Joint Forces: From Multithreaded Programming to GPU Computing

Frank Feinbube, Peter Tröger, and Andreas Polze,
Hasso Plattner Institute

// Using graphics hardware to enhance CPU-based standard desktop applications is a question not only of programming models but also of critical optimizations that are required to achieve true performance improvements. //

TWO MAJOR HARDWARE TRENDS make parallel programming a crucial issue for all software engineers today: the rise of many-core CPU architectures and the inclusion of powerful graphics processing units (GPUs) in every desktop computer. Modern CPUs use the ever-increasing transistor count predicted by Moore's law not only for larger caches and improved prediction logic but also for adding more parallel execution units ("cores") per chip. Projections show that future processor generations will offer hundreds or even thousands of cores per socket.[1] However, an application can only benefit from this development if it's prepared for parallel execution.

Currently, CPUs support parallel programming by assigning threads to different tasks and coordinating their activities through hardware-supported locking primitives in shared memory. Newer programming models also consider the nonuniform memory architecture (NUMA) of modern desktop systems, but they still rely on the underlying concept of parallel threads accessing one flat address space. CPU hardware is optimized for such coarse-grained, task-parallel programming with synchronized shared memory.

By contrast, GPU cards are mainly designed for fine-grained, data-parallel computation. The input data—not the computational task set—drives the algorithm design. Graphics processing is an embarrassingly parallel problem,[9] in which the serial code for initialization, synchronization, and result aggregation is very small in comparison to the parallel code. GPU hardware is optimized for this kind of load. It aims at combining a maximum number of simple parallel-processing elements, each having only a small amount of local memory. For example, the Nvidia Geforce GTX 480 graphics card supports up to 1,536 GPU threads on each of its 15 compute units. So, at full operational capacity, it can run 23,040 parallel execution streams.

The use of a GPU compute device in a desktop application is typically driven by the need to accelerate a specific computation. We can therefore assume that GPU-aware programs are seldom implemented from scratch. Besides the data-driven algorithm design, the GPU code compulsory demands optimizations to achieve scalability and performance improvements. We've collected a commonly agreed set of these strategies.

From CPU to GPU

Applications running on a computer can access GPU resources with the help of a control API implemented in user-mode libraries and the graphics card driver. The control application, which developers can write in high-level

TABLE 1

CPU, OpenCL, and CUDA terminology mappings.

CPU			OpenCL			CUDA		
Platform level	Memory level	Execution level	Platform level	Memory level	Execution level	Platform level	Memory level	Execution level
Symmetric multipro-cessing system	Main memory	Process	Compute device	Global and constant memory	Index range (NDRange)	Device	Global and constant memory	Grid
Processor	—	—	Compute unit	Local memory	Workgroup	Multipro-cessor	Shared memory	Thread block
Core	Registers, thread local storage	Thread	Processing element	Registers, private memory	Work items = kernels	Scalar pro-cessor	Registers, local memory	Threads = kernels

languages, submits extended C++ code (called a *work item*) to the card through this API. The driver starts the appropriate activities on the card asynchronously with the host application. Memory transfers between host and card are also expressed as commands to the driver.

The two major vendors in the GPU market, Nvidia and AMD (which acquired ATI and its GPU product line in 2006), provided their own API definitions in the past. The most famous approach is still CUDA, the Nvidia-specific combination of programming model and driver API. Recently, a consortium of leading GPU-interested companies defined the Open Computing Language (OpenCL), a vendor-neutral way of accessing compute resources. Open CL specifies the programming model with C++ language extensions and the control API to be implemented by the driver.[2] OpenCL implementations are available for major operating systems and recent GPU hardware.

Table 1 shows a simplified mapping of terms and concepts in the GPU and CPU worlds. OpenCL's lower-level terminology includes processing elements that can execute a set of code instances denoted as work items, or kernels (Nvidia's original term). Each processing element on the card has a large register set (currently 8,192 to 32,768) and some private memory. The OpenCL specification groups processing elements as compute units, each of which shares local memory (currently 16 to 48 Kbytes). It combines work items into workgroups, each of which uses the local memory for coordination and data exchange. All work items in all workgroups share a common global memory, which can be accessed from CPU and GPU code.

