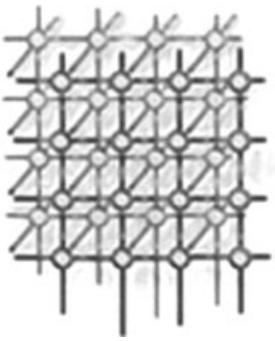
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2009; **21**:2400–2437
Published online 20 July 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1462



Exploiting graphical processing units for data-parallel scientific applications

A. Leist, D. P. Playne and K. A. Hawick*, †

Computer Science, Institute of Information and Mathematical Sciences, Massey University, Albany, Auckland, New Zealand

SUMMARY

Graphical processing units (GPUs) have recently attracted attention for scientific applications such as particle simulations. This is partially driven by low commodity pricing of GPUs but also by recent toolkit and library developments that make them more accessible to scientific programmers. We discuss the application of GPU programming to two significantly different paradigms—regular mesh field equations with unusual boundary conditions and graph analysis algorithms. The differing optimization techniques required for these two paradigms cover many of the challenges faced when developing GPU applications. We discuss the relevance of these application paradigms to simulation engines and games. GPUs were aimed primarily at the accelerated graphics market but since this is often closely coupled to advanced game products it is interesting to speculate about the future of fully integrated accelerator hardware for both visualization and simulation combined. As well as reporting the speed-up performance on selected simulation paradigms, we discuss suitable data-parallel algorithms and present code examples for exploiting GPU features like large numbers of threads and localized texture memory. We find a surprising variation in the performance that can be achieved on GPUs for our applications and discuss how these findings relate to past known effects in parallel computing such as memory speed-related super-linear speed up. Copyright © 2009 John Wiley & Sons, Ltd.

Received 2 December 2008; Revised 15 April 2009; Accepted 19 April 2009

KEY WORDS: data-parallelism; GPUs; field equations; graph algorithms; CUDA

1. INTRODUCTION

Arguably, Data-Parallel computing went through its last important 'golden age' in the late 1980s and early 1990s [1]. That era saw a great many scientific applications ported to the then important massively parallel and data-parallel supercomputer architectures such as the AMT DAP [2] and the

*Correspondence to: K. A. Hawick, Institute of Information and Mathematical Sciences, Massey University, North Shore 102-904, Auckland, New Zealand.
†E-mail: k.a.hawick@massey.ac.nz

Copyright © 2009 John Wiley & Sons, Ltd.

Thinking Machines Connection Machine [3,4]. The era also saw successful attempts at portable embodiments of data-parallel computing ideas into software libraries, kernels [5,6] and programming languages [7–10]. Much of this software still exists and is used, but was perhaps somewhat overshadowed by the subsequent era of coarser-grained multi-processor computer systems – perhaps best embodied by cluster computers, and also by software to exploit them such as message-passing systems like MPI [11]. We have since seen some blending together of these ideas in software languages and systems like OpenMP [12]. We are currently in an era where wide-ranging parallel computing ideas are being exploited successfully in multi-core central processing units (CPUs) and of course in graphical processing units (GPUs)—as discussed in this article.

Graphical Processing Units (GPUs) [13] have emerged in the recent years as an attractive platform for optimizing the speed of a number of application paradigms relevant to the games and computer-generated character animation industries [14]. While chip manufacturers have been successful in incorporating ideas in parallel programming into their family of CPUs, inevitably due to the need for backwards compatibility, it has not been trivial to make major advances with complex backward compatible CPU chips. The concept of the GPU as a separate chip with fewer legacy constraints is perhaps one explanation for the recent dramatic improvements in the use of parallel hardware and ideas at the chip level.

In principle, GPUs can of course be used for many parallel applications in addition to their original intended purpose of graphics processing algorithms. In practice however, it has taken some relatively recent innovations in high-level programming language support and accessibility for the wider applications programming community to consider using GPUs in this way [15–17]. The term general-purpose GPU programming [14] has been adopted to describe this rapidly growing applications level use of the GPU hardware. Exploiting parallel computing effectively at an applications level remains a challenge for programmers. In this article we focus on two broad application areas—partial differential field equations and graph combinatorics algorithms and explore the detailed coding techniques and performance optimization techniques offered by the Compute Unified Device Architecture (CUDA) [17] GPU programming language. NVIDIA's CUDA has emerged in the recent months as one of the leading programmer tools for exploiting GPUs. It is not the only one [18–21], and inevitably it builds on many past important ideas, concepts and discoveries in parallel computing.

In the course of our group's research on complex systems and simulations we have identified a number of scientific application paradigms such as those based on particles, field models, and on graph network representations [22]. The use of GPUs for paradigm of N-Body particle simulation has already been well reported in the literature [23,24], and hence in our present article we focus on complex field-based and graph-based scientific application examples.

In Section 2 we present some technical characteristics of GPUs, general ideas and explanatory material on how some GPUs work and how they can be programmed with a language like CUDA. We present some of our own code fragments by the way of explanation for exploiting some of the key features of moderns GPUs such as the different sorts of memory and the use of and groupings of multiple parallel executing threads.

In Section 3 we present some ideas, code examples and performance measurements for a specific partial differential equation simulation model—the Cahn–Hilliard (CH) field equation as used for simulating phase separation in material systems. We expected that this would parallelize well, and although we can obtain good results in the right system size regime, there are interesting and

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

surprising variations that arise from the nature of a practical simulation rather than a carefully chosen toy or benchmark scenario.

A problem area that we expected to be much more difficult to achieve good performance in is that of graph combinatoric problems. In Section 4 we focus on the all-pairs shortest path problem in graph theory as a highly specific algorithm used in searching, navigation and other problems relevant to the games and computer-generated character generation industries.

In both these problem areas we have been able to use the power of GPU programming to obtain some scientifically useful results in complex systems, scaling and growth research. In this present article we focus on how the parallel coding was done and how the general ideas might be of wider use to the scientific simulations programming community.

We offer some ideas and speculations as to the future directions for GPUs and their exploitation at the software level for scientific applications in Section 6.

2. GPU FEATURES

GPU architectures contain many processors which can execute instructions in parallel. These parallel processors provide the GPU with the high-performance parallel computational power. To make use of this parallel computational power, a program executing on the GPU must be decomposed into many parallel threads. Parallel decomposition of computational problems is a widely researched topic and is not discussed in detail here. However, one specific condition of parallel decomposition for GPUs is the number of threads the problem should be split into. As GPUs are capable of executing a large number of threads simultaneously, it is generally advantageous to have as many threads as possible. A large number of threads allow the GPU to maximize the efficiency of execution.

2.1. NVIDIA Multiprocessor Architecture

To maximize the performance, it is important to maximize the utilization of the available computing resources as described in the CUDA Programming Guide (PG) [25]. This means that there should be at least as many thread blocks as there are streaming multiprocessors (SM) in the device. And since each SM consists of 8 scalar processor cores—also called Streaming Processors (SP)—a GeForce GTX 280 can execute a total of 240 threads simultaneously. However, the multiprocessor SIMT unit groups the threads in a block into so-called *warps*, each consisting of 32 parallel threads that are created, managed, scheduled and executed together. A warp is the smallest unit of execution in CUDA. Thus, at least 960 threads should be created to keep a GTX 280 busy. (See PG Chapters 3.1 and 5.2.)

Furthermore, to enable the thread scheduler to hide latencies incurred by thread synchronizations (4 clock cycles), issuing memory instruction for a warp (4 clock cycles), and especially by accessing the device memory (400–600 clock cycles), an even larger number of active threads is needed per multiprocessor. As listed in Table I, the latest generation of GeForce GPUs supports up to 30 720 active threads (30 SMs $\times$ 32 warps $\times$ 32 threads per warp), significantly improving the thread scheduler's capability to hide latencies compared with the 12 288 active threads supported by the previous generations. Memory-bound kernels, in particular, require a large number of threads to

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Table I. Device capabilities of CUDA-enabled GPUs from different generations along with section references where we discuss their role in our applications.

	GTX 280	9800 GTX	8800 GTX	See Section
Texture processing clusters (TPC)	10	8	8	2.1
Streaming multiprocessors (SM) per TPC	3	2	2	2.1
32-bit streaming processors (SP) per SM	8	8	8	2.1
64-bit SPs per SM	1	N/A	N/A	2.1
Total 32-bit SPs	240	128	128	2.1
Total 64-bit SPs	30	N/A	N/A	2.1
Graphics Clock (MHz)	602	675	575	
SP Clock (MHz)	1296	1688	1350	
Single-precision peak performance (GFLOP)	933	648	518	
Double-precision peak performance (GFLOP)	78	N/A	N/A	
Memory clock (MHz)	1107	1100	900	
Memory interface width (bit)	512	256	384	
Memory bandwidth (GB/sec)	141.7	70.4	86.4	
Standard device memory config. (MB)	1024	512	768	2.2.1
Compute capability	1.3	1.1	1.0	
Registers (32-bit) per SM	16384	8192	8192	2.2.2
Shared memory per SM (KB)	16	16	16	2.2.3
Total constant memory (KB)	64	64	64	2.2.5
Constant memory cache per SM (KB)	8	8	8	2.2.5
Texture memory cache per SM (KB)	6−8	6-8	6-8	2.2.4
Warp Size (Threads)	32	32	32	2.1
Max. active thread blocks per SM	8	8	8	2.1
Max. active warps per SM	32	24	24	2.1
Max. active threads per SM	1024	768	768	2.1
Total active threads per Chip	30720	12288	12288	2.1

Relevant features without explicit section references are explained and discussed within this present section.

maximize the utilization of the computing resources provided by the GPU. (See PG Chapters 5.1.1 and 5.2.)

There are several possible ways to organize the threads once the thread count exceeds one warp per block per SM. Either the number of blocks per multiprocessor, the block size, or both can be increased, up to the respective limits. Increasing the number of active blocks on a multiprocessor to at least twice the number of multiprocessors allows the thread scheduler to hide thread synchronization latencies and also device memory reads/writes if there are not enough threads per block to cover these. The shared memory and registers of an SM are shared between all active blocks that reside on this multiprocessor. Thus, for two thread blocks to simultaneously reside on one multiprocessor, each of the blocks can use at most $\frac{1}{2}$ of the available resources. (See PG Chapter 5.2.)

But having even more than the maximum number of total active thread blocks on the device can be a good idea, as it will allow the kernel to scale to future hardware with more SMs or with more local resources available to each SM. If scaling to future device generations is desired, then it is advisable to create hundreds or even thousands of thread blocks. (See PG Chapter 5.2.)

Larger block sizes, on the other hand, have advantages when threads have to communicate and when the execution has to be synchronized at certain points. Threads within the same block can

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

communicate through shared memory and call the __syncthreads() barrier function, which guarantees that none of the threads proceeds before all other threads have also reached the barrier. The block size should always be a multiple of the warp size to avoid partially filled warps and because the number of registers used per block is always rounded up to the nearest multiple of 32. Choosing multiples of 64 is even better, as this helps the compiler and thread scheduler to achieve best results avoiding register memory bank conflicts. (See PG Chapters 2.1, 3.1, 5.1.2.5, 5.1.2.6 and 5.2.)

The maximum block size is 512 threads for all CUDA-enabled GPUs released so far. Thus, to reach the maximum number of active threads supported on a GTX 280, both the number of blocks and the block size have to be increased. The multiprocessor occupancy is a metric that states this ratio of active threads to the maximum number of active threads supported by the hardware. The definition is given in as follows (see PG Chapters 2.1 and 5.2):

$$Occupancy = \frac{\text{number of active warps per multiprocessor}}{\text{maximum number of active warps per multiprocessor}} \tag{1}$$

While trying to maximize the multiprocessor occupancy is generally a good idea, a higher occupancy does not always mean a better performance. The higher the value, the better the scheduler can hide latencies, thus making this metric especially important for memory-bound kernels. However, other factors that can have a large impact on the performance are not reflected, like the ability of more threads to communicate via shared memory when the block size is larger. Thus, if possible, the performance should always be measured for various block sizes to determine the best value for a specific kernel.

The number of active threads and thread blocks per SM depends on the block size and the shared memory and register usage per block. These are the input values to a spreadsheet provided by NVIDIA® as part of the CUDA SDK, which calculates the respective multiprocessor occupancy and how varying any of the input values affects the results. It is a valuable tool to increase the multiprocessor occupancy. However, when the input values depend on runtime parameters, then it can be desirable to be able to automatically determine the block size with the highest multiprocessor occupancy at runtime. This is exactly what the function in Listing 1 does.

2.2. Memory

GPUs contain several types of memory that are designed and optimized for different purposes. Although the optimization features for the memory types are focused towards the specific tasks of the graphics pipeline, their features can be utilized for other purposes. Using the type of memory best suited to the task can provide significant performance benefits. The memory access patterns used to retrieve/store data from these memory locations are equally important as the type of memory used. Correct access patterns are critical to the performance of the GPU program. CUDA-enabled GPUs provide access to six different types of memory that are stored both on-chip (within a multiprocessor) and on the device.

