Data-Parallel Computing

CHAS. BOYD, MICROSOFT

Data parallelism is a key concept in leveraging the power of today's manycore GPUs.

U

sers always care about performance.

Although often it's just a matter of making sure the software is doing only what it should, there are many cases where it is vital to get down to the metal and leverage the fundamental characteristics of the processor.

Until recently, performance improvement was not difficult. Processors just kept getting faster. Waiting a year for the customer's hardware to be upgraded was a valid optimization strategy. Nowadays, however, individual processors don't get much faster; systems just get more of them.

Much comment has been made on coding paradigms to target multiple-processor cores, but the data-parallel paradigm is a newer approach that may just turn out to be easier to code to, and easier for processor manufacturers to implement.

This article provides a high-level description of data-parallel computing and some practical information on how and where to use it. It also covers data-parallel programming environments, paying particular attention to those based on programmable graphics processors.

A BIT OF BACKGROUND

Although the rate of processor-performance growth seems almost magical, it is gated by fundamental laws of physics. For the entire decade of the '90s, these laws enabled processors to grow exponentially in performance as a result of improvements in gates-per-die, clock speed, and instruction-level parallelism. Beginning in 2003, though, the laws of physics (power and heat) put an end to growth in clock speed. Then the silicon area requirements

Data-Parallel Computing

for increasingly sophisticated ILP (instruction-level parallelism) schemes (branch prediction, speculative execution, etc.) became prohibitive. Today the only remaining basis for performance improvement is gate count.

Recognizing this, manufacturers have restructured to stop pushing clock rate and focus on gate count. Forecasts project that gates-per-die can double every two years for the next six to eight years at least. What do you do with all those gates? You make more cores. The number of cores per die will therefore double every two years, resulting in four times today's core counts (up to 32 cores) by 2012.

Customers will appreciate that growth rate, but they will benefit only if software becomes capable of scaling across all those new cores. This is the challenge that performance software faces in the next five to ten years. For the next decade, the limiting factor in software performance will be the ability of software developers to restructure code to scale at a rate that keeps up with the rate of core-count growth.

PARALLEL PROGRAMMING

Parallel programming is difficult. We deprecate the use of GOTO statements in most languages, but parallel execution is like having them randomly sprinkled throughout the code during execution. The assumptions about order of execution that programmers have made since their early education no longer apply.

The single-threaded von Neumann model is comprehensible because it is deterministic. Parallel code is subject to errors such as deadlock and livelock, race conditions, etc. that can be extremely subtle and difficult to identify, often because the bug is nonrepeatable. These issues are so severe that despite decades of effort and dozens of different approaches, none has really gained significant adoption or even agreement that it is the best solution to the problem.

An equally subtle challenge is performance scaling. Amdahl's law states that the maximum speedup attainable by parallelism is the reciprocal of the proportion of code that is not parallelizable. If 10 percent of a given code base is not parallel, even on an infinite number of processors it cannot attain more than a tenfold speedup.

Although this is a useful guideline, determining how much of the code ends up running in parallel fashion is very difficult. Serialization can arise unexpectedly as a result of contention for a shared resource or requirements to access too many distant memory locations.

The traditional methods of parallel programming (thread control via locks, message-passing interface, etc.) often have limited scaling ability because these mechanisms can require serialization phases that actually increase with core count. If each core has to synchronize with a single core, that produces a linear growth in serial code, but if each core has to synchronize with all other cores, there can be a combinatoric increase in serialization.

After all, any code that serializes is four times slower on a four-core machine, but 40 times slower on a 40-core machine.

Another issue with performance scaling is more fundamental. A common approach in multicore parallel programming for games is to start with a top-down breakdown. Relatively isolated subsystems are assigned to separate cores, but what happens once the number of subsystems in the code base is reached? Since restructuring code at this level can be pervasive, it often requires a major rewrite to break out subsystems at the next finer level, and again for each hardware generation.

For all these reasons, transitioning a major code base to parallel paradigms is time consuming. Getting all the subtle effects of nondeterminism down to an acceptable level can take years. It is likely that by that time, core-count growth will have already exceeded the level of parallelism that the new code structure can scale to. Unfortunately, the rate of core-count growth may be outstripping our ability to adapt to it.