On execution, the GPU runtime environment informs each work item about the range of data items it must process. The OpenCL interface expects the data partitioning for parallel computations to be defined by the total number of work items plus the size of workgroup chunks. Each chunk is assigned to one compute unit. The developer can formulate partitioning in different dimensions, but the common strategy is to rely on a 2D matrix representation of the input data. The subtle difference from threads running on a CPU is that work items on one compute unit aren't executed independently but in a lock-step fashion, also known as single instruction, multiple data (SIMD) execution.

Sudoku Example Application

Figure 1 lists example code for a small Sudoku validator, which checks whether a given solution is valid according to the game rules. A Sudoku field typically consists of 3 × 3 subfields with each having 3 × 3 places. Three facts make this problem a representative example of algorithms appropriate for GPU execution:

- data validation is the primary application task,
- the computational effort grows with the game field's size (that is, the problem size), and
- the workload belongs to the GPU-friendly class of embarrassingly parallel problems that have only a very small serial execution portion.

Other examples from this well-parallelizable class of algorithms include fractal calculations, brute-force searches in cryptography, and genetic algorithms. The matching of algorithm classes to hardware architectures is still an active research topic,[3] but other problem classes might also benefit from using GPUs by restructuring the problem or optimizing the data. In any case, identifying data-parallel parts in the application is always a good starting point to work on the GPU version of an algorithm.

Figure 1 shows our initial multithreaded version of the Sudoku validator. The `isValid` function checks if the presented Sudoku solution fulfills all relevant rules. It uses the `contains` function, which starts `n*n` threads (one thread per row). The double loop in the `invalidColumn` function is a perfect candidate for parallelization, since all iterations run on different data parts.

Figure 2 shows a straightforward reformulation of our OpenCL example. After finishing, the host can read the results from the memory buffer. The invalidColumn function is now realized as a GPU work item. We pass the input field and the shared variable as global memory pointers. The *y*-loop is replaced as well, so we can use n*n*n*n work items instead of n*n threads. This is more appropriate for the massively parallel GPU hardware. We use the work-item ID to derive the x and y coordinate and then apply the normal checking algorithm.

Even though the example is simple and not yet optimized, the GPU-enabled version already performs better than the multithreaded version (see Figure 3). For small problem sizes with this example, the AMD R800 GPU and the Nvidia GT200 GPU are about an order of magnitude faster than the Intel E8500 CPU. For bigger problem sizes, the gap becomes smaller.

Best CPU-GPU Practices

To push the performance of GPU-enabled desktop applications even further requires fine-grained tuning of data placement and parallel activities on the GPU card. Table 2 shows a synthesis of relevant optimization strategies, collected from documentation, vendor tutorials, and personal experience. We've compiled an exhaustive list of sources and related work focusing on GPU computing, available at www.dcl.hpi.uni-potsdam.de/research/gpureadings.

Every optimization must account for different views of the application requirements. Otherwise, the optimization doesn't pay off in significant performance improvements. First, you must focus on the parallelization task from a logical perspective. From this view, optimization is related to algorithm design, memory transfer, and control-flow layout. Second is memory usage. To achieve maximum bandwidth utilization, you must consider memory type hierarchies and access optimizations. Next, you can consider the hardware model perspective. To achieve maximal occupancy and instruction throughput, the relevant optimizations relate to sizing, instruction types, and supported numerical data formats.

Algorithm Design

The GPU's SIMD architecture requires splitting the data-parallel problem into many tiny work items. In addition to the parallelism on one device, all a host's compute devices can execute independently and asynchronously from the host itself. You can even run code on the card while reading or writing a device's data buffers.

Hardware directly supports mathematical operations inside the work items and executes them quickly. On the other hand, the number of registers available per work item and the bandwidth for communicating with off-chip memory are limited. This results in the (counterintuitive) recommendation to recompute values instead of storing them in registers or in device memory.

Work items should be simple, without complex control or data structures. Because the GPU hardware has no stack, all function calls are inlined. With nested function calls, all variables are kept in registers, so the number of registers needed per work item can increase dramatically. This can reduce hardware utilization efficiency when increased workloads cause your application to create additional work items. So you should try to keep the code small and simple.