Global Memory is part of the device memory. It is the largest memory on the GPU and is accessible by every thread in the program. However, it has no caching and has the slowest access time of any memory type—approximately 400–600 clock cycles. Lifetime = application. (See PG Chapter 5.1.2.1.)

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
/**
 * Returns the number of threads per block that gives the highest
 * multiprocessor occupancy. Prefers larger thread blocks over smaller
 * ones with the same occupancy. NOTE: Always returns 32 if compiled
 * for device emulation (the -deviceemu flag is passed to the
 * compiler).
 *
 * Params: deviceProp The device properties.
 *         registers The number of registers needed by each thread.
 *         sharedMemPerThread The shared memory needed per thread (bytes).
 *         sharedMemPerBlock The shared memory needed per block, independent
 *                           of the number of threads (bytes).
 */
int maxOccupancy(cudaDeviceProp* deviceProp, int registers,
                 int sharedMemPerThread, int sharedMemPerBlock) {
// the values for compute capability 1.0
int limitBlocksPerMP = 8;
int limitWarpsPerMP = 24;
// adjust these base values according to the actual compute capability
if (deviceProp->major >= 1 && deviceProp->minor >= 2) { // 1.2
  limitWarpsPerMP = 32;
}

int limitThreadsPerWarp = deviceProp->warpSize;
size_t limitSharedMemPerMP = deviceProp->sharedMemPerBlock;
int limitRegistersPerMP = deviceProp->regsPerBlock;

double maxOccupancy = 0.0;
int nThreadsMaxOccupancy = 0;
for (int nThreads = 512; nThreads >= 32; nThreads -= 32) {
  // the shared memory needed for this block size
  int sharedMem = sharedMemPerBlock + sharedMemPerThread * nThreads;

  int myWarpsPerBlock = ceil((double)nThreads / (double)limitThreadsPerWarp);
  int mySharedMemPerBlock = nextMultipleOfPower2(sharedMem, 512);
  int myRegsPerBlock = nextMultipleOfPower2(myWarpsPerBlock * 2, 4)
    * 16 * registers;

  int limitedByMaxWarpsPerMP = min(limitBlocksPerMP,
                                   (int)floor((double)limitWarpsPerMP
                                              / (double)myWarpsPerBlock));

  int limitedBySharedMemPerMP;
  if (sharedMem > 0) {
    limitedBySharedMemPerMP = floor((double)limitSharedMemPerMP
                                    / (double)mySharedMemPerBlock);
  } else {
    limitedBySharedMemPerMP = limitBlocksPerMP;
  }
```

Listing 1. Implementation of the occupancy calculation to determine the maximum multiprocessor occupancy on the available graphics hardware for a given register and shared memory usage.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
    int limitedByRegistersPerMP;
    if (registers > 0) {
      limitedByRegistersPerMP = floor((double)limitRegistersPerMP
                                      / (double)myRegsPerBlock);
    } else {
      limitedByRegistersPerMP = limitBlocksPerMP;
    }

    int activeThreadBlocksPerMP = min(limitedByMaxWarpsPerMP,
                                      min(limitedBySharedMemPerMP,
                                          limitedByRegistersPerMP));
    int activeWarpsPerMP = activeThreadBlocksPerMP * myWarpsPerBlock;
    double occupancy = (double)activeWarpsPerMP / (double)limitWarpsPerMP;

    if (occupancy > maxOccupancy) {
      maxOccupancy = occupancy;
        nThreadsMaxOccupancy = nThreads;
      }
    }

#ifdef __DEVICE_EMULATION__
  nThreadsMaxOccupancy = 32;
#endif

    return nThreadsMaxOccupancy;
}
```

Listing 1. *Continued.*

Registers are stored on-chip and are the fastest type of memory. The registers are used to store the local variables of a single thread and can only be accessed by that thread. Lifetime = thread. (See PG Chapter 5.1.2.6.)

Local Memory does not represent an actual physical area of memory, instead it is a section of device memory used when the variables of a thread do not fit within the registers available. Lifetime = thread. (See PG Chapters 4.2.2.4 and 5.1.2.2.)

Shared Memory is fast on-chip memory that is shared between all of the threads within a single block. It can be used to allow the threads within a block to co-operate and share information. Lifetime = block. (See PG Chapters 4.2.2.3 and 5.1.2.5.)

Texture Memory space is a cached memory region of global memory. Every SM has its own texture memory cache on-chip. Device memory reads are only necessary on cache misses, otherwise a texture fetch only costs a read from the texture cache. Lifetime = application. (See PG Chapters 5.1.2.4 and 5.4.)

Constant Memory space is a cached, read-only region of device memory. Like the texture cache, every SM has its own constant memory cache on-chip. Device memory reads are only necessary on cache misses, otherwise a constant memory read only costs a read from the constant cache. Lifetime = application. (See PG Chapters 5.1.2.3.)

The access times to these types of memory depend on the access pattern used. The access patterns and optimal uses for the types of memory are now discussed.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

2.2.1. *Global Memory Access*

Global memory has the slowest memory access times of any type of memory (400–600 clock cycles). Although the actual time taken to access global memory cannot be reduced, it is possible to increase the overall performance by combining the global memory transactions of the threads within the same half-warp into one single memory transaction. This process is known as coalescing and can greatly increase the performance of a GPU program.

Global memory transactions can be coalesced if sequential threads in a half-warp access 32-, 64- or 128-bit words from sequential addresses in memory and the addresses are aligned to a memory block. These sequential addresses depend on the size of the words accessed, i.e. each address must be the previous address plus the size of the word. 32- and 64-bit transactions are coalesced into a single transaction of 64-/128-bytes and 128-bit transactions are coalesced into two 128-byte transactions. Coalesced memory access provides a large performance improvement over the 16 global memory transactions required for non-coalesced memory access. (See PG Chapter 5.1.2.1 for the detailed requirements and the differences between compute capabilities.)

Sometimes it is not possible to access global memory in a coalesced fashion. For instance, let us assume that we have an array of sub-arrays of varying lengths. Storing this structure as a two-dimensional CUDA array would waste a lot of space if the array lengths vary considerably. Thus, we store it as a one-dimensional array with the index of the first element of each sub-array being pointed to by a second array. Furthermore, the first sub-array element specifies the length of the respective array. If each thread iterates over one of these sub-arrays, then the access is uncoalesced, resulting in many inefficient memory transactions.

One way to improve the use of the memory bandwidth is to use a built-in vector type like int4 as the type of the array. This is simply a correctly aligned struct of 4 integer values. We always group 4 elements of a sub-array into one such struct. Every sub-array begins with a new struct, thus some space is wasted if the length of a sub-array plus one for the length element is not a multiple of 4. The advantages are that now four values (i.e. 16 bytes) can be read in one memory transaction instead of only one and only a single instruction is required to write these values to registers. Listing 2 illustrates how a thread can iterate over a sub-array starting at index *structArrayIdx*.

One thing that should be considered when iterating over arrays of varying length is warp divergence. That is, threads within a warp taking diverging branches, or in this case iterating over a different number of elements. Warp divergence can reduce the performance significantly, as the SIMT-model forces diverging branches within a warp to be serialized. In this example, branch divergence can be reduced if the sub-arrays are organized so that the threads of the same warp process arrays of approximately the same length.

2.2.2. *Registers and local memory*

Registers and local memory are used to store the variables used by the threads of the program. The very fast registers are used first, however, if the variables of the threads cannot fit into the registers then local memory must be used. Fetching values from local memory requires access to the global memory which is always coalesced but is much slower than accessing the registers. Therefore, it is desirable to structure the code such that all the local variables can fit within the registers. It is

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
int4 myStruct = structArray[structArrayIdx];
int nIterations = myStruct.x;
for (int idx = 1; idx <= nIterations; idx++) {
  int structIdx = idx % 4;
  int value;
  if (structIdx == 0) {
    myStruct = structArray[++structArrayIdx]; // load the next vector
    value = myStruct.x;
  } else if (structIdx == 1) {
    value = myStruct.y;
  } else if (structIdx == 2) {
    value = myStruct.z;
  } else if (structIdx == 3) {
    value = myStruct.w;
  }
  ...
}
```

Listing 2. Using the built-in vector types to take advantage of the full memory bandwidth.

often more efficient to perform the same calculation several times within a thread to avoid storing values in local memory.

As long as no register read-after-write dependencies or register memory bank conflicts occur, accessing a register costs zero extra clock cycles per instruction. Delays due to read-after-write dependencies can be ignored if at least 192 active threads per multiprocessor exist to hide them. Registers are used to store the local variables of a thread and cannot be used to share data between threads.

2.2.3. Shared memory

Shared memory can be used to reduce the amount of global memory access required. If two or more threads within the same block must access the same piece of data, shared memory can be used to transfer data between them. The remaining threads can then access this value from the faster shared memory instead of through global memory accesses.

Shared memory is divided into 16 banks (compute capability 1.x), which are equally sized memory modules that can be accessed simultaneously. Successive 32-bit words are assigned to successive banks and each bank has a bandwidth of 32 bits per two clock cycles. A half-warp consisting of 16 threads and executing in two clock cycles only needs one shared memory request to read or write data as long as no bank conflicts occur. Thus, it is important to schedule shared memory requests so as to minimize bank conflicts. Writes to shared memory are guaranteed to be visible by other threads after a __syncthreads() call.

Shared memory can either be allocated statically or dynamically. The total size of dynamically allocated shared memory is determined at launch time as an optional parameter to the execution configuration of the kernel. If more than one variable points to addresses within the memory range allocated in this fashion, then this has to be done by using appropriate offsets as illustrated in Listing 3. (See PG Chapters 4.2.2.3 and 4.2.3.)

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
// all variables declared in this fashion start at the
// same address in shared memory
extern __shared__ char dynamicSharedMem[];

__global__ void myKernel() {
  __shared__ int staticSharedMemory[256];

  // split the dynamically allocated shared memory up into an int
  // array of size 2 * blockSize and a float array of size blockSize
  int* sharedInts = (int*) dynamicSharedMem;
  float* sharedFloats = (float*) &sharedInts[blockDim.x * 2];

  // avoid bank conflicts caused by access patterns like the
  // following (stride = 2)
  sharedInts[threadIdx.x * 2] = 0;
  sharedInts[threadIdx.x * 2 + 1] = 0;

  // use this access pattern instead (stride = 1)
  sharedInts[threadIdx.x] = 0;
  sharedInts[blockDim.x + threadIdx.x] = 0;
}

// call the kernel with the required dynamic shared memory
// as third parameter to the execution configuration
size_t sharedMemBytes = blockSize * 2 * sizeof(int)
                      + blockSize * sizeof(float);
myKernel<<<gridSize, blockSize, sharedMemBytes>>>();
```

Listing 3. Static and dynamic allocation of shared memory.

As mentioned before, bank conflicts should be avoided when accessing shared memory. One way to make sure that no bank conflicts occur is to use a stride of odd length when accessing 32-bit words. (See PG Chapter 5.1.2.5.)

If a global counter variable is used that is updated by every thread, then many uncoalesced writes to global memory can be reduced to a single write per thread block. Instead of directly updating the global variable, the threads in the same thread block atomically update a local counter in shared memory, which is only written back to the global counter once all threads in the block have reached a thread barrier. Listing 4 illustrates this optimization technique. Compute capability 1.2 or above is required for the atomic operations on shared memory.

2.2.4. Texture references

Texture memory space is a cached memory region of global memory. Every SM has its own texture memory cache with a size of 6–8 KB on-chip. Device memory reads are only necessary on cache misses, otherwise a texture fetch only costs a read from the texture cache. It is designed for 2D spatial locality and for streaming fetches with a constant latency. The texture cache is not kept coherent with global memory writes within the same kernel call. Writing to global memory that is accessed via a texture fetch within the same kernel call returns undefined data. (See PG Chapters 5.1.2.4 and 5.4.)

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
__shared__ int counterChanges;
if (threadIdx.x == 0) {
  counterChanges = 0; // initialise
}
__syncthreads();
...
atomicAdd(&counterChanges, 1); // modify the counter in shared memory
...
__syncthreads();
if (threadIdx.x == 0) { // write the changes back to global memory
  atomicAdd(globalCounter, counterChanges);
}
```

Listing 4. Using shared memory to reduce the number of writes to a global counter variable.

