two additional CUDA streams called *data_stream* and *intra_stream*. These streams are used for CPU $\leftrightarrow$ GPU intra-node boundary data transfers.

The basic strategy of the naive implementation is to augment the existing MPI+CUDA implementation using OpenMP, as outlined in Listing 2.

Using OpenMP, we spawn a number of CPU threads equal to the number of physical CPU cores, and declare the stencil compute loop as a parallel OpenMP region. First, we use the master thread to launch the different CUDA kernels and GPU $\rightarrow$ CPU data copies before proceeding to computation of the boundary points on the CPU. Upon completion of the last boundary on the CPU, we start the communication of the different boundaries using MPI, creating a sequential pipeline of MPI messages.

Before data from the GPU can be communicated, a cudaStreamSynchronize is necessary to ensure that data has arrived on the CPU. Moreover, MPI_Waitall is required before launching the unpacking kernels on the GPU. Next, computation of the interior points on the CPU, HostComputeInteriorPoints, is started.

Unpacking of the received boundary data on the CPU, HostUnpackBoundary, is not dependent on the computation of the interior points on the CPU, but in order to avoid delaying this computation, we call HostUnpackBoundary as soon as the computation has finished. Furthermore, before an entire iteration is concluded, the master thread, guarded by two barriers (one before and after) swaps the data pointers.

The naive implementation can be further simplified by using the multithreaded thread safety level (MPI_THREAD_MULTIPLE) provided by many MPI libraries. Although this mode greatly simplifies the actual code, it comes at the expense of performance. Because when the thread safety level is set to MPI_THREAD_MULTIPLE, the MPI library makes extensive use of synchronization to keep internal data structures safe, resulting in high communication overhead.

C. *Nested Implementation*

There are two drawbacks with the naive CPU+GPU implementation: a) MPI communication is not guaranteed to happen simultaneously with computation of the interior points on the CPU. b) Computation of the interior points and boundary points do not happen simultaneously on the CPU due to the lack of task parallelism. The use of the CPU's resources is too coarse-grained, leading to much idling, and thus, performance degradation.

The drawbacks to the naive implementation are addressed in an alternative implementation which we call *nested*, due to the use of nested parallelism. We insert task parallelism by dividing the OpenMP threads into two groups. This is done using nested parallel OpenMP regions. To enable nested parallelism support in OpenMP, the omp_set_nested flag must be set to true. Moreover, each parallel region uses the num_threads clause to explicitly specify the number of threads to use within each parallel region. The pseudocode for the nested implementation is shown in Listing 3.

First, two OpenMP threads are spawned in the outer parallel region. Then, each of the two threads starts an inner parallel region with its own thread group. The total number of threads equals the number of physical CPU cores.

The first group is responsible for computation of boundary points, controlling the GPU, and MPI communication, while the second group is responsible for computing the interior points on the CPU. Within each thread group, a new parallel region is created.

In the first thread group, MPI_Irecv and the CUDA kernels for computing boundary points are posted by the master thread, while the other threads perform computation of boundary points on the CPU. Once the master thread has completed its duties, it will join the other threads in its group in computing the boundary points.

Simultaneously, threads in the second group are busy with computing the interior points. Although the use of additional parallel regions implies extra overhead, this approach has the major advantage that thread synchronization in one thread group will not affect the threads in the other group. This means that the necessary synchronization required before MPI communication in the first thread group will not stall the computing threads in the other group.

```
omp_set_nested(true);

#pragma omp parallel default(shared) num_threads(2)
{
  int tid = omp_get_thread_num();

  for (int t = 0; t < iterations; t++) {
    if (tid == 0) {
      #pragma omp parallel num_threads(x)
      {
      #pragma omp master
      {
        for (auto i: direction_vector ex. intra boundary)
          MPI_Irecv(gpu_recv_buffer);
          MPI_Irecv(cpu_recv_buffer);
          ComputePackBoundary<<<dir_stream[i]>>>;
          cudaMemcpyAsync(DeviceToHost, dir_stream[i]);

        ComputeIntraBoundary<<<intra_stream>>>;
        cudaMemcpyAsync(DeviceToHost, intra_stream);

        ComputeInteriorPoints<<<inner_stream>>>;
      }

      for (auto i: direction_vector)
        HostComputePackInterBoundary();
        HostComputeIntraBoundary();

      #pragma omp barrier
      #pragma omp master
      {
        cudaMemcpyAsync(HostToDevice, data_stream);

        for (auto i: direction_vector ex. intra boundary)
          MPI_Isend(cpu_send_buffer);
          cudaStreamSynchronize(dir_stream[i]);
          MPI_Isend(gpu_send_buffer);

       MPI_Waitall();

       for (auto i: direction_vector ex. intra boundary)
         cudaMemcpyAsync(HostToDevice, dir_stream[i]);
         UnpackBoundary<<<dir_stream[i]>>>;
      }

      #pragma omp barrier
      for (auto i: direction_vector ex. intra boundary)
        HostUnpackBoundary();
      } // end omp region for the first thread group
    } else { // tid == 1
      #pragma omp parallel for num_threads(y)
      HostComputeInteriorPoints();
    }

  #pragma omp master
  {
    // cudaDeviceSynchronize and swap pointers
  }
  #pragma omp barrier
  }}
```