Thus, the time has come to look for a new paradigm—ideally one that scales with core count but without requiring restructuring of the application architecture every time a new core count is targeted. After all, it's not about choosing a paradigm that operates well at a fixed core count; it's about choosing one that continues to scale with an increasing number of cores without requiring code changes. We need to identify a finer level of granularity for parallelism.

DATA-PARALLEL PROGRAMMING

Given the difficulty of finding enough subsystem tasks to assign to dozens of cores—the only elements of which there are a comparable number are data elements—the data-parallel approach is simply to assign an individual data element to a separate logical core for processing. Instead of breaking code down by subsystems, we look for fine-grained inner loops within each subsystem and parallelize those.

For some tasks, there may be thousands to millions of data elements, enabling assignment to thousands of cores. (Although this may turn out to be a limitation in the future, it should enable code to scale for another decade or so.) For example, a modern GPU can support hundreds of ALUs (arithmetic logic units) with hundreds of threads per ALU for nearly 10,000 data elements on the die at once.

The history of data-parallel processors began with the efforts to create wider and wider vector machines. Much of the early work on both hardware and data-parallel algorithms was pioneered at companies such as MasPar, Tera, and Cray.

Today, a variety of fine-grained or data-parallel programming environments are available. Many of these have achieved recent visibility by supporting GPUs. They can be categorized as follows:

Older languages (C*, MPL, Co-Array Fortran, Cilk, etc.). Several languages have been developed for fine-grained parallel programming and vector processing. Many add only a very small difference in syntax from well-known languages. Few of them support a variety of platforms and they may not be available commercially or be supported long term as far as updates, documentation, and materials.

Newer languages (XMT-C, CUDA, CAL, etc.). These languages are being developed by the hardware company involved and therefore are well supported. They are also very close to current C++ programming models syntactically; however, this can cause problems because the language then provides no explicit representation of the unique aspects of data-parallel programming or the processor hardware. Although this can reduce the changes required for an initial port, the resulting code hides the parallel behavior, making it harder to comprehend, debug, and optimize. Simplifying the initial port of serial code through syntax is not that useful to begin with, since for best performance it is often an entire algorithm that must be replaced with a data-parallel version. Further, in the interest of simplicity, these APIs may not

expose the full features of the graphics-specific silicon, which implies an underutilized silicon area.

Array-based languages (RapidMind, Acceleware, Microsoft Accelerator, Ct, etc.). These languages are based on array data types and specific intrinsics that operate on them. Algorithms converted to these languages often result in code that is shorter, clearer, and very likely faster than before. The challenge of restructuring design concepts into array paradigms, however, remains a barrier to adoption of these languages because of the high level of abstraction at which it must be done.

Graphics APIs (OpenGL, Direct3D). Recent research in GPGPU (general-purpose computing on graphics processing units) has found that while the initial ramp-up of using graphics APIs can be difficult, they do provide a direct mapping to hardware that enables very specific optimizations, as well as access to hardware features that other approaches may not allow. For example, work by Naga Govindaraju¹ and Jens Krüger² relies on access to fixed-function triangle interpolators and blending units that the newer languages mentioned here often do not expose. Further, there is good commercial support and a large and experienced community of developers already using them.

GPUS AS DATA-PARALLEL MACHINES

The GPU is the second-most-heavily used processor in a typical PC. It has evolved rapidly over the past decade to reach performance levels that can exceed the CPU by

Data-Parallel Computing

a large factor, at least on appropriate workloads.³ GPU evolution has been driven by 3D rendering, an embarrassingly data-parallel problem, which makes the GPU an excellent target for data-parallel code. As a result of this significantly different workload design point (processing model, I/O patterns, and locality of reference), the GPU has a substantially different processor architecture and memory subsystem design, typically featuring a broader SIMD (single instruction, multiple data) width and a higher-latency, higher-bandwidth streaming memory system. The processing model exposed via a graphics API is a task-serial pipeline made up of a few data-parallel stages that use no interthread communication mechanisms at all. While separate stages appear for processing vertices or pixels, the actual architecture is somewhat simpler.