```
texture<int, 1, cudaReadModeElementType> textureRef;
...
const int prefetchCount = 5;
__shared__ int prefetchCache[prefetchCount * THREADS_PER_BLOCK];
for (int idx = 0; idx < nIterations; idx++) {
  int value;
  int prefetchIdx = idx % (prefetchCount + 1);
  if (prefetchIdx == 0) {
    value = tex1Dfetch(textureRef, beginIdx + idx);
    #pragma unroll
    for (int i = 0; i < prefetchCount; i++) {
      prefetchCache[blockDim.x * i + threadIdx.x] =
        tex1Dfetch(textureRef, beginIdx + idx + i + 1);
    }
  } else {
    value = prefetchCache[blockDim.x * (prefetchIdx - 1) + threadIdx.x];
  }
  ...
  // do something that overwrites the texture cache
}
```

Listing 5. Utilizing the texture cache to prefetch data to shared memory.

If there is no way to coalesce global memory transactions, texture memory can provide a potentially faster alternative. Texture memory performs spatial caching in one, two or three dimensions depending on the type of texture. This will cache the values surrounding an accessed value which allows them to be accessed very fast. Providing a speed increase if threads in the same block access values in texture memory that are spatially close to each other. If the value accessed is not in the cache, the transaction will be as slow as global memory access.

Listing 5 shows another prefetching approach than the one proposed in Listing 2 to speed-up uncoalesced reads of spatially nearby global memory addresses by the same thread. It uses the texture cache and shared memory. This only works if the threads in the same warp or even better in the same block access data that is close enough together to fit into the texture cache. It also assumes that the texture cache is overwritten by new values later on, otherwise the prefetching to shared memory can be omitted and the texture cache can be used directly.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

2.2.5. *Constant memory*

Like the texture cache, every SM has its own constant memory cache on-chip. Device memory reads are only necessary on cache misses, otherwise a constant memory read only costs a read from the constant cache. (See PG Chapter 5.1.2.3.)

Constant memory access is useful when many threads on the same SM read from the same memory address in constant space (i.e. the value is broadcasted to all the threads). If all threads of a half-warp read from the same cached constant memory address, then it is even as fast as reading from a register. (See PG Chapter 5.1.2.3.)

3. FIELD EQUATION MODELS

Field equations can be solved numerically using discrete meshes and finite differences in a well-known manner. Like image processing algorithms they are regular, rectilinear, stencil oriented and can make use of regular arrays. These simulations often require large field lengths and/or a large number of time steps to produce scientifically meaningful results.

The regular memory access of field equations and the simple way in which they can be parallelized make them good candidates for testing GPU capabilities. The purpose of testing these field equations is to compare the performance and strengths/weaknesses of the types of memory and methods of parallel decomposition made available by CUDA. The field equation selected to test these capabilities is the well-known CH equation (see Figure 1).

3.1. Cahn–Hilliard equation simulation

The CH equation is a well-known phase separation model that simulates the phase separation of binary alloy containing A- and B-atoms [26]. It approximates the ratio of A- and B-atoms within a discrete cell as a single concentration value in the range $[-1,1]$. The change in concentration value

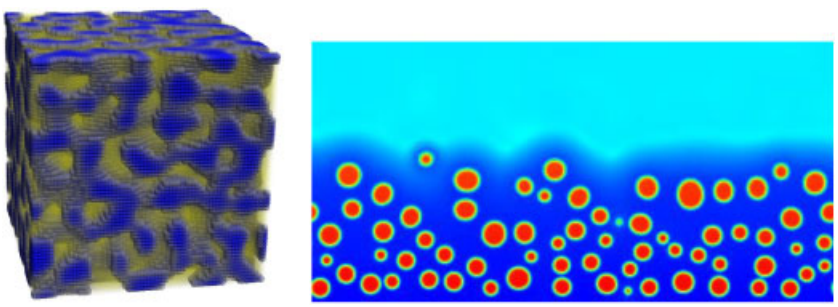


Figure 1. Simulated Cahn–Hilliard systems in: 3-d with a 50/50 symmetric concentration and periodic boundary conditions showing spinodal decomposition (left), and in 2-d with a 25/75 minority concentration showing droplet condensation, with fixed boundary conditions (right).

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

for each cell at a point in time can be calculated according to the formula:

$$\frac{\partial \phi}{\partial t} = m\nabla^2(-b\phi + u\phi^3 - K\nabla^2\phi) \tag{2}$$

This can be simulated numerically in a number of ways, either in Fourier space [27] or in real space using finite differences [28]. The simulation presented within this article is implemented with CUDA and uses the finite differences real space method.

To numerically simulate the CH equation, the rate of change in concentration must be calculated for each cell according to Equation (2). This value can then be used to integrate the total change in each cell's value over a discrete time step. This integration can be performed by any numerical integration method but the Runge–Kutta 2nd-order method or similar is required for accurate results. This change in concentration of each cell is calculated over many time steps.

Decomposing the CH simulation into a GPU program is not a trivial task. Although initial implementation is simple, proper optimization is a complex task. There are several types of memory that the simulation can use, each of which must be tested to find the optimal method. As there is no absolute way to calculate the performance of each of these methods so this optimization stage is often a trial-and-error process. Presented here are several different approaches and designs for the CH GPU program along with their individual merits and speed-up factors.

3.1.1. The function of a kernel

The first step in decomposing a problem into a GPU program is to determine the function of a single thread. It is desirable to split the problem into as many threads as possible so that the GPU schedulers can make the optimal use of the GPU multiprocessors. In the discrete CH simulation, the most logical and smallest unit of work is calculating the change in concentration of a single cell.

This approach does place a limit on the maximum size of the field. There is a maximum of 512 threads per block and a maximum of 65 535 blocks per grid. This provides a maximum field length of: 33 553 920 in 1D, 5792×5792 in 2D and $322 \times 322 \times 322$ in 3D. This maximum size is specific to the current generation of GPUs and will increase with future generations. In the mean time, if a larger system is required (feasible for a 3D or higher-dimension problem) the structure must either be changed such that each thread updates more than one cell or that a multi-stage kernel that makes several calls to update different sections of the field.

The advantage of writing a GPU program with many threads is that it will still be useful for future generations of GPUs. The CUDA Programming Guide [25] states that any program with 100 blocks per grid will scale well to future devices and programs with 1000 blocks and will still make effective use of GPU hardware for the next several generations. A typical CH simulation of the structure described here has 4096+ blocks and should scale well for many new generations of GPU. Once the performance of GPUs overtakes this computational requirement, it will simply become possible to increase the size of the system to make better use of the computational power available.

3.1.2. Simulation flow

For this CH simulation, each thread is responsible for updating the concentration value of one cell depending on the values of the cells surrounding it. The values of the cells in the field must be read

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

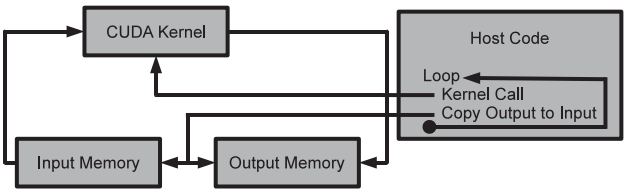


Figure 2. The data flow in the CH program. At each time step, the host program calls the kernel and then copies memory between the input and output memory. The kernel itself reads the field from the input memory, computes the change and writes it to the output memory.

out of one array and written into another to ensure that every calculation uses the correct values. When this function is performed by many threads on a GPU, it is vital that all threads complete the same time step before any thread starts the next or out of order reads/writes may occur.

However, CUDA does not provide a method of synchronizing threads across separate blocks. The only way to ensure that every thread has completed a time step is for the kernel function to complete and return to the host program. Thus, the following kernel and program structure must be used. Each kernel call will read the necessary data from memory, compute a single time step, write the result into a different array in memory and return. The loop that controls the number of time steps is within the host code that makes multiple kernel calls. This will guarantee that all threads in all blocks will be synchronized for all simulation time steps. Figure 2 shows this flow of control in the CH GPU program.

3.1.3. Implementing Runge–Kutta 2nd-order integration on the GPU

So far the function of the kernel discussed has been to calculate the change in concentration for each cell in a CH field, compute the new field value and save it to an output array. However, computing the new concentration value is not as simple as adding the change onto the current value. While that simple method can be employed, it is simply an implementation of the Euler integration method, the Runge–Kutta 2nd-order method (RK2) is a far better approach.

Incorporating the RK2 method into the CUDA implementation of the CHC simulation is a relatively simple task. To implement the RK2 method, the kernel must perform two different functions depending on which step it is performing. For the first step, the kernel simply has to perform the same process as the Euler method just with a time step of half the size. This will save the next state of each cell into a separate output array. The second step calculates the change in concentration based on the values calculated by the first step and then adds this change onto the values from the first input array.

This requires that the two input arrays be passed to the kernel. The first part of kernel simply reads the first input array and writes to the output array. This output array then becomes the second input array to the second part of the kernel. The kernel then calculates the change in concentration from the second input array and adds it onto the cell values from the first input array. These values are then written to the final output array. This process is shown in Figure 3.

This method of simulation of the CH equation is slower than the Euler method as there must be two kernel calls per time step. The second of these kernel calls requires an extra global memory access.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

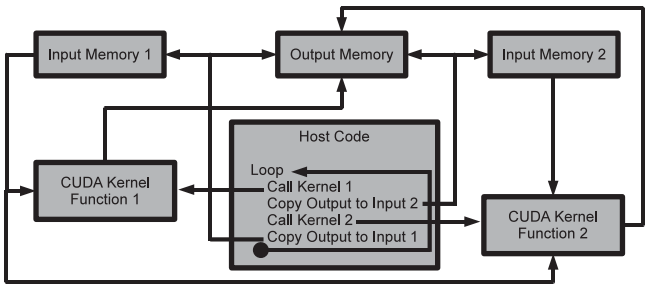


Figure 3. The flow of data in the CH architecture using the RK2 integration method. At each time step the host calls Kernel 1, copies the output into the second input memory, calls Kernel 2 and then copies the output back into the input memory. Kernel 1 computes the initial Euler half time step and Kernel 2 completes the RK2 step using input memory one and two.

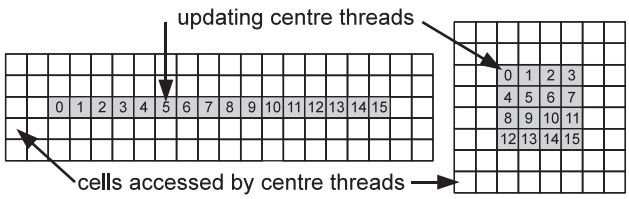


Figure 4. An illustration of two methods of allocating threads to cells within a Cahn–Hilliard field.

However, using this or another higher-order integration method is vital for any CH simulation. The Euler method is simply too unstable and inaccurate for any results it produces to be meaningful. This article discusses four kernel designs, all of which implement this RK2 method.

3.1.4. Thread allocation

Here we discuss two structures for organizing the threads within a block—strips and rectangles. When threads are organized into strips, they are allocated to sequential cells in the field. This is the only method of allocating cells in a 1D system but a rectangular structure can be used for two or more dimensions (this structure is discussed for two dimensions but can be extended into a cube for three dimensions, etc.). Figure 4 shows these two methods of arranging threads in two dimensions.

These two methods of allocating threads to cells are both valid and useful. However, to understand their individual merits a proper understanding of memory access is required.

3.1.5. Memory access

Each thread within the simulation is responsible for calculating the change in concentration for a single cell according to the discrete CH equation (2). This formula uses the Laplace and Laplace^2

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

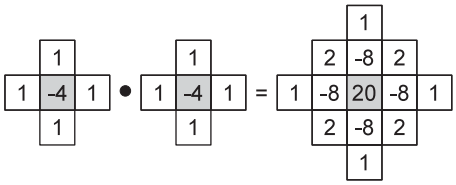


Figure 5. A stencil showing the surrounding cells a thread must access to calculate the change in concentration of the centre cell.

operators: ∇^2 and ∇^4. This requires each thread to access the concentration values of the cells surrounding the cell it is responsible for. The following memory access pattern is discussed for a two-dimensional system but the ideas can be extended to 3 or more dimensions. The values of the surrounding cells that each thread must access is shown by the stencil in Figure 5.

3.1.6. Kernel A—Global memory

Kernel A uses global memory for all data access. Each thread will read all the values required out of global memory, perform the calculation and write the result back into another array in global memory. If the threads are organized in either a strip or a rectangle with a width of a multiple of sixteen, any global memory transactions aligned to the memory block will be coalesced. This will result in five coalesced memory accesses per half-warp (one for each cell value at the horizontal centre of the stencil seen in Figure 5). The other eight memory transactions will not be aligned to the memory segment and thus cannot be coalesced into a single memory transaction. Devices of compute capability of 1.0 and 1.1 must perform 16 separate memory transactions to fetch these values and devices of compute capability 1.2 and higher must perform two coalesced transactions (one for each memory segment). If a strip is used and the length is not a multiple of sixteen, the last warp will not have coalesced memory access. If a rectangle is used and the width is not a multiple of sixteen, the global memory transactions will not be coalesced. Provided that the strip length or rectangle width is a multiple of sixteen, both methods will provide a similar performance. A code snippet from Kernel A can be seen in Listing 6.

This method is faster than the CPU version, however, there is a lot of inefficiency with this design. Many threads within the same block have to read the same value from global memory. It would be more efficient for these threads to share the values through shared memory.

3.1.7. Kernel B—Shared memory

Kernel B uses shared memory to cache values read from global memory. To make use of shared memory, each thread can access one value from the global input memory and load it into shared memory. The surrounding threads in the same block can then access this shared data much faster than global memory. However, this raises the question of the optimum arrangement for threads to minimize the global data access.

If the threads are allocated to sequential cells in a strip then every thread must retrieve the values in the two cells above it and the two cells below it from global memory (see Figure 5). They will

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
//Read Input
float crcm1     = global_input[y   * N + xm1];
float crm1c     = global_input[ym1 * N + x];
float crc       = global_input[y   * N + x];
float crp1c     = global_input[yp1 * N + x];
float crcp1     = global_input[y   * N + xp1];
...

//Perform calculation
...

//Write output
global_output[y * N + x] = result;
```

Listing 6. Code showing the input and output of Kernel A (the reading of only five values are shown here, the others are accessed in the same manner).

only be able to use shared memory to access values from the cells to the left and right. Threads that are arranged into a rectangle of cells can make more efficient use of shared memory. The threads within the centre of the grid will be able to access all neighbouring cell values from shared memory.

