TIM KALDEWEY

programming video cards for database applications

Tim Kaldewey is a Researcher in the Special Projects Team at Oracle Corporation and a Ph.D. candidate at UCSC's Computer Systems Research Group, headed by Professor Scott Brandt. He previously held positions at IBM, SAP, Lufthansa, and SAG. His research focuses on predictable high-performance data management and, in particular, on parallel architectures.

tim.kaldewey@oracle.com

TERAFLOPS AND OVER 100 GB/SEC memory bandwidth do not only realize gamer dreams of "better"-looking monsters, they also attract developers of other performance-hungry applications. While the hardware specifications of high-end graphics processors (GPUs) with hundreds of cores make multicore CPUs look like toys, the complexity of leveraging these exotic hardware platforms for general-purpose applications puts a high price tag on application development. Even though new development environments allow C-style programming, efficient implementations still require extensive knowledge of computer architecture as well as analytical and debugging skills, going beyond standard tools. In this article, I would like to share my experiences in programming video cards for database operations over the last three years.

In the past, using video cards for non-graphics applications has been considered one of the "black arts" of computer programming, practiced only by a handful of hackers and researchers. The programmer needed to fool the GPU into thinking it would draw a scene to display, while it was in fact performing a general-purpose computation. This required mapping data to graphics objects described by floating-point vectors and ensuring that results were "drawn" within visible screen space; otherwise they were no longer accessible or not even computed. CPU results did not necessarily match GPU results, since for the dominant application, that is, games, speed was more important than accuracy, and most video cards did not implement 32-bit floating-point precision. Despite the difficulties, the impressive performance of early prototypes started a wave of general-purpose GPU applications [7].

Until two years ago, implementing general-purpose applications required using Graphics APIs such as OpenGL and Cg. In early 2007, our first prototype implementation of parallel search used the color information of each pixel in an image to store data. Using a 24" screen with a resolution of 1920×1200 pixels, the biggest possible data set that it could handle with this method was 9.2 million characters or 8.8MB. However, the physical size was four times as much, since each 8-bit character had to be stored as a 32-bit floating-point value.

New software development environments such as NVIDIA's Compute Unified Device Architecture (CUDA) greatly simplify programming GPUs for non-graphics tasks [10]. It is no longer necessary to use graphics data types and drawing primitives or to limit data-set sizes to the maximum screen resolution. Besides a few additional instructions and function type qualifiers, which determine the degree of parallelism and where a piece of code is executed (GPU or CPU), CUDA allows standard C-style programming.

The programming obstacles removed, commercial software developers started evaluating GPUs for computationally intense tasks, as an alternative to clusters. With data centers reaching their physical limitations in terms of space, power consumption, and cooling, alternative solutions with higher computational performance per watt and per square foot become very appealing. Using GPUs with more than one teraflop of compute performance each, a 100 teraflop data center could be realized with less than 100 GPUs [2]. To achieve the same with conventional PC/server hardware would require more than 1400 CPUs, 70 gigaflops each. Assuming a power consumption of roughly 200 watts per GPU and 130 watts per CPU, a GPU solution would require only a tenth of the power required by CPUs. Including the power consumption of other components required for each machine will favor a GPU solution even more, since it requires only 25 machines with four video cards each.

Besides teraflops, the latest GPUs feature up to 4GB of memory and memory throughput beyond 100GB/sec, which makes them attractive for data-intensive applications, e.g., databases. Over the past few years, the growth rates of main memory size have outstripped the growth rates of structured data in the enterprise, particularly when ignoring historical data. Gartner predicts that in-memory analytics will soon become feasible even for large data-warehousing applications [11]. Databases also offer plenty of opportunity for parallel execution, as they usually handle many queries simultaneously.

However, GPU hardware development continues to be driven by the mass market for games and multimedia, and implementing general-purpose applications, which do not necessarily resemble graphics applications, remains a challenge. The non-uniform memory architecture between CPU and GPU requires explicit data copies and address translation, and the PCI-express bus turns out to be a bottleneck, making it difficult to leverage the GPU for data-intensive applications. Overall, the subset of applications that can potentially benefit from using the GPU as a co-processor has to be parallelizable and complex enough to not be dominated by data transfers between main and video memory.

After a brief introduction of the GPU architecture, I will use search as an example to describe the hoops to jump through in order to achieve good performance on GPU applications. Whether performance gains of GPU implementations justify excessive development efforts has to be answered for the individual application. On the other hand, the trend towards increasingly parallel architectures requires rethinking traditional serial applications all the way down to the algorithmic level, and exploring alternative parallel architectures provides opportunities to get a head start.