As shown in figure 1, a modern DirectX10-class GPU has a single array of processors that perform the computational work of each stage in conjunction with specialized

hardware. After polygon-vertex processing, a specialized hardware interpolator unit is used to turn each polygon into pixels for the pixel-processing stage. This unit can be thought of as an address generator. At the end of the pipeline, another specialized unit blends completed pixels into the image buffer. This hardware is often useful in accumulating results into a destination array. Further, all processing stages have access to a dedicated texture-sampling unit that performs linearly interpolated reads on 1D, 2D, or 3D source arrays in a variety of data-element formats.

Shaped by these special workload requirements, the modern GPU has:

- Ten times the GFLOPS of CPU chips for similar price and power consumption
- Thousands of threads distributed over hundreds of single-precision floating-point ALUs
- A dedicated streaming-memory system with 10 times the memory bandwidth of a CPU
- Dedicated memory capacity similar to the CPU system memory capacity
- Specialized cores for filtering, blending, rasterizing, and video processing

A GPU's memory subsystem is designed for higher I/O latency to achieve increased throughput. It assumes only very limited data reuse (locality in read/write access), featuring small input and output caches designed more as FIFO (first in, first out) buffers than as mechanisms to avoid round-trips to memory.

Recent research has looked into applying these processors to other algorithms beyond 3D rendering. There have been applications that have shown significant benefits over CPU code. In general, those that most closely match the original design workload of 3D graphics (such as image processing) and can find a way to leverage either the tenfold compute advantage or the tenfold bandwidth advantage have done well. (Much of this work is cataloged on the Web at <http://www.gpgpu.org>.)

This research has identified interesting algorithms. For example, compacting an array of variable-length records is a task that has a data-parallel implementation on the parallel prefix sum or scan. The prefix-sum algorithm computes the sum of all previous array elements (i.e.,

the first output element in a row r is r_0 , while the second is $o_1 = r_0 + r_1$, and the n th output element is $o_n = r_0 + r_1 + \dots + r_n$. Using this, a list of record sizes can be accumulated to compute the absolute addresses where each record element is to be written. Then the writes can occur completely in parallel. Note that if the writes are done in order, the memory-access pattern is still completely sequential.⁴

MAKING CODE DATA-PARALLEL

Before starting to write your code, check for tasks that are known data-parallel cases. Often you can find library routines already available for accelerating common tasks using data-parallel hardware. Most data-parallel programming environments include such libraries as a convenient way for users to begin adopting their technology.

If you need to write custom data-parallel code, the process is similar to a localized optimization effort. You can adopt data-parallel programming incrementally, since you can identify and optimize the key inner loops one at a time, without perturbing the larger-scale structure of the code base. Here are the basic steps for converting code to the data-parallel model:

1. Identify a key task that looks data-parallel.
2. Identify a data-parallel algorithm for this task.
3. Select a data-parallel programming environment.
4. Implement code.
5. Evaluate performance scaling rate.
6. Go to step 1.

STEP 1: IDENTIFY A KEY TASK THAT LOOKS DATA-PARALLEL

Look for a segment of code that doesn't rely greatly on cross communication between data elements, or conversely, a set of data elements that can be processed without requiring too much knowledge of each other. Look for data-access patterns that can be regularized, as opposed to arbitrary/random (such as linear arrays versus sparse-tree data structures).

While searching for candidates to parallelize, you can evaluate performance potential via Amdahl's law: just comment out this candidate task (simulate infinite parallelism) and check to see total performance change. If there isn't a significant improvement, going through the effort of parallelizing won't pay off.

STEP 2: IDENTIFY A DATA-PARALLEL ALGORITHM FOR THIS TASK

Often a good place to look is in the history books (math)

or in routines developed by Tera/Cray for its vector processors. For example, bitonic sorts were identified as interesting before computers were developed, but fell out of favor during the rise of current cache-based machines. Other examples are radix sorts, and prefix sum (scan) operations used for packing sparse data.