The problem with this is that the threads on the edge of the rectangle (or end of the strip) must perform more global memory transactions than the other threads. This causes a branch in the kernel which can seriously impact the performance of the program. If any two threads within a warp take different branches, all the threads must perform the instructions for both branches but the appropriate threads are disabled for one of the branches. This causes a significant performance loss.

To eliminate this problem, the threads are rearranged such that there is a border of two surplus threads around the outside of each thread grid that simply access the cell values from global memory and save them to shared memory. The downside of this approach is that the thread grids must overlap resulting in more threads than there are cells. However, this approach is still faster than kernels with branches within them. The allocation of these threads to cells can be seen in Figure 4 where a thread is assigned to each empty surrounding cell. Listing 7 shows how each kernel accesses the data.

In order for this approach to work, the thread rectangle must be applied over the entire CH lattice such that each cell has one updating thread assigned to it. The number of updating threads in the thread grid must divide the size of the lattice perfectly. A CH model lattice is normally a square grid with a size of some power of two. However, this is often for convention and in this case the size of the grid can have a serious impact on the performance of the program. As memory access is fastest if all threads in a half-warp (16 threads) access sequential addresses in memory (as the transactions can be coalesced), it is advantageous if each set of 16 threads in the grid are allocated to a single row in memory. It is also desirable for the number of threads within a grid to be a multiple of 32 as each warp will always be full and make the maximum use of the GPUs multiprocessors.

For this reason the optimal thread grid structure is a 16×16 block. This 16×16 block has a 12×12 block of updating threads within it where global memory transactions can be coalesced if the values are aligned to a segment of memory. For devices of compute capability 1.0 and 1.1 the global memory transaction of every fourth block will be coalesced (as every fourth 12×12 block will align with memory). Devices of compute capability 1.2 and higher can coalesce all memory transactions and will simply require two transactions for blocks that are not aligned to the memory

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
__shared__ float shared_data[BLOCK_SIZE * BLOCK_SIZE];

//Read into shared memory
float crc = global_input[y * N + x];
shared_data[ty * BLOCK_SIZE + tx] = crc;
__syncthreads();

//Work out if this thread is a calculating thread or a border thread
bool writer = (tx > 1) && (tx < BLOCK_SIZE - 2) && (ty > 1) && (ty < BLOCK_SIZE - 2);

if(writer) {
        //Read from shared memory
        float crcm1     = shared_data[ty * BLOCK_SIZE + txm1];
        float crm1c     = shared_data[tym1 * BLOCK_SIZE + tx];
        float crp1c     = shared_data[typ1 * BLOCK_SIZE + tx];
        float crcp1     = shared_data[ty * BLOCK_SIZE + txp1];
        ...

        //Perform calculation
        ...

        //Write output
        global_output[y * N + x] = result;
}
```

Listing 7. Code showing the input and output of Kernel B (the reading of only five values is shown here, the others are accessed in the same manner).

segments. It can be assumed that any useful CH simulation that requires a GPU to execute in a reasonable amount of time will be larger than 12×12, thus this block structure will fit perfectly into any lattice where the length is divisible by 12.

3.1.8. Kernel C—Texture memory

Kernel C uses another option for accessing data, texture memory. Texture memory can provide several performance benefits over global memory as input for the kernel [25]. When values are retrieved from texture memory, a global memory access is only required if the value is not already stored within the texture cache. This effectively performs the caching effect of using shared memory but is performed by the GPU hardware rather than within the code itself. This also leaves the shared memory free for other purposes. The caching ability of textures can provide significant performance gains and can remove the often troublesome problem of arranging access to global memory to ensure coalesced access.

If the values required by the thread do not fit into the texture cache it will have to be constantly re-read from global memory which could seriously reduce the performance of the program. The actual size of the texture cache depends on the graphics card, however it should be at least 6 KB per multiprocessor (see PG Appendix A.1.1). Easily capable of storing the memory required for a 16×16 block of threads (maximum power of two square under the 512 limit). Listing 8 shows the code from Kernel C that reads the data from a texture.

There are some other advantages provided by texture memory such as the ability to automatically handle the boundary conditions of a field. Cells on the edge of a field must either wrap around to the other side of the field or mirror back into the field. Texture memory in a normalized access mode

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
texture<float, 2, cudaReadModeElementType> texture_input;

//Read from texture memory
float crcm1     = tex2D(texture_input, xm1, y);
float crm1c     = tex2D(texture_input, x, ym1);
float crc       = tex2D(texture_input, x, y);
float crp1c     = tex2D(texture_input, x, yp1);
float crcp1     = tex2D(texture_input, xp1, y);
...

//Perform calculation
...

//Write output
global_output[y * N + x] = result;
```

Listing 8. Code showing the input and output of Kernel C (the reading of only five values are shown here, the others are accessed in the same manner).

can automatically perform these boundary condition functions which can help to avoid expensive conditional statements that can cause threads within the same warp to take different branches and cause significant impact on the performance.

Another limitation of texture memory is that it can only support up to three dimensions. Any CH system that wishes to operate on four or more dimensions can no longer easily make use of functions of texture memory. For texture memory to be used for any system that operates in more than three dimensions must create an array of three-dimensional texture memory segments. These memory segments would only be cached in three dimensions and would provide no caching ability in the fourth or higher dimension.

3.1.9. Kernel D—Texture and shared memory

For the sake of completeness, a fourth kernel has been implemented. This kernel uses texture and shared memory to read the values from memory. The data are read out of texture memory into shared memory and then each thread accesses the values from shared memory. This is not particularly efficient because the data will be cached by the texture memory and then again by the shared memory. However, it was implemented so that every method was properly explored. The code for this kernel is shown in Listing 9.

One interesting problem encountered when implementing this kernel was the size of the texture. Because the size of the field must be divisible by the size of the inner rectangle (12×12, see Section 3.1.7) the texture size must be a multiple of 12. However, when normalized coordinates are used to provide automatic boundary condition handling (see Section 3.1.8) the texture memory access no longer worked. It was found that normalized coordinates only worked correctly when the texture size was a power of 2 (tested on an NVidia GeForce 8800 GTS). No documentation was found that warned of this requirement and the kernel had to be re-written to use non-normalized coordinates and hand-coded boundary condition handling.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
texture<float, 2, cudaReadModeElementType> texture_input;
__shared__ float shared_data[BLOCK_SIZE * BLOCK_SIZE];

//Read into shared memory
float crc = tex2D(texture_input, x, y);
shared_data[ty * BLOCK_SIZE + tx] = crc;
__syncthreads();

//Work out if this thread is a calculating thread or a border thread
bool writer = (tx > 1) && (tx < BLOCK_SIZE - 2) && (ty > 1) && (ty < BLOCK_SIZE - 2);

if(writer) {
        //Read from shared memory
        float crcm1     = shared_data[ty * BLOCK_SIZE + txm1];
        float crm1c     = shared_data[tym1 * BLOCK_SIZE + tx];
        float crp1c     = shared_data[typ1 * BLOCK_SIZE + tx];
        float crcp1     = shared_data[ty * BLOCK_SIZE + txp1];
        ...

        //Perform calculation
        ...