Memory Transfer

The control application must initially copy input data for GPU computation, using the PCI Express bus. The bus has a comparatively low bandwidth, so memory transfers between host and GPU are expensive. Consequently, you should try to reduce the amount of data that needs to be transferred upfront.

```
bool isValid (int* field ) {
 return
   // check for numbers > n*n or < 0
   !contains(field, invalidNumber) &&
   // no duplicates in a row
   !contains(field, invalidRow) &&
   // no duplicates in a column
   !contains(field, invalidColumn) &&
   // no duplicates in a subfield
   ! contains ( field, invalidSubfield);
}

bool invalidFieldDetected = false;

bool contains (int* field, ThreadStartRoutine checkFunction ) {
 invalidFieldDetected = false;
 for (int x=0; x < n*n ; x++) {
  ThreadParameter* tp = new ThreadParameter ();
  tp->threadId = x; tp->field = field;
  ts[x] = StartThread(checkFunction, tp);
 }
 for (int x=0; x < n*n; x++) {WaitForThread ( ts[x], INFINITE); }
 return invalidFieldDetected;
}

unsigned long invalidColumn (LPVOID data) {
 ThreadParameter* p = ( ThreadParameter*) data;
 int* field = p->field;
 int x = p->threadId;
 for (int y = 0; y < n*n-1; y++)
  for (int y2 = y+1; y2 < n*n; y2++)
   if (field [x+y*n*n] == field [x+y2*n*n])
    invalidFieldDetected = true;
}
...
```

FIGURE 1. Multithreaded Sudoku validator. The algorithm checks for all Sudoku rules that a valid solution must fulfill.

```
bool contains (int* field, cl_kernel checkKernel ) {
  // ... initializing memory buffers ...
  // ... set arguments ...
  size_t local = 32;
  size_t global = ((workItemCount/local)+1)* local;
  // start parallel execution of work items
  clEnqueueNDRangeKernel(commands, checkKernel, 1, NULL, &global, &local, 0, NULL, NULL);
  // ... wait for the commands to finish ...
  // ... read results from buffers, cleanup ...
  return (invalidFieldDetected [0] == 1);
}
__kernel void invalidColumn (
  __global int* field,
  __global int* invalidFieldDetected,
  const unsigned int n)
{
  uint id = get global id (0);
  if (id >=n*n*n*n) return;                                       // range checking
  uint x=id/(n*n);
  uint y=id%(n*n);
  if (y >=n*n-1) return;                                          // range checking
  for (int y2 = y+1; y2 < n*n; y2++)
   if (field [x+y*n*n] == field [x+y2*n*n])
     invalidFieldDetected [0] = 1;
}
```

FIGURE 2. OpenCL version of the Sudoku validator. In comparison to the multithreaded version, this code was modified to run the parallelizable functions on a GPU card.

If multiple work items use the same input data or run one after another, you can improve performance by moving data to the device once, chaining work items, and reading the results back after they have been executed. Another way to reduce memory bandwidth usage is to move more operations from the host to the GPU. You can use separate command queues to the same device to overlap memory transfers and compute operations, thus hiding bandwidth latencies. If little or no data reuse occurs within or between work items, you can make host-allocated memory available to the GPU code. This eliminates the need to copy and synchronize data.

Control Flow

CPU-optimized programs often use several execution paths and choose the right one according to the current context to reduce workload. The CPU world relies upon condition variables, branches, and loops. GPU compute devices demand a completely different approach.

For example, each hardware-processing element of the Nvidia Geforce GTX 480 graphics card can execute up to 48 work items simultaneously. This holds as long as all work items on a processing element branch equally. Otherwise, their execution is serialized, resulting in a worst-case execution time 48 times greater than normal if all work items use different execution paths. To avoid branching, even in the case when the control flow of some work items is divergent, the compiler can use predication. You can help the compiler with this by making the divergent parts within branching conditions as short as possible.