Listing 3. A nested MPI + OpenMP + CUDA implementation.

By using the Nvidia Nvprof profiling application, we ac-

TABLE I
AN ARCHITECTURAL OVERVIEW OF THE CLUSTERS USED.

Cluster	**Stampede**	**Wilkes**
Processor	Xeon E5-2680	Xeon E5-2630v2
Architecture	Sandy Bridge-EP	Ivy Bridge-EP
Cores	8	6
Sockets	2	2
# GPUs per node	1	2
Clock freq. [GHz]	2.7	2.6
L3 cache/chip	20 MB	15 MB
Memory size	32 GB	64 GB
Peak DP, GFLOPs	345.6	249.6
Peak BW [GB/s]	102.4	119.4
STREAM [GB/s]	77	72.95
Compiler	icc 13.1	icc 13.1
MPI	mvapich-2 1.9	mvapich-2 2.1

TABLE II
AN ARCHITECTURAL OVERVIEW OF THE ACCELERATOR PRESENT IN BOTH SYSTEMS.

Accelerator	**Nvidia Tesla K20**
Architecture	Kepler
# of SMs	13
Clock freq. [GHz]	0.7
On-chip memory	64 kB
Memory size	5 GB
Peak DP, GFLOPs	1170
Peak BW [GB/s]	208
STREAM [GB/s]	151
Compiler	nvcc 6.0

quired vital profiling data that is rendered in Figure 2. As the illustration shows, in the nested implementation, computation of the interior points on the CPU is overlapped with computation of the boundary points and communication. The profiling also reveals that while the different functions are overlapped on the CPU, this is not the case on the GPU.

Neither the packing nor the unpacking kernels are overlapped with computation of the interior points on the GPU. Because the computation of the boundary points occupies all the SMs on the GPU, the execution of the kernel that computes the interior points is suspended until enough resources become available. Likewise, the computation of the interior points occupies all the SMs and thus delays the start of the unpacking kernels until it relinquishes some SMs.

IV. PERFORMANCE PROJECTIONS

The overall goal of a CPU+GPU implementation is to exploit the entire pool of hardware resources on a compute node in order to solve a given problem as quickly as possible. However, as Tables I and II show, the CPU and the GPU are not equally fast. Thus, a good CPU+GPU implementation must take the different computational speeds into account. Failing to do so will generally lead to a severe load imbalance because the fast GPU will constantly wait for the slow CPU to complete its workload, and thus to poor performance.

Because our numerical kernel is memory-bound, we balance the load according to the realistic memory bandwidth values, CPU_{Bw} in Table I and GPU_{Bw} in Table II, which are measured by the STREAM benchmark.

We use $CPU_{Bw}/(GPU_{Bw} + CPU_{Bw})$ to determine the workload ratio assigned to the CPU, and assign the remaining work to the GPU.

V. EXPERIMENTAL SETUP AND RESULTS

In this section we present experimental results obtained using the GPU-only implementation described in Section II-B, and the two CPU+GPU implementations described in Section III. We used two supercomputers, Stampede and Wilkes to perform our experiments.

TACC Stampede is a primarily Xeon Phi cluster, but a small number of the nodes are equipped Tesla K20m GPUs instead (one per node). While 128 GPU nodes are available, they have been partitioned in such a way that it is not possible to access more than 32 GPU nodes at a time.