        //Write output
        global_output[y * N + x] = result;
}
```

Listing 9. Code showing the input and output of Kernel D (the reading of only five values are shown here, the others are accessed in the same manner).

3.1.10. Output from the kernel

The only way for the threads to output the result of the calculation is by writing to an array in global memory. As the output must be performed via global memory, it is important that it can be organized into coalesced write transactions. As discussed, a shared memory approach uses a thread grid with a 16×16 block of threads with a 12×12 block of updating threads in the centre and a texture or global memory approach uses a 16×16 block of threads. For the shared memory approach, although there is only a 12×12 block of updating cells in the centre of the grid, all memory writes aligned to a memory block will still be coalesced. Coalescing takes place even if some of the threads in a half-warp are not participating in the write. Thus, when each row of 16 threads attempts to write output to memory, a maximum of 12 of those threads will actually be writing data. The writing transaction of those 12 threads will be coalesced into a single memory transaction provided that they are aligned with a memory block. As with read transactions, compute capability devices 1.2 and higher will combine non-aligned writes into two coalesced writes.

3.1.11. Performance results

The only way to truly compare the different implementations of the CH simulation in CUDA is to benchmark them in a performance test. Each of the discussed simulation implementations was set to calculate the CH simulation for various field lengths and the computation time (in seconds) per simulation time step was measured. The CPU program shown is a single-thread program written

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

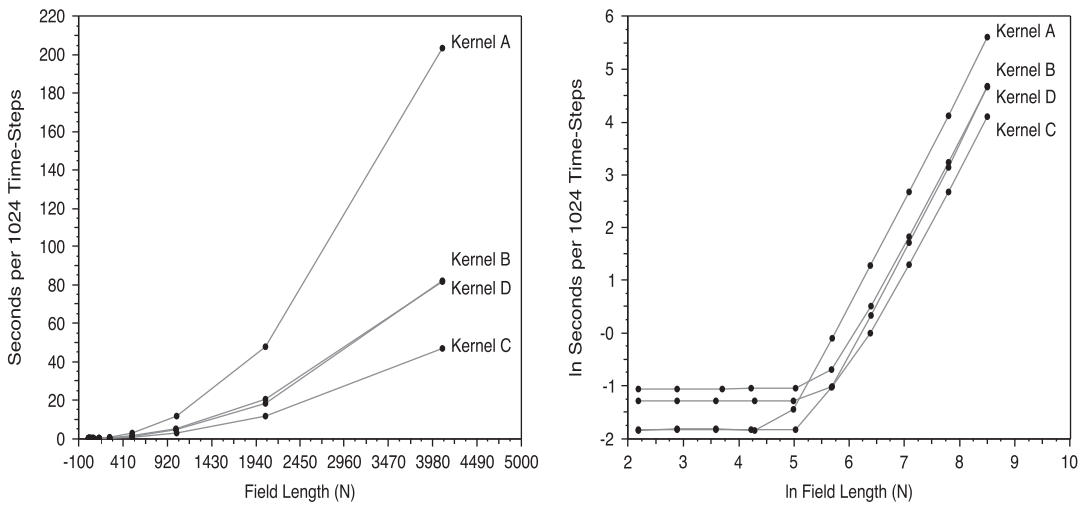


Figure 6. A comparison of the CH GPU simulation implementing RK2 with various types of memory—Texture, Global, Texture and Shared, and Global and Shared. Figure 7 shows the encouraging performance gain that these methods provide over the CPU implementation.

in C that was executed on a 2.66 GHz Intel® Xeon processor with 2 GBytes of memory. All GPU programs were executed on an NVIDIA® GeForce® 8800 GTS operating with an Intel Core™2 6600. These results can be seen in Figure 6.

It can be seen from Figure 7 that the GPU implementation with the highest performance is the version that uses texture memory (Kernel C). Kernel C provides a 50× speed-up for field lengths 1024 and above. As the texture memory cache stores all the values needed by the threads in a block, the only access to global memory required is the initial access that loads the values into the texture cache. From this point on, all memory accesses simply read from this fast texture cache.

The global memory method requires many more costly global transactions. Although some of these transactions are coalesced, it is significantly slower than accessing the texture cache (13× speed-up as compared with 50×). The two methods that use shared memory are effectively mimicking the caching ability of texture memory. However, this caching is performed within the program itself and is therefore slower than the automatic texture cache (Kernels B and D both provide speed-ups of 30×).

To appreciate the advantages of the GPU implementations, the comparison between the GPU implementations and a CPU version must be shown. As is expected, the GPU implementations of the simulation execute significantly faster for large field lengths. As is common with parallel implementations, very small problems can be computed by a serial CPU faster as there is no parallel overhead. However, as the field length (N) increases the problem becomes far more efficient to calculate in parallel using the GPU. As it is large simulations that provide the most meaningful results, the GPU implementations of simulations have been proven to be extremely valuable.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

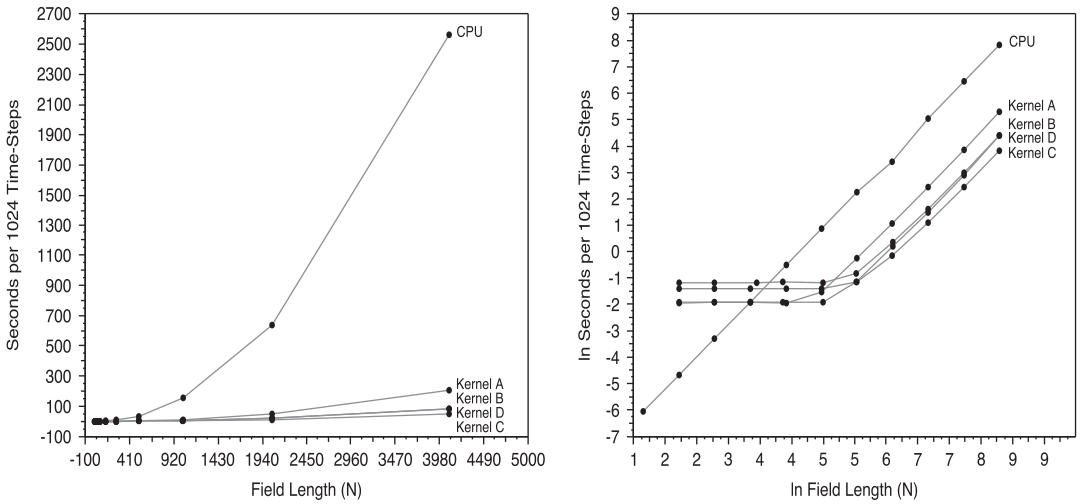


Figure 7. The performance gains of the GPU kernels over the CPU implementation. This should be read in conjunction with Figure 6 which compares the GPU kernels against each other in more detail. Note also that the N limiting logarithmic behaviour of the CPU and GPU kernel implementations shows the same slope (2) as would be expected for an $O(N^2)$ problem.

4. GRAPH AND NETWORK METRICS

Complex networks and their properties are being studied in many scientific fields to gain insights into topics as diverse as computer networks like the World Wide Web [29], metabolic networks from biology [30–32], and social networks [33–35] investigating scientific collaborations [36,37] or the structure of criminal organizations [38] to name but a few. These networks often consist of many thousands or even millions of nodes and many times more connections between the nodes. As diverse as the networks described in these articles may appear to be, they all share some common properties that can be used to categorize them. They all show characteristics of small-world networks, scale-free networks, or both. These two types of networks have therefore received much attention in the literature recently.

Networks are said to show the small-world effect if the mean geodesic distance (i.e. the shortest path between any two nodes) scales logarithmically or slower with the network size for fixed degree k [39]. They are highly clustered, like a regular graph, yet with small characteristic path length, like a random graph [40]. Scale-free networks have a power-law degree distribution for which the probability of a node having k links follows $P(k) \sim k^{-\gamma}$ [41]. For example, in Price's paper [42] concerning the number of references in scientific papers—one of the earliest published examples of scale-free networks—the exponent γ lies between 2.5 and 3.0 for the number of times a paper is cited.

Determining some of the computationally more expensive metrics, like the mean shortest paths between nodes [43,44], the various clustering coefficients [37,40,45,46], or the number of cycles/circuits of a specific length, which are commonly used to study the properties of networks,

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

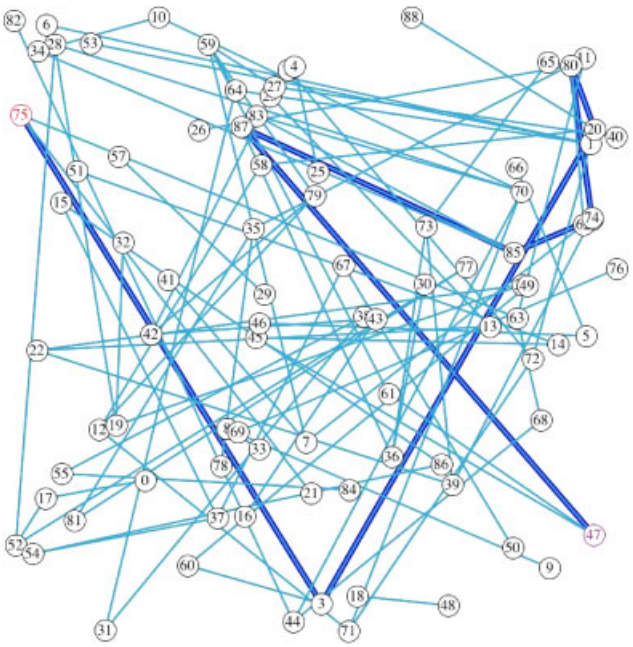


Figure 8. A random graph embedded in 2-space, with the shortest path shown between nodes 47 and 75.

can be infeasible on general purpose CPUs for such huge data structures. We utilize the raw processing power of today's commodity GPUs instead of expensive super computers or clusters to gain significant speed-ups (1–3 orders of magnitude) when executing these algorithms.

Networks are often modelled as graphs, a data structure that consists of a set of vertices (the nodes of the network) and a set of either edges (undirected connections between vertices) or arcs (directed connections between vertices). The terms are used interchangeably in this article.

Figure 8 shows a sample graph embedded in 2-space, where the shortest path has been computed between two nodes. The All-Pairs Shortest Path (APSP) metric—which determines the mean shortest path between any two vertices in a graph—is a characteristic of the whole graph, which is approximately 1.36 hops for the sample shown.

Sometimes we are interested in studying a specific imported graph, but more often we want to statistically study lots of network instances generated with the same algorithm. The consequence is that we have to calculate the metrics not only for one graph, but for dozens or even hundreds. This makes the need for fast implementations of the algorithms even more obvious. But it also means that we can independently, without any communication overhead, execute the same algorithm for each of the network instances on a cluster of CPUs or GPUs. A coarse level of parallelization can thus be achieved easily. The results are finally aggregated and analysed.

To utilize the processing power offered by GPUs, the algorithms themselves have to be adapted to support the SIMD (or SIMT in CUDA terminology) model. A basic implementation that runs on the GPU can be implemented quite easily for some of the graph algorithms that perform the

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

same operation on every vertex or edge of the graph. However, optimizing the implementation to make the most out of the available processing power can be quite challenging. It is necessary to think about the optimal memory layout to reduce global memory transfers and make use of the available bandwidth through coalescing or texture fetches. If the threads have to communicate with each other at certain time steps, then it is a good idea to arrange them in a way that allows them to use shared memory for most of the communication and only use global memory when absolutely necessary. Furthermore, divergent branches within warps caused by conditional execution or loops of different lengths should be avoided where possible, as every branch taken by at least one thread within the same warp has to be serialized and executed by every thread of that warp.

The key problems that we had when implementing graph algorithms are related to the layout and access of global memory and to branch divergence. The graph representation in memory and the patterns used to access the vertices and their respective edge lists have a large impact on the final performance as shown in the following sections. Graph algorithms often iterate over the adjacency-lists of vertices to visit the nodes reachable from each vertex. As there is no common pattern of how vertices are connected with each other that is shared among different types of networks, it is often impossible to completely avoid random memory accesses. Section 4.2 proposes different memory layouts that can be used to represent graphs on a CUDA-enabled device and explains why some provide better performance than others. Section 4.3 describes the implementation of a specific graph algorithm and how it was optimized. Each optimization step also states the respective performance gains to show which changes have the largest impact. Finally, Section 4.4 compares the CUDA implementation with our implementation for the CPU.

4.1. Method and approach

We implemented the APSP algorithm based on the approach described by Harish and Narayanan [47]. The algorithm is basically a breadth-first search starting from a source vertex s with a frontier that expands outwards by one step in each iteration (kernel call). This is done until all vertices that can be reached from s have been visited. The algorithm is invoked once for every $s \in V$, where V is the set of vertices of a graph. We then optimized the implementation, tested various data structures, different ways to access global memory and utilize shared memory, reduced data transfer between the host and device, and more. The following sections describe several of the optimization techniques that were applied to the algorithm.

We use two different graph generation algorithms to generate the network instances that are used to test the GPU and CPU implementations. For every tested network size N, 10 instances of each of the two network types are generated with the same parameters (size, degree, etc.). The execution times for the implementations are measured and the mean values of the timing results are used to generate the graphs given in the following sections. For the CPU implementation, the standard deviations from the mean values are calculated and displayed as error bars. The execution time of the GPU implementation, however, depends largely on the result value, as the breadth-first search from a source vertex s requires as many kernel calls as the longest geodesic distance from s to any other vertex. Thus, networks with a high APSP value take much longer to process than networks with a small APSP value. An almost linear relationship exists for networks that are otherwise equal. To accommodate for this fact, the timing results for the CUDA implementations are normalized before the standard deviations used for the error bars are calculated. The formula used to normalize

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

the values is:

$$\text{Normalized time} = \frac{(\text{time in seconds})}{\text{APSP}} * (\text{mean APSP}) \tag{3}$$

The first algorithm generates networks based on Watts' α-model [48]. This is an algorithm that, depending on the provided parameters, generates a strongly clustered small-world network that may or may not be fully connected. Two different parameter sets are used to generate the networks, both produce graphs that are always fully connected and strongly clustered. The networks created with the first parameter set (1) have a rather high mean shortest path length. A minimally connected ring substrate, in which every vertex has precisely two edges, is used to ensure that every vertex can be reached from every other vertex. The second set of parameters (2) produces graphs with slightly lower clustering coefficients due to a higher randomness of the connections, but with much lower mean shortest path lengths. As it does not use a substrate, the fully connectedness of the resulting graphs is not guaranteed by the algorithm, but rather through graph analysis after they were generated. The degree of the resulting networks is $k = 10$ in both cases.

The second algorithm generates networks (3) based on the gradual growth and preferential attachment model published by Barabási and Albert [41]. The graphs follow a power-law degree distribution, are always fully connected and have a small mean shortest path length. The degree of the resulting networks is $k = 100$.

4.2. Data structures

A graph $G = (V, E)$ with a set of V vertices and a set of E edges or arcs can be represented in a variety of ways. Figure 9 illustrates a number of representations for directed graphs. While they can also be used to represent undirected graphs, they are not very efficient, as an edge has to be represented by two arcs going in either direction. Thus, the information is stored twice, increasing the required storage space and the risk of inconsistent data when one of the arcs gets lost. However, our algorithms have to be able to work with directed graphs. Therefore, they use data structures that are based on arcs and not on edges.

Example (a) graphically shows the vertices and how they are connected with each other. This representation is often used to visualize a network for humans, whereas the other representations are more likely to be used as data structures read and modified by programs. The adjacency-matrix representation (b) with its $O(N^2)$ memory usage wastes a lot of space if the graph is sparse, but it allows to quickly determine if a specific arc exists. This is the reason why this representation is commonly used in graph algorithms like the APSP algorithm. However, we avoid the identity-matrix representation for the GPU implementations due to the tighter memory limitations on graphics cards, thus making it possible to process larger networks. The data structure illustrated in (c) is much more memory efficient for sparse graphs, since only existing arcs are stored. These three graph representations can commonly be found in the literature [49].

The data structure shown in (d) is similar to the one in (c), but—unless the neighbours-lists only contain a single element—it wastes even less space as it gets rid of the pointers required to point from one element of the linked lists to the next one. The data structure shown in (e) is the most memory efficient data structure presented here. It merely requires $O(E + V + 1)$ space. The adjacency-list length of a vertex v is not stored explicitly, but it can easily be calculated with the

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

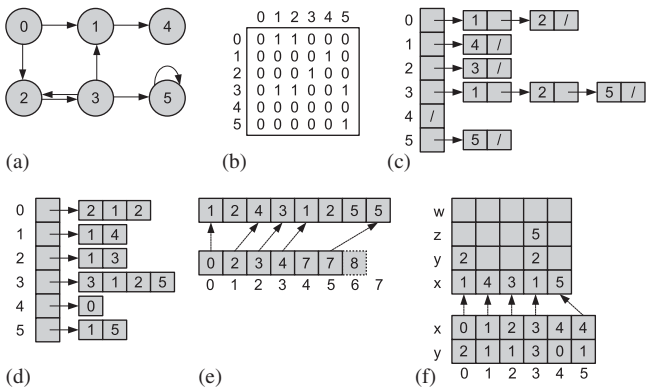


Figure 9. Six representations of a directed graph G with 6 vertices and 8 arcs. (a) A visual representation of G. (b) The adjacency-matrix representation of G. (c) An adjacency-list representation of G based on an array of linked-lists. (d) An adjacency list representation of G based on an array of arrays. The first element of every neighbours-list is used to record its length. (e) A representation of G based on two separate one-dimensional arrays. The array on the top contains the adjacency-lists of all vertices. The arcs of the first vertex are followed by the arcs of the second vertex and so on. The second array represents the vertices and points to the first element of the respective neighbours-list. (f), Like (e), uses an array for the arcs and an array for the vertices. However, the arcs are grouped into structures of up to 4 arcs each to allow for a better utilization of the memory bandwidth. The vertices array consists of tuples of 2 elements, the first pointing to the index of the first structure of arcs that form the adjacency-list of this vertex, the second recording the total number of arcs in that list.

index values stored in the elements v and $v+1$ of the vertices array ($vertices[v+1]-vertices[v]$). The vertices array is of length $V+1=N+1$ to make this work with the neighbours-list of the last vertex as well. The data structure shown in (f) improves the utilization of the available memory bandwidth by using structs that can hold 4 integers as described in Section 2.2.1 and Listing 2, only that the length of the sub-lists is stored in the vertices array. Some space is wasted if the lengths of the adjacency-lists is not a multiple of 4.

The data structures (e) and even more so (f) are less commonly used by algorithms running on the CPU, but they turn out to be well suited for CUDA applications. They efficiently represent sparse graphs and in the case of (f) are specifically optimized for the way memory is accessed on the graphics card. These two data structures or slight variations of them are used in the implementations described in the following section. It should be noted, however, that they are not necessarily the best choice when the kernels modify the graph structure.

Efficiently adding additional arcs or vertices to the graph from within a kernel is a challenge, as memory can only be allocated by the host and not by the kernels in CUDA. If the maximum degree is known and enough device memory is available to store a data structure of size $N*k_{\max}$, then allocating a one-dimensional array like in (e) may be a good solution. The second array used to point to the first index of each neighbours-list is not needed in this case, as the fixed degree makes it easy to calculate it. If the maximum degree is not known, then a flexible data structure like the one described in (c) with the maximum storage space available for the graph allocated in advance may be a good candidate. The best solution always depends on the particular problem and testing more than one data structure can sometimes yield unexpected results.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

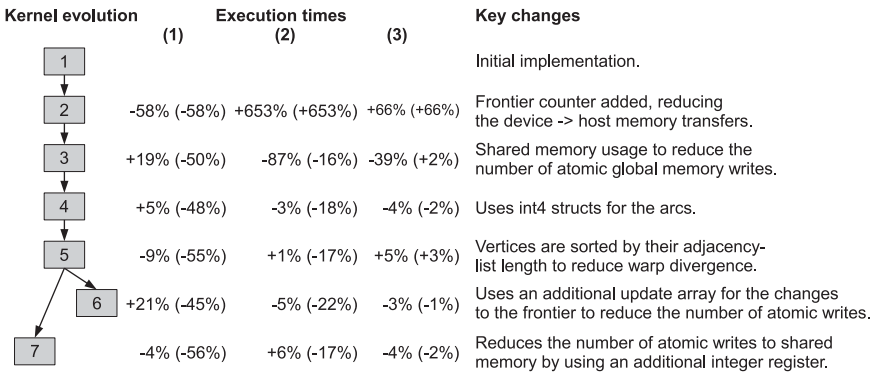


Figure 10. The evolution of the kernels, the key changes between the implementations and how the changes affect the execution time when executed on graphs generated with Watts' α-model (1) and (2) and Barabási's scale-free network model (3). The first percentage value is the difference to the previous kernel in the evolution hierarchy, the second value (in parentheses) is the difference to the initial implementation (kernel 1). The percentages are based on the execution time measurement results for the networks of size $N = 30000$.

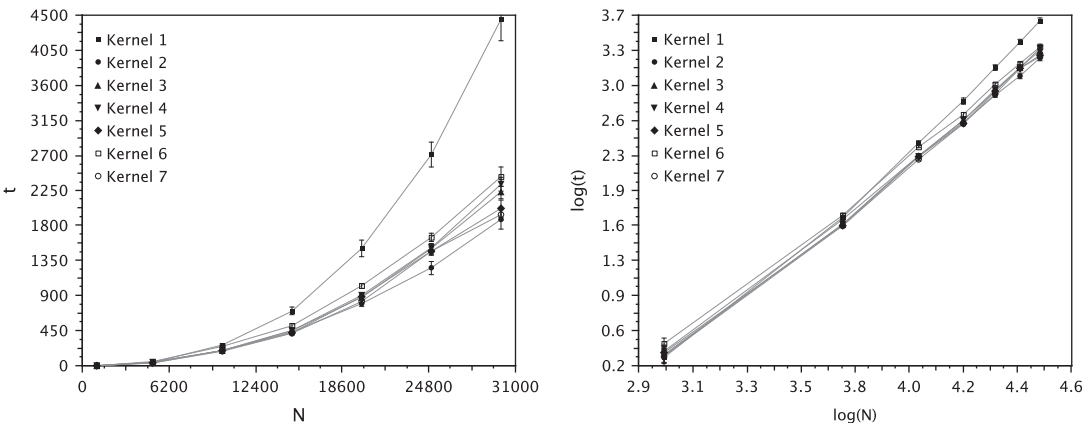


Figure 11. The execution times on the GPU for graphs generated with approach (1) on the left and the same data on a log–log scale on the right. The slopes on the log–log plot are in the range of $\approx (2.04, \ldots, 2.45)$. The uncertainties are mostly comparable to the symbol sizes.

4.3. Optimizing the APSP algorithm

This section describes the optimization steps applied to the APSP algorithm and the effects that they have on the performance. Figure 10 gives an overview of the differences between the 7 implementations that were tested. The timing results illustrated in Figures 11–13 are for the graphs generated with Watts' α-model (1) and (2) and Barabási's scale-free network model (3), respectively. The tested networks are of size $N = \{1000, \ldots, 30000\}$. The execution times t are given in seconds. Error bars are displayed in all plots, but in some cases they are too small to be visible.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

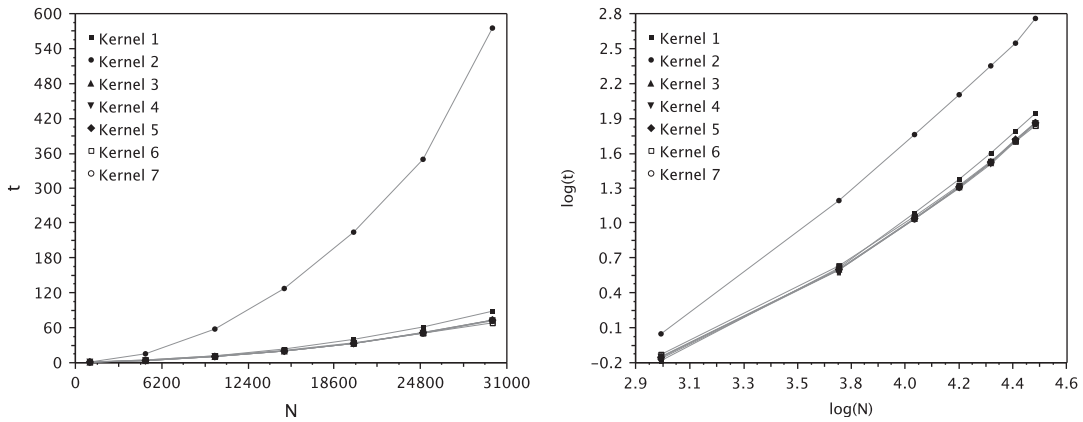


Figure 12. The execution times on the GPU for graphs generated with approach (2) on the left and the same data on a log–log scale on the right. The slopes on the log–log plot are in the range of $\approx(1.49, \ldots, 1.94)$. The uncertainties are comparable to the symbol sizes.

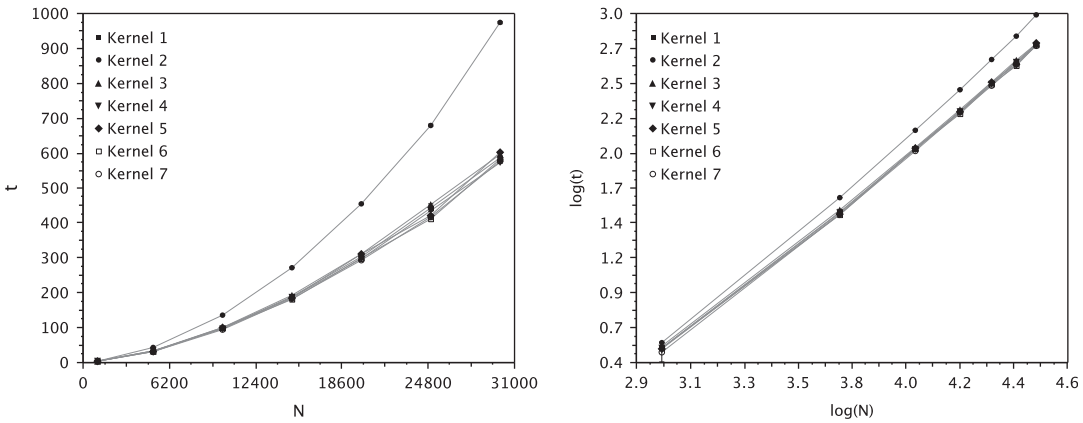


Figure 13. The execution times on the GPU for graphs generated with approach (3) on the left and the same data on a log–log scale on the right. The slopes on the log–log plot are in the range of $\approx(1.51, \ldots, 1.71)$. The uncertainties are mostly comparable to the symbol sizes.

The roughly straight lines on the log–log plots show that the execution times for a specific network-type scale with the network size according to the usual power laws, with the fitted slopes giving the exponents. The slopes for the different kernels vary between $\approx(2.04, \ldots, 2.45)$ for (1), $\approx(1.49, \ldots, 1.94)$ for (2) and $\approx(1.51, \ldots, 1.71)$ for (3).

Kernel 1 is the initial implementation. It uses the data structure described in Figure 9(e). A frontier array of length N indicates which vertices are to be processed in each iteration. A vertex is flagged to be processed in the next iteration if it is in the adjacency-list of one of the currently active vertices and this path has a lower cost than any of the previously discovered paths going

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
// if this vertex (tid = thread ID = vertex ID) is active in the frontier
if (frontier[tid]) {
  frontier[tid] = false; // remove the vertex from the frontier

  int arcIdx = vertices[tid];
  int arcsEnd = vertices[tid + 1]; // begin of the next adjacency list
  int newCostForNbrs = cost[tid] + 1; // every arc has a cost of 1
  while (arcIdx < arcsEnd) {
    int nbr = arcs[arcIdx++]; // a neighbour
    // update the nbrs cost and mark it as active if its current cost is
    // higher than newCostForNbrs
  }
}
```