STEP 3: SELECT A DATA-PARALLEL PROGRAMMING ENVIRONMENT

Many data-parallel programming environments are available today. Many of the criteria to use in evaluation are the same as for any development environment. The areas to look for are:

- **Abstraction level.** Do you need a library, a set of data-abstraction utilities, or a language?
- **Syntax clarity.** Are limitations of the implementation explicit in the syntax or hidden by it?
- **Maintainability.** Would the resulting code complexity be manageable?
- **Support.** Are there user groups or support services?
- **Availability.** How broadly distributed is the environment or any hardware that it requires?
- **Compatibility.** Is the environment compatible with a broad range of systems or only a specific subset?
- **Lifespan.** Is the environment compatible with future hardware, even from the same vendor?
- **Documentation.** Do the docs make sense? Are the samples useful?
- **Cost.** How much will it cost users of your product to get any required hardware or software?

STEP 4: IMPLEMENT CODE

Code it up, at least at the pseudocode level. If implementation turns out to require more than one or two places where interthread communication is required, then this may not be a sufficiently data-parallel algorithm. In that case, it may be necessary to look for another algorithm (step 2) or another task to parallelize (step 1).

STEP 5: EVALUATE PERFORMANCE SCALING

Performance at a given core count is interesting but not the key point. (If you are going to check that, be sure to compare using a realistic "before" case.) A more important metric to check is how the new code scales with increasing core count. If there is no sign of a performance plateau, the system will have some scaling headroom. After all, absolute performance relative to a single core is not as relevant as how it scales with core-count growth over time.

Data-Parallel Computing

In summary:

- **Understand the paradigm.** What are data-parallel computation and the streaming-memory model?
- **Understand your code.** Which portions operate at which level of granularity?
- **Understand the environment.** How does it help solve the problem?

GPU PERFORMANCE HINTS

If targeting a GPU, are there operations that can leverage the existing graphics-related hardware? Are your data types small enough? GPUs are designed to operate on small data elements so media data (image/video pixels or audio samples) is a good fit. Or when sorting on the GPU, working with key-index pairs separately is often a win. Then the actual movement of data records can be done on the CPU, or on the GPU as a separate pass.

GPUs are optimized for work with 1D, 2D, or 3D arrays of similar data elements. Array operations are often faster using GPU hardware because it can transparently optimize them for spatially coherent access.

When reading such arrays, the GPU can easily linearly interpolate regular array data. This effectively enables a floating-point (fuzzy) array index. Many mathematical algorithms use either a simple linear interpolation of array elements or slightly higher-order schemes that can be implemented as a few linear interpolations. GPU hardware has a significant proportion of silicon allocated to optimizing the performance of these operations.

Algorithms that involve an accumulation or summation of values into a set of results (instead of just a write/copy) can leverage yet another large chunk of special silicon on GPUs: the blender is designed for efficiently compositing or accumulating values into an array. Some matrix math algorithms and reduction operations have shown benefits here.

REGISTER PRESSURE

Some architectures (such as GPUs) are flexible in that they can assign variable numbers of threads to a core based on how many registers each thread uses. This enables more threads to be used when fewer temporary registers are needed, but reduces the threads available (and the paral-

lelism) for algorithms that need more registers. The key is to break tasks into simpler steps that can be executed across even more parallel threads. This is the essence of data-parallel programming.

For example, a standard 8x8 image DCT (discrete cosine transform) algorithm operates on transposed data for its second half. The transpose can take dozens of registers to execute in place, but breaking it into two passes so that the transpose happens in the intervening I/O results in only a handful of registers needed for each half. This approach improved performance from far slower than a CPU to three times that of a highly optimized SSE assembly routine.

HINTS FOR REDUCTIONS

Reductions are common operations: find the total, average, min, max, or histogram of a set of data. The computations are easily data-parallel, but the output write is an example of cross-thread communication that must be managed carefully.

Initial implementations allocated a single shared location for all the threads to write into, but execution was completely serialized by write contention to that location. Allocating multiple copies of the reduction destination and then reducing these down in a separate step was found to be much faster. The key is to allocate enough intermediate locations to cover the number of cores (hundreds) and, therefore, performance level that you want to scale to.