Wilkes is a GPU cluster at the University of Cambridge, UK. The cluster consists of 128 nodes, where each node is equipped with two Tesla K20c GPUs. The CPUs on Wilkes are weaker than those found in Stampede. We were unable to use all of the 128 nodes due to several non-operational nodes.

A. Strong Scaling

For the strong scaling experiments, we fix the problem size at $512 \times 512 \times 1024$, while varying the CPU workload ratio from 10% to 25% using a step size of 5%. With the availability of 16 CPU threads and only a single GPU per node on Stampede, one needs to pay extra attention to the nested implementation so that the two thread groups are balanced correctly. In our experiments, the optimum thread distribution was 10 CPU threads on computation of the interior points, and the remaining six threads in the other thread group.

Figures 3(a) and 3(b) show a comparison of the different implementations using 2D and 3D domain decompositions on Stampede, while Figures 4(a) — 4(d) display the same information for the Wilkes cluster. Both of our CPU+GPU implementations scale well, and they are able to outperform the GPU-only implementation. The benefit of the nested implementation is more evident when 16 or more nodes are used.

The difference between the two CPU+GPU implementations is smaller on Wilkes. This is due to the availability of fewer CPU cores, which means a smaller contribution from the CPU segment. Moreover, a prerequisite for achieving good performance using the nested implementation is that the CPU cores in the different thread groups are not overutilized. For example, if one thread group simply has too much work to do, it might lead to threads in the opposite group to idle excessively. Ideally, we would like both thread groups to be perfectly balanced so that they complete their tasks simultaneously.

Results where both GPUs on each Wilkes node are used, are shown in Figures 4(c) and 4(d). In order to make use of both GPUs of Wilkes, the number of MPI processes is doubled on each compute node for the GPU-only and the CPU+GPU implementations. When both GPUs and 2D decomposition are used, the GPU-only implementation is faster than the naive version. The performance degradation observed in both of the

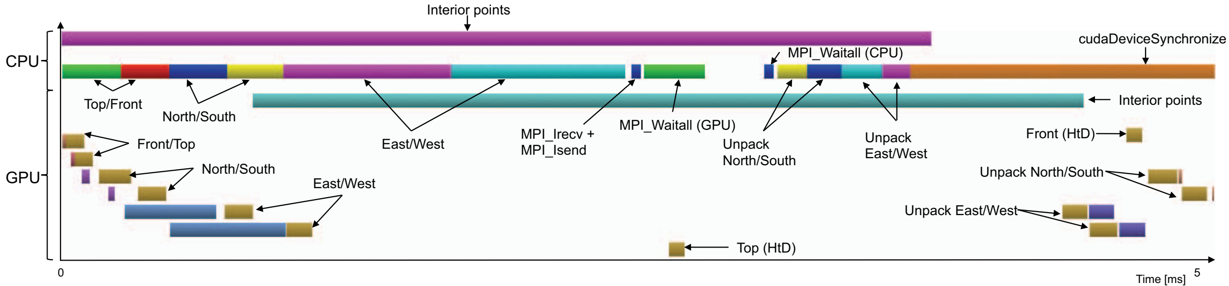


Fig. 2. An actual profiling snapshot for the nested heterogeneous implementation, focusing on a single compute node that has six neighbors.

CPU+GPU implementations can be explained by the increased number of MPI processes per node. On Wilkes, using two MPI processes per compute node means that each process gets access to only six OpenMP threads, which easily leads to a CPU workload that is too high. In spite of this, we observe that both CPU+GPU implementations are marginally faster than the GPU-only implementation when using 3D decomposition.

B. Weak Scaling

For our weak scaling experiments, we keep the problem size fixed at 512^3 for each MPI process. Figures 5(a) — 5(c) show the weak scaling performance on Stampede and Wilkes using 3D domain decomposition. We observe that both CPU+GPU implementations are faster than the GPU-only version, illustrating that the CPU can increase the overall compute performance, and thus improve the overall scalability. Moreover, as the number of nodes increases, the nested version increases its lead compared to the naive implementation. This is due to the nested version's ability to better overlap computation with communication.

C. Sensitivity to Workload Division

For this particular study, we conduct strong scaling experiments, but vary the CPU's workload ratio from 10% to 40%, using a step size of 1%. The same experiment is repeated with an increasing number of nodes. The goal of this study is to find the best CPU workload ratio for a given problem size.