Listing 10. Kernel 1 with many global memory operations (coalesced or atomic).

```
...
frontier[tid] = false;
atomicSub(frontierCounter, 1);
...
while (arcIdx < arcsEnd) {
  ...
  // if the cost of the nbr is updated
  atomicAdd(frontierCounter, 1);
}
```

Listing 11. Kernel 2 introduces the frontier counter in global memory.

through that vertex. The frontier array is copied back from the device to the host machine after every iteration. The host machine determines if at least one vertex is active in the frontier, otherwise all shortest paths from the source vertex s have been discovered and the next vertex can be selected as the source.

Kernel 2 introduces a frontier counter in the form of a single integer allocated in global device memory. Atomic writes are used to increment and decrement this value whenever a vertex is added to or removed from the frontier. Instead of the frontier array only the counter value has to be copied back after every iteration and the host does not need to iterate over the array to determine if any vertices are active in the frontier.

For graphs of type (1), this optimization works particularly well. Copying the frontier array and determining if there is more work to do was very expensive with the large number of iterations needed due to the high mean shortest path result values. Graphs of type (2) and (3), however, show the opposite effect. They have a small mean geodesic distance, and the overhead introduced with the atomic operations causes a great loss in the performance.

Kernel 3 allocates an integer in shared memory that is used as a local counter for the changes to the global frontier counter. The threads in a thread block atomically update this value. Once all threads in the block have done their work, the value of the local counter is atomically written back to the global counter. This replaces most of the global atomic operations with atomic operations on shared memory. This technique is illustrated in Section 2.2.3 Listing 4.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
...
// the global memory read needed to determine the vertex
int vertex = verticesSorted[tid];
if (frontier[vertex]) {
...
  // the vertices array now also contains the adjacency-list length
  // it only takes one instruction to load both values into registers
  int2 vertexStruct = vertices[vertex];
  ...
  // iterate over the neighbours
  for (int nbrIdx = 0, arcIdx = vertexStruct.x;
       nbrIdx < vertexStruct.y; nbrIdx++) {
    ...
  }
}
```

Listing 12. Kernel 5 sorts the vertices by their adjacency-list lengths and uses a new data structure.

This kernel shows the opposite effect of the previous version. Surprisingly, the faster shared memory access is not enough to counter the overhead of the shared memory usage for type (1). Graphs of type (2) and (3), on the other hand, benefit considerably from this change.

Kernel 4 uses the data structure described in Section 2.2.1 and Listing 2. It is basically the layout illustrated in Figure 9(f), only that the neighbours-list lengths are stored in the arcs array as the first element of each sub-list.

These changes do not affect any of the tested networks very much. The overhead of this data structure in general and of loading too much data for the degree $k = 10$ of the graphs of type (1) causes it to perform slightly worse. The results for type (2) and (3) show a small benefit when using this data structure.

Kernel 5 uses the data structure illustrated in Figure 9(f). It also introduces an additional array that contains the vertex IDs sorted by the length of the respective neighbours-list. The effect that this has is that all threads in the same warp process neighbours that have a similar or the same number of neighbours, which reduces the warp divergence caused by looping over lists of different lengths. However, it requires an additional coalesced read from global memory to determine the vertex ID processed by each thread, which was equal to the thread ID in the previous kernels.

The reduced warp divergence gives a decent improvement of the execution speed for the graphs of type (1). The degree of these networks does not vary extremely like in the scale-free network, but it still spreads out quite a bit around the mean. Type (2) graphs show almost no change compared with the previous kernel. Interestingly, graphs of type (3) do not benefit from this, most likely because only very few vertices have a very large degree, while most of the vertices have a smaller and quite similar degree. The overhead caused to determine the thread ID appears to be larger than the benefit of the reduced warp divergence.

Kernel 6 introduces an additional frontier update array that is used to store the frontier for the next iteration. This removes the need for some atomic operations on global memory. They are replaced with normal, largely uncoalesced writes to global memory and a device to device memory copy to reset the frontier update array before every iteration except the first one. The frontier counter has been replaced with a frontier flag that is set to true if at least one vertex is still active in the frontier. No atomic operations on shared memory are needed anymore.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
__shared__ int sharedFrontierFlag;
if (threadIdx.x == 0) {
  sharedFrontierFlag = false;
}
__syncthreads();

// the frontier is set to false by the host code, thus
// frontier[vertex] = false is not needed anymore
...
// mark the neighbour as active in the frontier update array
frontierUpdate[nbr] = true;
// no atomic operation needed, as it is always changed to true
// and never to false (if it is modified at all)
sharedFrontierFlag = true;
...
// set the global frontier flag if the shared one is set
__syncthreads();
if (threadIdx.x == 0 && sharedFrontierFlag) {
  *frontierFlag = true;
}

/*** HOST CODE ***/
// the host machine swaps the frontier update and frontier arrays
// and resets the frontier update array and flag before every iteration
int* tmp = d_frontier;
d_frontier = d_frontierUpdate;
d_frontierUpdate = tmp;
cudaMemcpyAsync(d_frontierUpdate, d_frontierReset, nVertices * sizeof(int),
                cudaMemcpyDeviceToDevice, 0);
cudaMemcpyAsync(d_frontierFlag, d_frontierReset, sizeof(int),
                cudaMemcpyDeviceToDevice, 0);
```