GPGPU

When computers were mostly used for scientific applications and accounting, there was no need to develop a processing unit devoted to graphics.

With the evolution of hardware, software, and users' taste, many applications—especially computer games—started using graphical output. At the beginning of the computer graphics era, the CPU was in charge of all graphics operations. Mainly driven by the growing demand for more realistic computer games, more and more complex operations were offloaded to the GPU. A standard *graphics pipeline* would perform a fixed geometrical transformation on graphics data, vertices of triangles, followed by coloring.

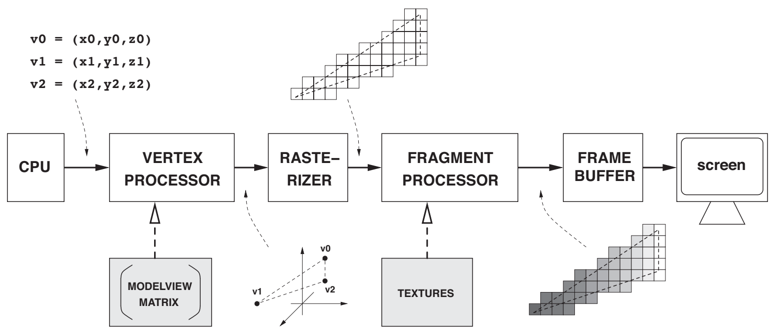

FIGURE 1: THE GRAPHICS PIPELINE

GPUs kept evolving in two directions. First, memory sizes increased, significantly more than required for the *frame buffer*, the part of memory which maps directly to the screen. Second, the programmable graphics pipeline model (Figure 1) became the sequence of a *vertex processor*, to perform geometric transformations of vertices in 3D space; a *rasterizer*, to transform geometric primitives (such as lines or triangles) into actual pixels based on the screen resolution; and a *fragment* processor, to color the pixels. While the rasterizer's function has been fixed, the vertex processor and the fragment processor are now effectively programmable. Moreover, while the initial graphics pipeline would simply *stream* the data through once, the programmable pipeline can access the larger memory in a more flexible way, storing multiple images (textures) and intermediate-rendering passes of complex computations. Modern GPUs comprise multiple vertex and fragment processors executing the same programs on different primitives in parallel. When, thanks to their massively parallel architecture, GPUs started becoming more powerful than CPUs, some programmers began exploring them for non-graphics computations, leading to the birth of the *General-Purpose Graphics Processing Unit* (GPGPU).

However, programming the GPU was not simple, given the rigid limitations in functionality, data types, and memory access. Both the hardware and the software support were geared exclusively toward graphics computation. Graphics APIs like OpenGL and Cg required mapping variables to graphics objects such as textures, and algorithms to geometric and color transformations. Textures are two-dimensional arrays of four-wide single-precision floating-point vectors storing color information for each pixel in terms of red, green, blue, and opacity (rgba). Vertices are stored as four-wide floating-point vectors for x-,y-,z-coordinates and w for normalizing coordinates.

While the vertex processor allowed writing results to any coordinate, i.e., memory scatter, it could not read data from multiple locations, i.e., memory gather, limiting the input for computation to an individual data point. On the other hand, the fragment processor could gather data from up to eight different textures but did not support scatter, thus could only write results to a single fixed memory location, determined by the current pixel position.

Using graphics APIs, the following steps were necessary to invoke GPU computation: One had to organize the data into a two-dimensional array.

This array was mapped to the physical screen as one pixel per element, referred to as *screen-sized viewport*. Then one had to load a fragment program that was to be executed on each data element or pixel and, finally, pretend to "draw" the screen-sized image to actually run the code on each pixel. If the results were graphical in nature, one could just leave them displayed on the screen, but in the general case, one would copy the results (i.e., content) from the frame buffer on which the "image" was rendered back to another texture or main memory.

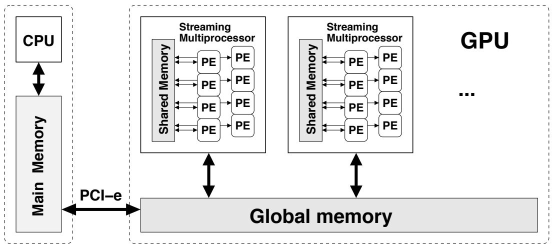

FIGURE 2: ARCHITECTURE OF AN NVIDIA GEFORCE 8 SERIES GPU

NVIDIA's CUDA, allowing people to program the GPU directly, was a major leap ahead [10]. Instead of dedicated hardware for each stage, CUDA-capable GPUs are based on flexible programmable processors, capable of any of the steps performed by a conventional graphics pipeline. At the top level, a CUDA application consists of two parts: a serial program running on the CPU, and a parallel part, called a *kernel*, running on the GPU.