Figure 5(d) shows the impact of the workload ratio on the two evaluation platforms. We observe different optimum workload ratios for different numbers of nodes. This means that predicting an optimum workload ratio beforehand can be quite challenging since the workload ratio is heavily influenced by multiple factors such as problem size, decomposition, and node count. These factors highlight the importance of implementations in which the workload division can be easily adjusted.

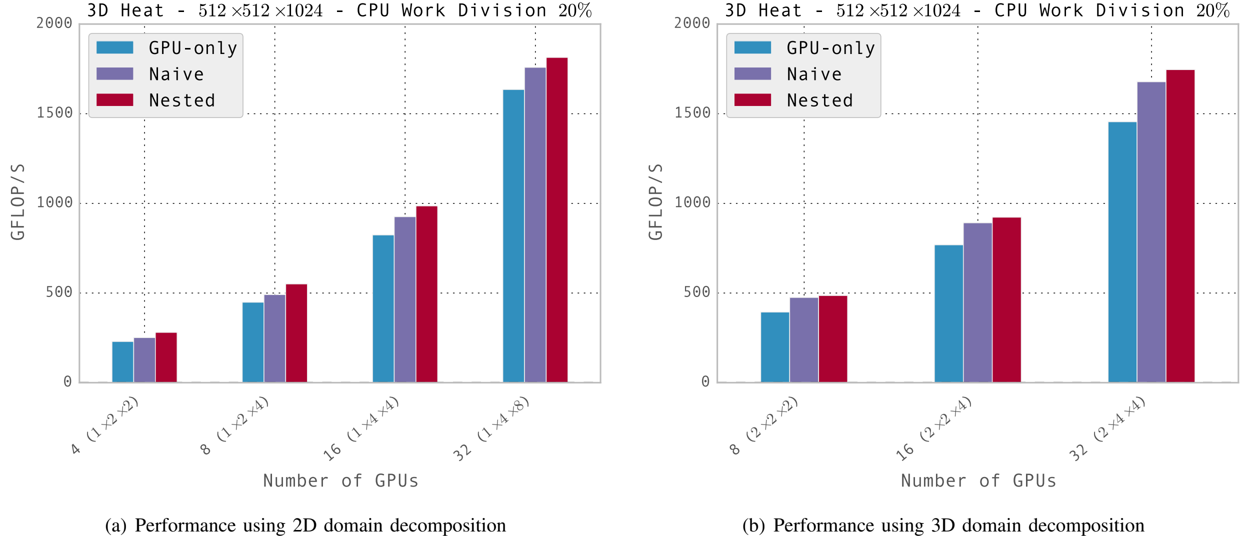
The underlying architecture also plays a pivotal role with respect to the workload ratio. For example, we observed that in the strong scaling studies when the number of subdomains is large, the CPU's share of computation might fit into its L3 cache, which creates a workload imbalance between the CPU and the GPU. In such a scenario, the CPU will wait on the GPU. To address this problem, we increased the CPU's workload.

Based on the performance model discussed in Section IV, we can project the optimum workload ratio. As shown in Figure 5(d), peak performance for Stampede is achieved when the CPU workload division ratio is 23% and 25%, for 8 and 32 nodes, respectively. On Wilkes, the optimal performance is achieved at CPU workload division ratios of 19% and 17%, indicating that our simple performance model is indeed capable of predicting a reasonable initial workload ratio (29% for Stampede and 22% for Wilkes). Moreover, we observe that a good work division strategy is to always give the CPU a slightly smaller workload than suggested by the performance model. The result of a small CPU workload might leave the CPUs underutilized, but it does not degrade the performance as much as a workload that is too big. As mentioned previously, an oversized CPU workload can lead to catastrophic consequences since vital parts of the application such as MPI communication and launching of GPU kernels might be substantially delayed.

VI. Related Work

Although stencil applications using both single- and multi-GPU programming have by now been thoroughly studied [13], [14], [15], fewer works have considered the topic of large-scale stencil CPU+GPU computing. At the broader scale, the topic of CPU+GPU computing has been discussed in many scietific works. For example, Horton et al. [16] updated the MAGMA library so that Cholesky, QR, and LU factorizations can be performed conccurently on the CPU and the GPU. Liu and Luk [17], and Wang et al. [18] have looked at scientific CPU+GPU computation in the context of power efficiency, while Rahimian et al. have developed a large-scale CPU+GPU blood-flow simulator for real-world use [19]. Moreover, a lot of research activities have also focused on dynamic scheduling algorithms for CPU+GPU computations. Prominent examples are DAGuE [20], ClusterSs [21], StarPU [22], and CAP [23]. While being successful in reaching their stated goals, none of these works describes performance projections, domain decomposition, communication handling, and other fundamental programming details with respect to stencil computation.