Listing 13. Kernel 6 gets rid of several atomic operations and replaces the frontier counter with a frontier flag.

These changes reduced the performance when processing graphs of type (1), probably due to the same reason why they do not perform well with kernel 3. The small performance gains with networks of type (2) and (3) did not justify to keep these changes for the following kernels.

Kernel 7 reduces the number of atomic operations on shared memory. It uses an additional integer variable (i.e. one register per thread) to record the changes made to the frontier counter by every thread. Once a thread is done, it performs one atomic operation to write this value to the shared memory variable introduced with kernel 3, which is finally written to global memory after all the threads in a block have finished their work.

This simple optimization provided a small performance gain for networks of type (1) and (3). Interestingly, type (2) graphs take slightly longer to process with this kernel. This is the only kernel where type (2) and type (3) networks do not show the same behaviour. Nevertheless, this kernel turns out to be our best implementation overall.

As mentioned before, the execution times on the GPU not only depend on the graph size, but also strongly on the degree and the APSP result values. Figure 14 shows the mean APSP values for the graphs. The strong deviations from the mean values for (1) are the reason for the normalization of the timing results as described in Section 4.1.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

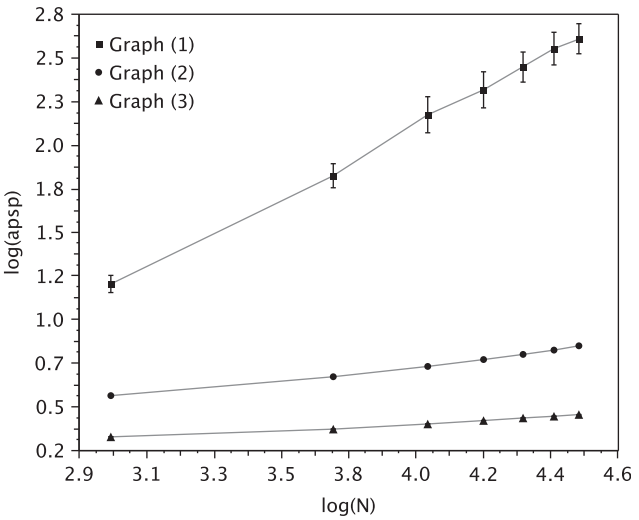


Figure 14. The mean shortest paths for the three tested networks on a log–log scale. The slopes are ≈ 0.995, ≈ 0.223, and ≈ 0.106 for the graphs of type (1), (2), and (3) respectively.

All previous tests were performed with the thread block size set to 256, which is one of several values that gives an occupancy of 100% (see Section 2.1) for the kernels. Figure 15 illustrates that, while this value is a good choice overall, the actual timing results vary considerably depending on the type of the input graph. This shows that it is always a good idea to actually test the implementation with different block sizes and choose the one that performs the best for the given problem.

4.4. Performance results

Now that we have compared the different CUDA implementations, it is time to put them alongside our CPU implementation of Floyd's APSP [43] algorithm. We implemented this algorithm in D, a statically typed multiparadigm programming language whose 'focus is on combining the power and high performance of C and C++ with the programmer productivity of modern languages like Ruby and Python' [50]. Like C(++), it directly compiles to machine code. The execution times of the CPU implementation are illustrated in Figure 16. While the CUDA implementation is rather time-bound, the CPU implementation with its $O(N^3)$ time complexity and $O(N^2)$ memory usage is also heavily memory-bound. Owing to this scaling, the largest networks tested for this article have $N = V = 30\,000$ vertices.

The CPU implementation is single-threaded, thus only one processor core is used by each process. The simulations were performed on Intel® Xeon® X5355 processors running at 2.66 GHz clock speed. 4 GByte of memory were available to every process.

The execution times of the CUDA implementations were measured on an Intel® Core® 2 6600 running at 2.4 GHz with 4 GByte of memory. The graphics card used in this system is an NVIDIA® GeForce® GTX260 with 896 MByte of device memory.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

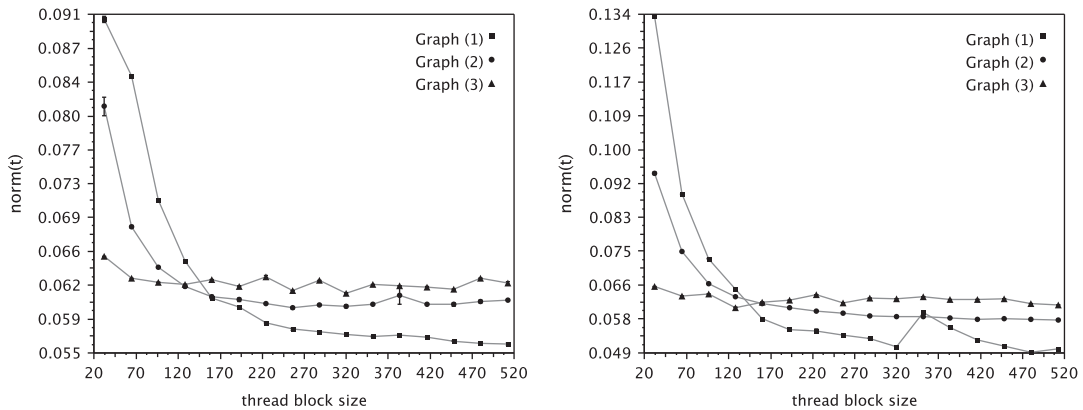


Figure 15. Normalized execution times for graphs of size $N = 10000$ on the left and $N = 30000$ on the right processed with kernel 7 and an increasing number of threads per block. The block sizes are multiples of 32 up to the maximum of 512. Each data point is the mean time taken to process the same graph instance 5 times. The standard deviations are represented by error bars. These are mostly hidden by the symbols, as any deviation is mainly caused by background processes run by the operating system and these have been deactivated where possible. The results differ considerably between the different input graph types. The size of the input graphs, on the other hand, does not appear to affect the results much. Only the sudden increase in execution time of type (1) graphs around a block size of 352 that only occurs in the $N = 30000$ case stands out.

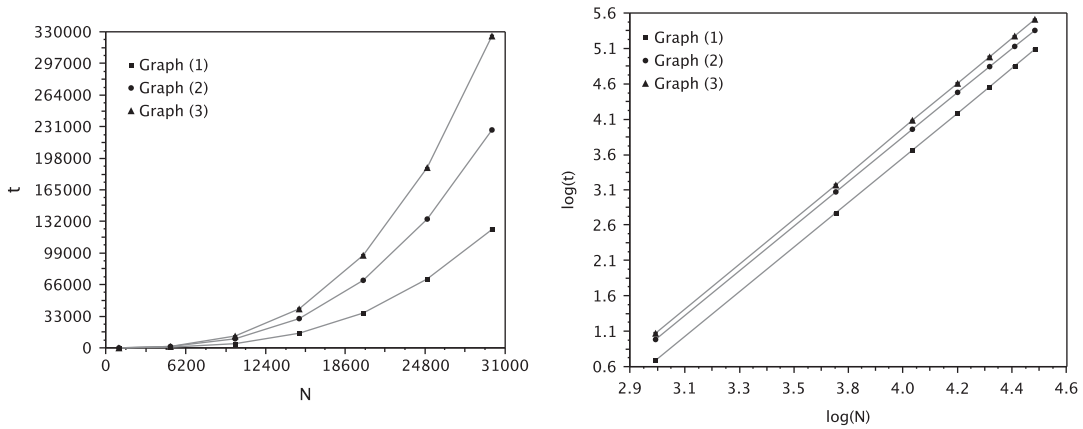


Figure 16. The execution times of the CPU implementation on the left and the same data on a log–log scale on the right. Like the CUDA kernels, the execution times scale with the network size. The slope for all graphs is ~ 3.

The actual execution times in seconds and the speed-ups gained by using the GPU are recorded in Table II. With a performance gain of up to ~ 3000 times, the CUDA implementation clearly outperforms the CPU implementation in every test. These results also show that the breadth-first search-based algorithm used on the GPU depends much more on the input data than Floyd's

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Table II. Performance comparisons between the CPU implementation of Floyd's APSP algorithm and the CUDA implementation (kernel 7).

Network size	1k	5k	10k	15k	20k	25k	30k
Graph (1)							
CPU	5	585	4602	15 467	36 546	71 462	123 624
GPU/CUDA	2	39	183	419	825	1470	1938
Speed-up	3×	15×	25×	37×	44×	49×	64×
Graph (2)							
CPU	10	1175	9166	30 243	70 237	134 264	227 808
GPU/CUDA	1	4	11	20	33	51	73
Speed-up	10×	294×	833×	1512×	2128×	2633×	**3121**×
Graph (3)							
CPU	12	1506	12 123	40 798	96 695	188 349	325 355
GPU/CUDA	3	32	95	184	292	417	578
Speed-up	4×	47×	128×	222×	331×	452×	563×

The execution times are given in seconds and are meaningful to two significant digits.

algorithm. The performance varies with the average degree and the mean shortest path result value of the input network. The slopes on the log–log plots and the timing results stated in the table also show that the GPU implementation scales better than the CPU implementation for all tested networks.

5. DISCUSSION

One of the problems in fairly assessing a specific piece of hardware such as a GPU is that the optimal algorithm we might use for an application on it might be appreciably different from the optimal one for a general purpose CPU. We have quoted specific speed-up figures compared with our CPU implementations just to set them in context. Their fair and true value are likely more in the relative speeds we have been able to achieve with the different kernel algorithm approaches to our two applications. There is no doubt that the performance of the CPU implementations could be improved by the use of optimization techniques such as multi-threading or cache-rearrangement.

One area of difficulty with the GPU platforms at present is the lack of instrumentation and timing facilities. It is not simple to monitor what the GPU is doing without interfering with the operations. The CUDA libraries include a profiler that can be used to gather data about memory access. The speed-up figures we quote must therefore be considered carefully in context. Generally, we have managed to achieve speed comparisons that we believe can be safely interpreted to around two significant figures. This is sufficient to determine which kernel algorithm is the best in each problem size regime.

For the applications we have considered, the precision of floating point is not especially important. Generally, we would expect to solve field equations as discussed in Section 3 using the native

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

precision of the CPU or GPU. In most cases that will be 64-bits for modern CPUs, but some GPUs still only offer 32-bit floating point. It is likely that almost all modern GPUs in the future will offer 64-bits or higher however.

We have explored as many of the memory and threading features of the GPUs architectures we had to hand as far as possible. We are still considering applications that might be able to make good use of constant memory—those that have a large number of slowly varying parameters for example.

An inevitable issue with data-parallel computing is that a real application may not lend itself to a single optimal data layout. Different parts of the application may favour different data layouts. An advantage of the combined CPU+GPU programming approach is that it is relatively easy to program data layout conversions to use the one that is optimal for the particular GPU kernel. Some scientific applications or speed optimized libraries might therefore have several different kernel routines, ostensibly doing the same calculation, but with each optimized for a different data layout.

In general, we have been impressed with the CUDA language and environment. It seems likely that this style of integration with a programming language like C/C++ or D will be supportable for some time to come, and will be worthwhile for applications programmers who are able to justify software development at those levels.