For simple loops, the compiler will use unrolling and predication automatically. Special compiler pragmas can also trigger this behavior. For complex loops, unrolling should be performed manually.[4]

Memory Types

OpenCL specifies different types of memory. Registers and local memory are on-chip and therefore the fastest memory types. Both are scarce resources, so limiting their use keeps the number of work items per compute unit high.

Each work item has additional private memory, which is mapped to off-chip device memory and therefore slow. The same holds for global memory. Even the improved caching support expected in future cards doesn't save the GPU developer from considering the memory hierarchy.

Graphic cards also have special cached read-only memory: the constant memory and the texture memory. Storing the arguments for a work-item execution is a good use case for constant memory because it saves local memory. Texture memory applies well to mid-size data structures with lots of reuse and a hard-to-predict access pattern, because its cache has a 2D layout.

We strongly recommend using local memory as cache for global memory. It significantly improves performance if data is reused. Arrays that are allocated per work item are stored in private memory and therefore reside off-chip. They're ideal candidates to store in local memory. For example, each work item could exclusively use a subsection of local memory and store its arrays there. This lets the hardware use collaborative writes.

Memory Access

Every memory access, especially to off-chip memory, is performed synchronously and therefore stalls the requesting workgroup. GPUs hide this latency by scheduling other runnable workgroups on the processing element during the waiting phase. In contrast to the case with CPUs, such context switch-

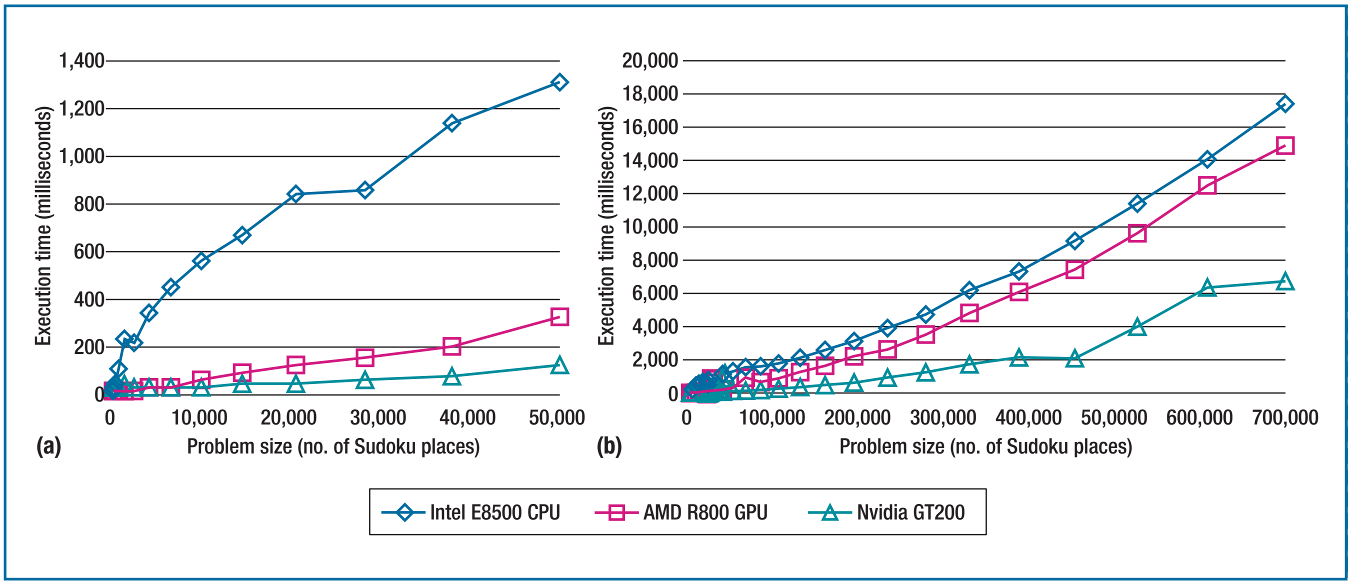

FIGURE 3. Execution time of the Sudoku validation on different compute devices: (a) problem size of 10,000 to 50,000 possible Sudoku places and (b) problem size of 100,000 to 700,000 Sudoku places.

ing is free of costs so long as the compute unit has enough runnable workgroups. On the other hand, registers and shared memory are very fast and can be used to decrease memory latencies. You must consider this trade-off and evaluate which approach best fits your algorithm.