Langguth and Cai [24] have studied CPU+GPU finite volume computations on unstructured grids using a single



(a) Performance using 2D domain decomposition

(b) Performance using 3D domain decomposition

Fig. 3. Strong scaling performance results measured in GFLOPs on the GPU nodes of Stampede.

compute node. Similarly, Wang and JaJa [25] have focused on accelerating an FFT-based Poisson solver on a single compute node. Both report that their CPU+GPU implementations outperform the corresponding GPU-only implementations.

Venkatasubramanian and Vuduc [26] present a small-scale CPU+GPU implementation together with a performance model for a 2D Poisson equation on a square domain using Jacobi's method. Thanks to algorithms such as chaotic relaxation and asynchroneous iteration, CPU-GPU synchronization are minimized. The authors report that their single node CPU+GPU implementations are able to outperform the corresponding GPU-only implementation by 8% and 11% (depending on the system). However, the authors are unable to observe the same performance gain for their CPU+GPU implementation when moving on to multiple nodes. The big performance difference between the GPU and the CPU is given as the reason. Despite a slower multi-node CPU+GPU implementation, the authors conclude that CPU+GPU implementations will play an important role in future CPU-GPU architectures.

To the best of our knowledge, only the works by Shimokawabe et al. [2], Yang et al. [27], and Langguth et al. [28] compare directly with our work, as they perform CPU+GPU computation at a larger scale.

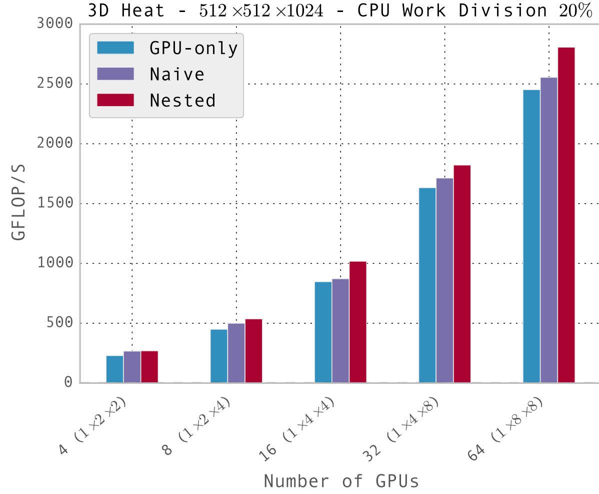
Our work differs from [2] in multiple ways. First of all, in [2], due to problems with the memory accesses on the GPU, only a global 2D domain decomposition is used. Our work, on the other hand has demonstrated good scaling for both 2D and 3D decompositions. Another crucial difference between our work and the study by Shimokawabe et al. is CPU utilization. In Shimokawabe et al., the CPU is used only for lightweight boundary computations, while interior point computations are left to the GPU. When Shimokawabe et al. attempted to allow the CPUs to handle a larger part of the computation they observed that the CPU became a bottleneck. In our strategy, the CPU computes a slice of the total domain that is commensurate with its computational speed.

In [27], a CPU+GPU implementation is developed to perform atmospheric simulations on a cubed-sphere domain. Similarly to [2], the implementation presented in [27] uses the CPU for boundary computation only. This means that the powerful CPU cores are only used for a very small amount of computation, while all the remaining computations occur on the GPU. Moreover, the implementation presented in [27] is unable to overlap CPU→GPU and GPU→CPU data transfers.