PROGRAMMING THE MEMORY SUBSYSTEM

The data-parallel paradigm extends to the memory subsystem as well. A full data-parallel machine is able not only to process individual data elements separately, but also to read and write those elements in parallel. This characteristic of the memory subsystem is as important to performance as the execution model. For example, I/O ports are a shared resource, and performance is improved if multiple threads are not contending for the same one.

Data structures manipulated imply memory-access patterns. We have seen cases where switching from pointer-based data structures such as linked lists or sparse trees to data-parallel-friendly ones (regular arrays, grids, packed

streams, etc.) allows code to become compute-bound instead of memory-bound (which can be as much as 10 times faster on GPUs). This is because memory is typically organized into pages, and there is some overhead in switching between pages. Grouping data elements and threads so that many results can be read from (or written to) the same page helps with performance.

Many types of trees and other sparse-data structures have data-parallel-friendly array-based implementations. Although using these structures is quite conventional, their implementations are nonintuitive to developers trained on pointer-based schemes.⁵

The most important characteristic of the GPU memory subsystem is the cache architecture. Unlike a CPU, the GPU has hardly any read/write cache. It is assumed that so much data will be streaming through the processor that it will overflow just about any cache. As a result, the only caches present are separate read-through and write-through buffers that smooth out the data flow. Therefore, it is critical to select algorithms that do not rely on reuse of data at scales larger than the few local registers available. For example, histogram computation requires more read/write storage to contain the histogram bins than typical register allocation supports. Upcoming GPU architectures are beginning to add read/write caches so that more algorithms will work, including reasonably sized histograms, but since these caches are still 10 to 100 times smaller than those on the CPU, this will remain a key criterion when choosing an algorithm.

GPUS AS DATA-PARALLEL HARDWARE

GPU systems are cheap and widely available, and many programmers (such as game developers) have identified key approaches to programming them efficiently.

First, it can be important to leverage all the silicon on the die. Applications that don't light up the graphics-specific gates are already at a disadvantage compared with a CPU. For example, Govindaraju's sort implementations show significant benefits from using the blending hardware.⁶

Another way to ensure programming efficiency is to keep the data elements small. This extra hardware is assuming graphics data types that are optimal when they are 16 or fewer bytes in size, and ideally four bytes. If you can make your data look like what a GPU usually processes, you will get large benefits.

Unfortunately, the GPU's high-speed memory system (10 times faster throughput than the CPU front side bus) is typically connected to the CPU by a link that is 10

times slower than CPU memory. Minimizing data and control traffic through this link is vital to GPU performance in low-latency scenarios. The secret is to keep data in the GPU's memory as long as possible, bringing it back to the CPU only for persistent storage. Sometimes this may involve executing a small non-data-parallel task on the GPU because the cost of sending the required data across to the CPU, synchronizing it, and sending it back may be even greater.

GPU GENERALITY

With shorter design cycles, GPUs have been evolving more rapidly than CPUs. This evolution has typically been in the direction of increased generality. Now we are seeing GPU generality growing beyond the needs of basic rendering to more general applications. For example, in the past year new GPU environments have become available that expose features that the graphics APIs do not. Some now support sharing of data among threads and more flexible memory-access options.

This enables entirely new classes of algorithms on GPUs. Most obviously, more general approaches to 3D processing are becoming feasible, including manipulation of acceleration data structures for ray tracing, radiosity, or collision detection. Other obvious applications are in media processing (photo, video, and audio data) where the data types are similar to those of 3D rendering. Other

Fundamental Algorithms Sorted by Granularity of Parallelism

granularity	coarse	serial entropic encoding sparse structure editing sparse linear algebra sparse structure creation sparse structure query dense linear algebra video motion estimation bitonic sort histogram generation reduce explicit finite difference image convolution pixel lighting map
	fine	

FIG 2

Data-Parallel Computing

domains using similar data types are seismic and medical analysis.

FUTURE HARDWARE EVOLUTION: CPU/GPU CONVERGENCE?

Processor features such as instruction formats will likely converge as a result of pressure for a consistent programming model. GPUs may migrate to narrower SIMD widths to increase performance on branching code, while CPUs move to broader SIMD width to improve instruction efficiency.