You can optimize parallel memory accesses by following some specific requirements for memory alignment and density, called *coalesced memory access*. The detailed requirements are hardware-specific. In the best case, all work items on a compute unit can get their global memory data in a single fetch. In the worst case, each work item requires a separate fetch. Coalescing is therefore crucial, and you should apply it whenever possible. Because private memory is also mapped to device memory, you must coalesce access to it as well.[5]

Even though the work-item code can access shared memory without considering coalescing, you must still think about memory-bank conflicts. If these occur, the accesses will be serialized, and the performance might be degraded by one order of magnitude. The latest profilers detect such memory-bank conflicts. You can reduce them by using an appropriate alignment to access memory addresses.

If you want to optimize access to texture memory, consider its 2D cache layout. Workgroups can also be represented in a 2D fashion, so testing different aspect ratios to find the optimal workgroup layout can be helpful.

Sizing

The workgroup size should be a multiple of the compute unit's native execution size to ensure optimal utilization of the compute unit for coalescing—that is, at least 32 running work items for Nvidia cards and 64 for AMD cards.

Many cases require some benchmarking with different local workgroup sizes to find what's most appropriate for your kernel.

Instructions

Because GPUs were originally designed for specific mathematical workloads, they have to emulate some of the available CPU instructions. For example, try to replace GPU integer modulo operator or integer division operations with bitwise operations like shift operators whenever possible.

On the other hand, some crucial GPU operations are very fast, such as fused multiply-add and reciprocal square root. If you have a choice, use these functions wherever possible.

The performance impact of most instructions is hardware-specific. AMD's GPU compute devices are vector machines, so vectorizing your code using OpenCL vector types and instructions can improve your work-item code's performance drastically.[5] In contrast, Nvidia uses a scalar architecture in its cards, which implies using more work items instead of larger vectors per work item.[5]

Precision

Although current GPUs are really fast on single-precision floating-point operations, they miss or have limited support for double-precision operations. First-generation OpenCL-enabled GPUs have no double-precision hardware units at all, so they convert doubles to floats. These unnecessary conversions degrade performance. Even with the latest OpenCL-enabled cards, the amount of double-precision floating-point units is still smaller than the number of single-precision arithmetic units.

Nvidia's native math library provides low-precision math functions like `native_sin(x)` and `native_cos(x)` that map

TABLE 2

Best practices for GPU code optimization (bold font indicates crucial practices).

Strategy	Importance	
	ATI	Nvidia
Algorithm design		
Use host asynchronously	Low	Low
Recompute instead of transferring	**High**	**High**
Use simple work items	Medium	Medium
Memory transfer		
Reduce input data size	**High**	**High**
Chain work items	Medium	Medium
Move more operations to GPU	Medium	Medium
Overlap transfer and compute time	Medium	Medium
Pass memory directly to the work item	Low	Low
Control flow		
Avoid divergent branching	**High**	**High**
Support predication	Medium	Low
Memory types		
Reduce per-thread memory usage	**High**	**High**
Store arguments in constant memory	Low	Low
Use OpenCL images for data structures with hard-to-predict access patterns	Low	Low
Use local memory as a cache for global memory	**High**	**High**
Avoid global arrays on work-item stacks	Medium	Medium
Collaboratively write to local memory	Medium	Medium
Memory access		
Consider trade-off between work-item count and memory usage	**High**	**High**
Avoid memory-bank conflicts	**High**	**High**
Access texture memory with appropriate workgroup aspect ratio	Low	Low
Sizing		
Local workgroup size should be a multiple of the processing element's execution size	**High**	**High**
Ensure device memory accesses are coalesced	**High**	**High**
Evaluate different workgroup sizes	Medium	Medium
Instructions		
Use shift operations instead of division and modulo	Low	Low
Use fused multiply add when appropriate	Low	Low
Use vector types and operations	**High**	–
Precision		
Avoid automatic conversion of doubles to floats	Low	Low
Use the native math library for low-precision tasks	–	Medium
Use build options that trade precision for speed	–	**High**

directly to the hardware. Some compilers also provide options to reduce precision and thereby improve speed automatically.[6]

Developer Support

Nvidia and AMD offer software development kits with different C compilers for Windows- and Linux-based systems. Developers can also utilize special libraries, such as AMD's Core Math Library, Nvidia's libraries for basic linear algebra subroutines and fast Fourier transforms, OpenNL for numerics, and CULA for linear algebra. Nvidia's Performance Primitives library is intended to be a collection of common GPU-accelerated processing functionality.