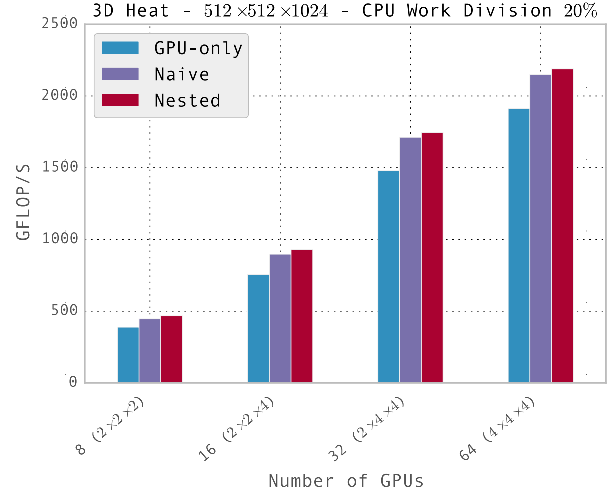
Our task-based approach is more fine-grained because the CPU threads are divided into two separate groups so that CPU idling is minimized. We use only a handful of threads for boundary point computation, and the remaining CPU threads are used for computing the interior points. Because communication happens in one thread group, we are also able to completely mask intra-node CPU ↔ GPU data transfers.

The work originally presented in [24], was further extended by Langguth et al. in [28], so that CPU+GPU computations can be executed across multiple nodes. Because the computational domain is different to the one represented in this paper, it is difficult to make a fair and direct comparision. However, we note that the implementation presented in [24] performs a manual thread to core assignment, which effectively means that OpenMP directives can no longer be used. This means that the abstraction layer that OpenMP otherwise provides, is peeled off, and as a concequence, the user is exposed to many low-level programming details.

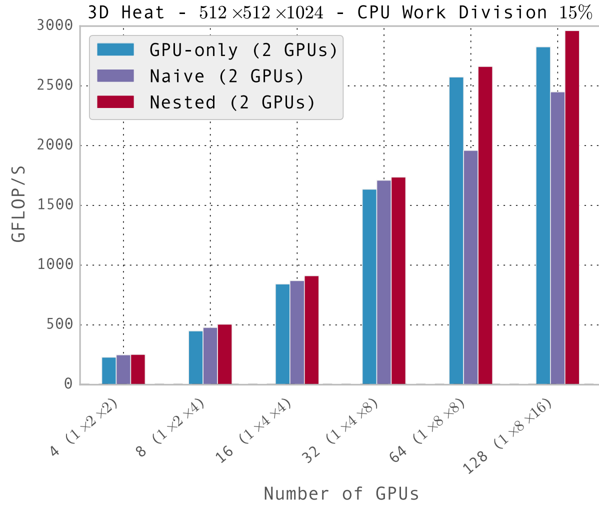
Due to the use of OpenMP's own nested parallelism capabilities, we are able to use OpenMP's directives in our code, without the need to for example manually divide the iteration space when performing computations. This is not only easier, but that also provides better portability when moving the code



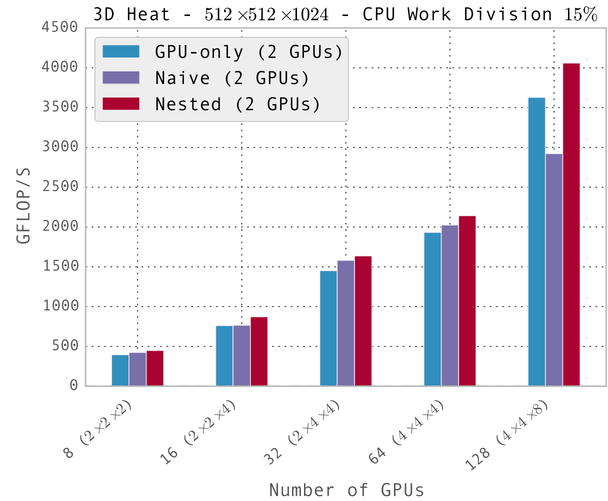
(a) Performance using 2D domain decomposition and one GPU per node



(b) Performance using 3D domain decomposition and one GPU per node



(c) Performance using 2D domain decomposition and two GPUs per node



(d) Performance using 3D domain decomposition and two GPUs per node

Fig. 4. Strong scaling performance results measured in GFLOPs on the GPU nodes of Wilkes.

from one cluster to another.

VII. Conclusions

In this paper, we have presented and evaluated two CPU+GPU implementations. We have demonstrated that by letting the CPU take part in the computations, the overall solution time for a stencil application on two different GPU clusters is reduced. At the same time, we have made effective use of all the resources available. We have also introduced a simple performance model for CPU+GPU implementations, which can provide guidance for workload division.