The fact remains, however, that some tasks can be executed more efficiently using data-parallel algorithms. Since efficiency is so critical in this era of constrained power consumption, a two-point design that enables the optimal mapping of tasks to each processor model may persist for some time to come.

Further, if the hardware continues to lead the software, it is likely that systems will have more cores than the application can deal with at a given point in time, so providing a choice of processor types increases the chance of more of them being used.

Conceivably, a data-parallel system could support the entire feature set of a modern serial CPU core, including a rich set of interthread communications and synchronization mechanisms. The presence of such features, however, may not matter in the longer term because the more such traditional synchronization features are used, the worse performance will scale to high core counts. The fastest apps are not those that port their existing single-threaded or even dual-threaded code across, but those that switch to a different parallel algorithm that scales better because it relies less on general synchronization capabilities.

Figure 2 shows a list of algorithms that have been implemented using data-parallel paradigms with varying degrees of success. They are sorted roughly in order of how well they match the data-parallel model.

Data-parallel processors are becoming more broadly available, especially now that consumer GPUs support data-parallel programming environments. This paradigm shift presents a new opportunity for programmers who adapt in time.

The data-parallel industry is evolving without much

guidance from software developers. The first to arrive will have the best chance to drive and shape upcoming data-parallel hardware architectures and development environments to meet the needs of their particular application space.

When programmed effectively, GPUs can be faster than current PC CPUs. The time has come to take advantage of this new processor type by making sure each task in your code base is assigned to the processor and memory model that is optimal for that task. Q

REFERENCES

1. Govindaraju, N.K., Gray, J., Kumar, R., Manocha, D. 2006. GPU TeraSort: High-performance graphics coprocessor sorting for large database management. *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*; http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-2005-183.
2. Krüger, J., Westermann, R. 2003. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics* 22(3).
3. Blythe, D. 2008. The Rise of the GPU. *Proceedings of the IEEE* 96(5).
4. Shubhabrata, S., Lefohn, A.E., Owens, J.D. 2006. A work-efficient step-efficient prefix sum algorithm. *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures*: D-26-27.
5. Lefohn, A.E., Kniss, J., Strzodka, R., Sengupta, S., Owens, J.D. 2006. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics* 25(1).
6. See reference 1.

SUGGESTED FURTHER READING

GPU Gems 2:

http://developer.nvidia.com/object/gpu_gems_2_home.html

GPU Gems 3:

<http://developer.nvidia.com/object/gpu-gems-3.html> Ch 39 on prefix sum

Glift data structures:

<http://graphics.cs.ucdavis.edu/~lefohn/work/glift/>

Rapidmind:

<http://www.rapidmind.net/index.php>

Intel Ct:

http://www.intel.com/research/platform/terascale/TeraScale_whitepaper.pdf

Microsoft DirectX SDK:

<http://msdn2.microsoft.com/en-us/library/aa139763.aspx>

Direct3D HLSL:

<http://msdn2.microsoft.com/en-us/library/bb509561.aspx>

Nvidia CUDA SDK:

<http://developer.nvidia.com/object/cuda.html>

AMD Firestream SDK:

<http://ati.amd.com/technology/streamcomputing/stream-computing.pdf>

Microsoft Research's Accelerator:

<http://research.microsoft.com/research/pubs/view.aspx?type=technical%20report&id=1040&sr=a>

<http://research.microsoft.com/research/downloads/Details/25e1bea3-142e-4694-bde5-f0d44f9d8709/Details.aspx>

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

CHAS. BOYD is a software architect at Microsoft. He joined the Direct3D team in 1995 and has contributed to releases since DirectX 3. During that time he has worked closely with hardware and software developers to drive the adoption of features such as programmable hardware shaders and float pixel processing into consumer graphics. Recently he has been investigating new processing architectures and applications for mass-market consumer systems.

© 2008 ACM 1542-7730/08/0300 \$5.00

acm
queue
architecting tomorrow's computing

Object-Relational Mappers

The End of Transactions
ORM in Dynamic Languages
LINQ and Entity Framework

Coming Soon in Queue