Nvidia and AMD also provide big knowledge bases with tutorials, examples, articles, use cases, and developer forums on their websites. Mature profiling and debugging tools are available for analyzing optimization effects—for example, Nvidia's Parallel Nsight for Microsoft Visual Studio, AMD's ATI Stream Profiler, or the platform-independent alternative gDEBugger. AMD's StreamKernel Analyzer supports the study of GPU assembler code for different GPUs and detects execution bottlenecks.

The GPU market continues to evolve quickly, but some trends seem clear. Nvidia has already started distinguishing between GPU computing as an add-on for normal graphic cards and as a sole-purpose activity on processors such as its Tesla series. Nvidia extends Fermi-architecture-based cards with a cache hierarchy for compute units and native double-precision support.

Some of the best practices we've presented are efforts to circumvent current hardware limitations, which might not be relevant with future cards. Other issues are inherent to GPU computing. In many domains, dedicated libraries provide high-level functionality at great performance while hiding the complexity of using GPU computing power. New language extensions emerge that reduce the GPU programming burden and automate some of the optimizations we've described. However, this is still an open research field for both software engineering and programming language design. Higher-level languages like Java and C# can also benefit from GPU computing by using GPU-based libraries.

The journey to heterogeneous computing has just begun, but it could lead to a new software engineering style that considers the parallel execution of software on heterogeneous compute devices as the default case.

ABOUT THE AUTHORS

FRANK FEINBUBE is PhD candidate at the Hasso Plattner Institute and a member of the HPI Research School on Service-Oriented Systems Engineering. His research focuses on parallel programming models, especially on new ways of programming complex heterogeneous parallel architectures. Feinbube has an MSc in IT systems engineering from the Hasso Plattner Institute. Contact him at frank.feinbube@hpi.uni-potsdam.de.

PETER TRÖGER is a senior researcher at the Hasso Plattner Institute. His research focuses on dependable multicore systems, especially on fault prediction and injection. Tröger has a PhD in computer science from the University of Potsdam. Contact him at peter.troeger@hpi.uni-potsdam.de.

ANDREAS POLZE leads the Hasso Plattner Institute's Operating Systems and Middleware group. His research focuses on operating system architectures, component-based middleware, and predictable distributed and cloud computing. Polze has a PhD from Freie University and habilitation (tenure) from Humboldt University, both in Berlin. Contact him at andreas.polze@hpi.uni-potsdam.de.

References

1. K. Asanovic et al., *The Landscape of Parallel Computing Research: A View from Berkeley*, tech. report UCB/EECS-2006-183, Univ. of California, Berkeley, Dec. 2006.
2. *The OpenCL Specification–Version 1.1*, Khronos OpenCL Working Group, Sept. 2010; www.khronos.org/registry/cl.
3. V.W. Lee, "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," *Proc. 37th Ann. Int'l Symp. Computer Architecture* (ISCA 10), ACM Press, 2010, pp. 451–460.
4. S.-Z. Ueng et al., "CUDA-Lite: Reducing GPU Programming Complexity," *Proc. 21th Int'l Workshop Languages and Compilers for Parallel Computing* (LCPC 08), LNCS 5335, Springer, 2008, pp. 1–15.
5. Advanced Micro Devices, *ATI Stream Computing OpenCL Programming Guide*, June 2010; http://developer.amd.com/zones/OpenCLZone/programming/Pages/default.aspx.
6. Nvidia, *Nvidia OpenCL Best Practices Guide*, Version 2.3, Aug. 2009; www.nvidia.com/object/cuda_opencl_new.html.