Acknowledgments

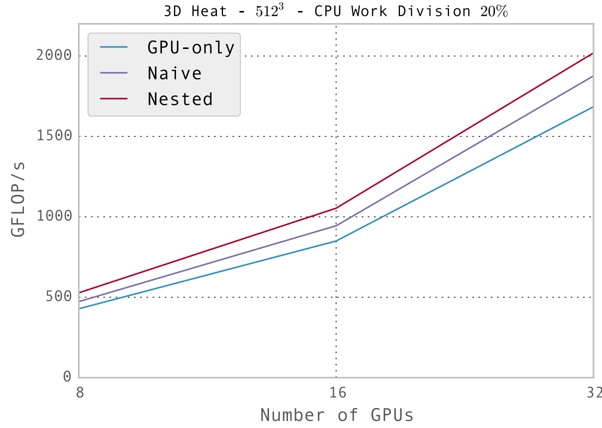
This work was supported by the FriNatek program of the Research Council of Norway, through grant No. 214113/F20. The authors thank High Performance Computing Service at the University of Cambridge, UK, and TACC at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

References

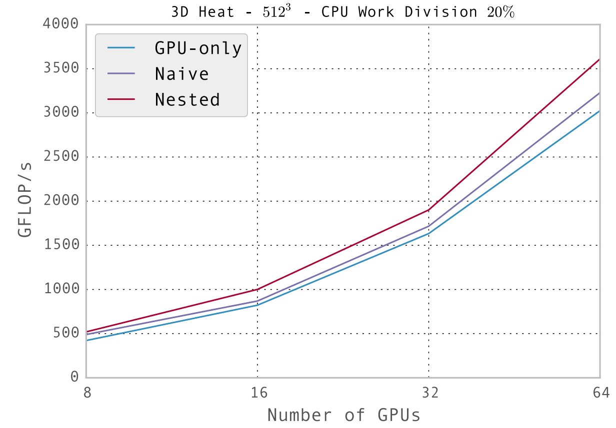
[1] J. Zhou, Y. Cui, E. Poyraz, D. J. Choi, and C. C. Guest, "Multi-GPU implementation of a 3D finite difference time domain earthquake code on heterogeneous supercomputers," in *Proceedings of the International Conference on Computational Science*, Jun. 2013, pp. 1255–1264.

[2] T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka, "Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2011, pp. 3:1–3:11.

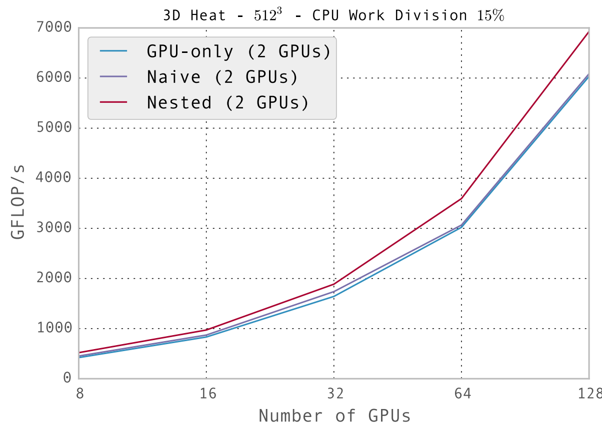
[3] Oak Ridge Leadership Computing Facility, "Summit," https://olcf.ornl.gov/summit/, [Online; accessed 29-May-2015].



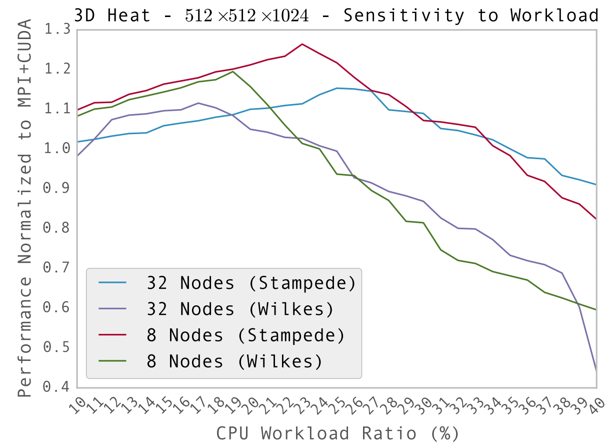
(a) Weak scaling on Stampede using 3D domain decomposition



(b) Weak scaling on Wilkes, one GPU per node, 3D decomposition



(c) Weak scaling on Wilkes, two GPUs per node, 3D decomposition



(d) CPU workload division sensitivity using 2D domain decomposition

Fig. 5. Weak scaling results on Stampede and Wilkes.