Although we have approached two applications that are of intrinsic scientific interest for simulations in computational science, we believe that these sorts of algorithms are also of strong relevance to games engines. Simulating fields for rendering atmospheric or water effects can use a regular field equation approach. Navigating agents in a game or character scenario can also make use of path-finding algorithms. GPUs were originally developed for graphical algorithm optimization, but it seems clear that they will find increasing uses as parts of simulated environment generation packages, physics engines [51] and other higher-level application components of interactive games and animation systems.

6. SUMMARY AND CONCLUSIONS

We have described some key characteristics of GPUs in general and our use of the CUDA programming language to implement scientific applications. We have reported on various performance optimization techniques involving both multi-threading and combinations of different memory arrangements. We have focused on NVIDIA's GPU hardware and CUDA development language as being the most advanced at the time of writing, although it is clear that there are also other past and emerging offerings from other vendors.

One specific regular-mesh application that we investigated used a finite-difference solver for a partial differential field equation such as the CH system. Not surprisingly, we found that a parallel geometric decomposition involving a large number of parallel threads that execute independently performed well on the GPUs. However less obviously, the performance was also heavily dependent on how effectively we used the different sorts of memory available on the GPU. We showed that a sensible use of fast local texture memory was critical to achieving the most dramatic performance improvements.

We also investigated graph and network applications as typified by computation of the all-pairs, shortest-path metric on various sorts of scale-free and other graph instances. These algorithms

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

are less obviously parallelizable, but we obtained significant performance speed-ups again from arranging the algorithm so that it could best exploit a multi-threaded architecture such as the particular GPUs that we used. The correct approach was less obvious for these problems and we experimented with several different parallelization techniques. We found that it was possible to achieve a very significant speed-up over a typical CPU by having the application's network data suitably arranged in the GPU's fast and local memory.

This effect is reminiscent of the *super-linear speed-ups* commonly reported in the late 1980s when just having a multiprocessor computer with more memory overall than was typically found on a single-processor system, allowed applications to avoid slow memory or very slow disk swapping. Similar effects were not uncommon in the early 1990s when sophisticated multi-level caches became prevalent in CPUs. Optimizing compilers can make some headway in cache prediction optimizations, and it is likely that this is easier to do for a regular mesh problem than for an arbitrary graph problem. This likely explains why we have been able to hand optimize a more dramatic speed-up (relative to typical CPU performance) for graph problems in our hand-optimized GPU CUDA code than we could with the regular mesh application.

We found that the GPU architectures and their available features very interesting to use, and we observe that a language like CUDA makes these ideas much more accessible to applications programmers than has previously been the case. Nevertheless parallel computing on GPUs remains non-trivial to use. Not all applications programmers will find the performance gains worth the additional thinking and development effort necessary to produce an optimized and maintainable code. The effort will likely continue to be worthwhile for game developers and those interested in fast and cheap scientific computing resources, like ourselves.

The economics of the games industry seems to have been the main driver for the low cost and wide availability of GPUs. It is interesting that data-parallel computing is once again attracting attention and development effort as the ideas it embodies become available in a useful and relatively easily exploitable manner on affordable hardware. We speculate that in the future more data-parallel concepts, as used now in *GPUs*, will be absorbed onto future multi-core *CPUs*.

It remains to be seen however, exactly what tool developments will be needed to ensure that these features can be widely used by applications programmers. It is likely that sophisticated scientific libraries using data-parallel ideas will be ported to such future chips. It is still not yet clear how many of the data-parallel ideas can be made available as standard in general-purpose language compilers. Modern high-level languages such as Fortress [52] may be able to embed run-time library components that have been optimized for such hardware. In the meantime, other mid level approaches such as using scripting language bindings such as PyCUDA [53] may be a worthwhile approach. There does still seem to be a need for general-purpose programming languages that better integrate data-parallel ideas. There will also continue to be interesting research opportunities on tradeoff analyses to determine what balance of computation is best located on a CPU and its co-GPU(s).

ACKNOWLEDGEMENTS

It is a pleasure to thank C. H. Ling, I. A. Bond and C. J. Scogings for the helpful discussions.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

REFERENCES

1. Fox GC, Williams RD, Messina PC. *Parallel Computing Works*! Morgan Kaufmann Publishers, Inc.: Los Altos, CA, 1994. ISBN 1-55860-253-4.
2. Wilding N, Trew A, Hawick K, Pawley G. Scientific modeling with massively parallel SIMD computers. *Proceedings of the IEEE* 1991; **79**(4):574–585. DOI: 10.1109/5.92050.
3. Hillis WD. *The Connection Machine*. MIT Press: Cambridge, MA, 1985.
4. Johnsson SL. Data parallel supercomputing. *Preprint YALEU/DCS/TR-741*, Yale University, New Haven, CT, September 1989. *The Use of Parallel Procesors in Meteorology*. Springer: Berlin.
5. Yau HW, Fox GC, Hawick KA. Evaluation of high performance fortran through applications kernels. *Proceedings of the High Performance Computing and Networking 1997*, Vienna, Austria, 1997.
6. Hawick KA, Havlak P. High performance Fortran motivating applications and user feedback. *Technical Report NPAC Technical Report SCCS-692*, Northeast Parallel Architectures Center, January 1995.
7. Fox G. *Fortran D as a Portable Software System for Parallel Computers*, Center for Research on Parallel Computation, Rice University, Houston, Texas, TX, CRPC-TR91128, June 1991.
8. Chapman B, Mehrotra P, Zima H. Vienna Fortran—a Fortran language extension for distributed memory multiprocessors. *ICASE Interim Report 91-72*, Institute for Computer Applications in Science and Engineering, NASA, Langley Research Center, Hampton, Virginia, September 1991.
9. Bozkus Z, Choudhary A, Fox GC, Haupt T, Ranka S. Fortran 90D/HPF compiler for distributed-memory MIMD computers: Design, implementation, and performance results. *Proceedings of the Supercomputing '93*, Portland, OR, 1993; 351–360.
10. Bogucz EA, Fox GC, Haupt T, Hawick KA, Ranka S. Preliminary evaluation of high-performance Fortran as a language for computational fluid dynamics. *Proceedings AIAA Fluid Dynamics Conference*, *AIAA 94-2262*, Colorado Springs, 1994.
11. Pacheco PS. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc.: Los Altos, CA, 1996.
12. Chandra R, Dagum L, Kohr D, Menon R, Maydan D, McDonald J. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc.: Los Altos, CA, 2001.
13. Fatahalian K, Houston M. A closer look at GPUs. *Communications of the ACM* 2008; **51**(10):50–57.
14. Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, Purcell T. A survey of general-purpose computation on graphics hardware. *Eurographics 2005*, *State of the Art Reports*, 2005; 21–51.
15. Langdon W, Banzhaf W. A SIMD interpreter for genetic programming on GPU graphics cards. *Proceedings of the EuroGP* (*Lecture Notes in Computer Science*, vol. 4971), O'Neill M, Vanneschi L, Gustafson S, Alcazar AE, Falco ID, Cioppa AD, Tarantino E (eds.). Springer: Berlin, 2008; 73–85.
16. Messmer P, Mullowney PJ, Granger BE. GPULi0b: GPU computing in high-level languages. *Computing in Science and Engineering* 2008; **10**(5):70–73.
17. Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with CUDA. *ACM Queue* 2008; **6**(2):40–53.
18. Buck I, Foley T, Horn D, Sugerman J, Fatahalian K, Houston M, Hanrahan P. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics* 2004; **23**(3):777–786. DOI: http://doi.acm.org/10.1145/1015706.1015800.
19. AMD. *ATI CTM Guide*, 2006.
20. McCool M, Toit SD. *Metaprogramming GPUs with Sh*. A K Peters, Ltd.: Wellesley, MA, U.S.A., 2004.
21. Khronos Group. OpenCL 2008. Available at: URL: http://www.khronos.org/opencl/.
22. Playne DP, Gerdelan AP, Leist A, Scogings CJ, Hawick KA. Simulation modelling and visualisation: Toolkits for building artificial worlds. *Research Letters in the Information and Mathematical Sciences* 2008; **12**:25–50.
23. Stock MJ, Gharakhani A. Toward efficient GPU-accelerated N-body simulations. *Forty-Sixth AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV, U.S.A., *AIAA 2008-608*, 2008.
24. Kavanagh GD, Lewis MC, Massingill BL. GPGPU planetary simulations with CUDA. *Proceedings of the 2008 International Conference on Scientific Computing*, Las Vegas, NV, U.S.A., 2008.
25. NVIDIA® Corporation. *CUDA™ 2.0 Programming Guide*, 2008. Available at: http://www.nvidia.com/ [last accessed November 2008].
26. Cahn JW, Hilliard JE. Free energy of a nonuniform system. I. Interfacial free energy. *The Journal of Chemical Physics* 1958; **28**(2):258–267.
27. Chen LQ, Shen J. Applications of semi-implicit Fourier-spectral method to phase field equations. *Computer Physics Communications* 1998; **108**:147–158.
28. Hawick KA, Playne DP. Modelling and visualizing the Cahn–Hilliard–Cook equation. *Proceedings of 2008 International Conference on Modeling*, *Simulation and Visualization Methods* (*MSV'08*), Las Vegas, NV, 2008.
29. Albert R, Jeong H, Barabsi AL. Diameter of the World-Wide Web. *Nature* 1999; **401**(6749):130–131.

Copyright © 2009 John Wiley & Sons, Ltd.

30. Jeong H, Tombor B, Albert R, Oltvai Z, Barabsi AL. The large-scale organization of metabolic networks. *Nature* 2000; **407**(6804):651–654.
31. Fell DA, Wagner A. The small world of metabolism. *Nature Biotechnology* 2000; **18**(11):1121–1122.
32. Wagner A, Fell DA. The small world inside large metabolic networks. *Proceedings of the Royal Society B* 2001; **268**(1478):1803–1810.
33. Milgram S. The small-world problem. *Psychology Today* 1967; **1**:61–67.
34. Liljeros F, Edling C, Amaral L, Stanley H, Aberg Y. The web of human sexual contacts. *Nature* 2001; **411**(6840): 907–908.
35. Liben-Nowell D, Kleinberg J. Tracing information flow on a global scale using Internet chain-letter data. *Proceedings of the National Academy of Sciences* 2008; **105**(12):4633–4638.
36. Newman MEJ. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences* 2001; **98**(2):404–409.
37. Newman MEJ. Ego-centered networks and the ripple effect. *Social Networks* 2003; **25**(1):83–95.
38. Xu J, Chen H. The topology of dark networks. *Communications of the ACM* 2008; **51**(10):58–65. DOI: http://doi.acm.org/10.1145/1400181.1400198.
39. Newman MEJ. The structure and function of complex networks. *SIAM Review* 2003; **45**(2):167–256.
40. Watts DJ, Strogatz SH. Collective dynamics of 'small-world' networks. *Nature* 1998; **393**(6684):440–442.
41. Barabsi AL, Albert R. Emergence of scaling in random networks. *Science* 1999; **286**(5439):509–512.
42. Price DJd. Networks of scientific papers. *Science* 1965; **149**(3683):510–515. DOI: 10.1126/science.149.3683.
43. Floyd RW. Algorithm 97: Shortest path. *Communications of the ACM* 1962; **5**(6):345.
44. Dijkstra E. A note on two problems in connexion with graphs. *Numerische Mathematik* 1959; **1**(1):269–271.
45. Newman MEJ, Strogatz SH, Watts DJ. Random graphs with arbitrary degree distributions and their applications. *Physical Review E* July 2001; **64**(2):26–118. DOI: 10.1103/PhysRevE.64.026118.
46. Abdo AH, de Moura APS. Clustering as a measure of the local topology of networks. *Technical Report*, Universidade de São Paulo, University of Aberdeen, February 2008. Available at: http://arxiv.org/abs/physics/0605235v4.
47. Harish P, Narayanan P. Accelerating large graph algorithms on the GPU using CUDA. *High Performance Computing—HiPC 2007*: *14th International Conference*, *Proceedings*, vol. 4873, Aluru S, Parashar M, Badrinath R, Prasanna V (eds.). Springer: Goa, India, 2007; 197–208.
48. Watts DJ. *Small Worlds*: *The Structure of Networks between Order and Randomness*. Princeton University Press: Princeton, NJ, 1999.
49. Cormen TH, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms* (2nd edn). MIT Press: Cambridge, MA, 2001.
50. Bright W. *D Programming Language*, Digital Mars, 2008. Available at: www.digitalmars.com/d/ [accessed November 2008].
51. Conger D, LaMothe A. *Physics Modeling for Game Programmers*. Thompson: Boston, MA, U.S.A., 2004.
52. Allen E, Chase D, Flood C, Luchangco V, Maessen JW, Ryu S, Steele JrGL. Project Fortress—A multicore language for multicore processors. *Linux Magazine*, September 2007; 38–43.
53. Klockner A. *PyCUDA v0.91.1 Documentation*, December 2008. Available at: URL: http://documen.tician.de/pycuda/.

Copyright © 2009 John Wiley & Sons, Ltd.