[4] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[5] NVIDIA, "NVIDIA GPUDirect," https://developer.nvidia.com/gpudirect, [Online; accessed 05-March-2015].

[6] A. Schäfer and D. Fey, "High performance stencil code algorithms for GPGPUs," in *Proceedings of 2011 International Conference on Computational Sciences (ICCS)*, vol. 4, Jun. 2011, pp. 2027–2036.

[7] H. Su, N. Wu, M. Wen, C. Zhang, and X. Cai, "On the GPU performance of 3D stencil computations implemented in OpenCL," in *Proceedings of the 28th International Supercomputing Conference*, vol. 7905, Jun. 2013, pp. 125–135.

[8] F. B. Kjolstad and M. Snir, "Ghost cell pattern," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, Mar. 2010, pp. 4:1–4:9.

[9] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters," *Computer Science-Research and Development*, vol. 26, no. 3-4, pp. 257–266, Apr. 2011.

[10] M. Sourouri, T. Gillberg, S. Baden, and X. Cai, "Effective multi-GPU communication using multiple CUDA streams and threads," in *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, Dec. 2014, pp. 981–986.

[11] J. Treibig, G. Wellein, and G. Hager, "Efficient multicore-aware parallelization strategies for iterative stencil computations," *Journal of Computational Science*, vol. 2, no. 2, pp. 130–137, May 2011.

[12] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3D scientific computations," in *Proceedings of the 2000 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2000.

[13] G. Zumbusch, "Vectorized higher order finite difference kernels," in *Proceedings of the 11th International Conference on Applied Parallel and Scientific Computing*, Jun. 2013, pp. 343–357.

[14] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-2*, pp. 79–84, Mar. 2009.

[15] E. H. Phillips and M. Fatica, "Implementing the Himeno benchmark with CUDA on GPU clusters," in *Proceedings of 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1–10.

[16] M. Horton, S. Tomov, and J. Dongarra, "A class of hybrid LAPACK algorithms for multicore and GPU architectures," in *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, Jul. 2011, pp. 150–158.

[17] Q. Liu and W. Luk, "Heterogeneous systems for energy efficient scientific computing," in *Proceedings of the 8th International Symposium on Reconfigurable Computing: Architectures, Tools and Applications*, Mar. 2012, vol. 7199, pp. 64–75.

[18] G. Wang and X. Ren, "Power-efficient work distribution method for CPU-GPU heterogeneous system," in *Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on*, Sep.

2010, pp. 122–129.
[19] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros, "Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2010, pp. 1–11.
[20] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," *Parallel Computing*, vol. 38, no. 1–2, pp. 37–51, 2012.
[21] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta, "ClusterSs: A task-based programming model for clusters," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, Jun. 2011, pp. 267–268.
[22] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput.: Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011.
[23] Z. Wang, L. Zheng, Q. Chen, and M. Guo, "CPU+GPU scheduling with asymptotic profiling," *Parallel Computing*, vol. 40, no. 2, pp. 107–115, Feb. 2014.
[24] J. Langguth and X. Cai, "Heterogeneous CPU-GPU computing for the finite volume method on 3D unstructured meshes," in *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, Dec. 2014, pp. 191–199.
[25] J. Wu and J. JaJa, "High performance FFT based poisson solver on a CPU-GPU heterogeneous platform," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, May 2013, pp. 115–125.
[26] S. Venkatasubramanian and R. W. Vuduc, "Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems," in *Proceedings of the 23rd International Conference on Supercomputing*, Jun. 2009, pp. 244–255.
[27] C. Yang, W. Xue, H. Fu, L. Gan, L. Li, Y. Xu, Y. Lu, J. Sun, G. Yang, and W. Zheng, "A peta-scalable CPU-GPU algorithm for global atmospheric simulations," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2013, pp. 1–12.
[28] J. Langguth, M. Sourouri, G. T. Lines, S. B. Baden, and X. Cai, "Scalable heterogeneous CPU-GPU computations for unstructured tetrahedral meshes," *Micro, IEEE*, vol. 35, no. 4, pp. 6–15, Jul. 2015.