The kernel is organized as a number of *blocks* of *threads*, with one block running all of its threads to completion on one of the several *streaming multiprocessors* (SMs). When the number of blocks as defined by the programmer exceeds the number of physical multiprocessors, blocks are queued automatically. Each SM has eight processing elements, PEs (Figure 2), which execute the same instruction at the same time in *Single Instruction-Multiple Data* (SIMD) mode [5].

To optimize SM utilization, the GPU groups threads within a block following the same code path into so-called *warps* for SIMD-parallel execution. Due to this mechanism, NVIDIA calls its GPU architecture *Single Instruction Multiple Threads* (SIMT). Threads running on the same SM share a set of registers as well as a low-latency *shared memory* located on the processor chip. This shared memory is small (16KB on the G80) but about 100x faster than the larger *global* memory on the GPU board. A careful memory access strategy is even more important on the GPU than it is on the CPU, because caching on the GPU is minimal and mainly the programmer's responsibility.

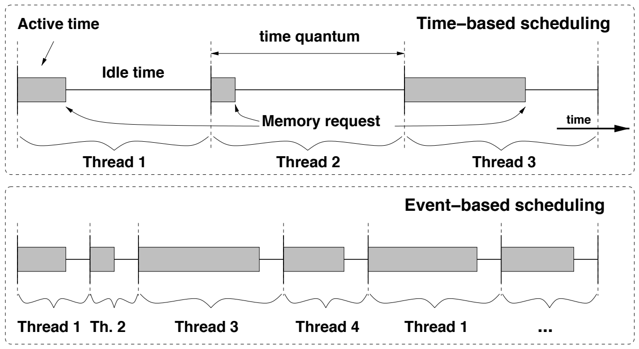

FIGURE 3: COMPARISON OF SCHEDULING TECHNIQUES. *EVENT-BASED SCHEDULING* ON THE GPU MAXIMIZES PROCESSOR UTILIZATION BY SUSPENDING THREADS AS SOON AS THEY ISSUE A MEMORY REQUEST, WITHOUT WAITING FOR A *TIME QUANTUM* TO EXPIRE, AS ON THE CPU.

To compensate for the lack of caching, GPUs employ massive multi-threading to effectively hide memory latency. The scheduler within an SM decides for each cycle which group of threads (warp) to run, such that warps with threads accessing memory can be suspended at no cost until the requested data is available. The seamless multi-threading is made possible by thousands of registers in each SM; each thread keeps its variables in registers and context switching is free. Effectively, this approach implements what I would naively describe as event-based scheduling (Figure 3) and benefits large, latency-bound workloads.

On the other hand, CPUs employ large caches but rely on a single set of registers, requiring context switches to preserve the state of execution of the current thread before loading the next. As context-switching is expensive and schedulers are implemented in software, CPU scheduling is based on time quanta; in case of a cache miss a thread sits idle until the memory request returns or its time quantum expires.

These characteristics make the GPU an interesting platform to explore for parallel database processing.

Programming GPUs

Before taking the plunge into video card programming using search operations, I would like to briefly discuss the corresponding CPU implementation, to show the differences.

Implementing search on text indexes—in the simplest case, sorted lists—may not require much more than "stitching" together a few standard library calls to produce the desired results. For example, searching for all documents containing the word "Flughafenbahnhof" (the lengthy German word for a train station at the airport) in the ideal case requires only a few lines of code.

```
char searchkey[16]= "Flughafenbahnhof";
result = bsearch( (void*)&searchkey,index, numentries,
                  sizeof(char)*maxwordlength,
                  (int(*)(const void*,const void*)) strcmp);
```

Although standard library functions like string comparison and binary search have been around for decades and are highly optimized, on modern multi-core CPUs there is still plenty of room for optimizations.

Large-scale database servers may handle more than thousands of queries per second, of which many can be served simultaneously by using multiple threads. A multi-core CPU can execute up to `#cores` of those queries in parallel. Even for memory-bound operations like index search it is imperative to employ multi-threading, as a single core cannot achieve maximum memory performance [8]. Since search by itself involves no data manipulations, a multi-threaded implementation is straightforward and does not require special caution.

German, which happens to contain many long words, is not the only language for which comparing strings character by character seems suboptimal in terms of memory performance. In fact, performance of byte-wise vs. multi-word (vector) memory accesses can differ by more than an order of magnitude [8]. On x86 CPUs we can leverage the SSE vector unit to load 16 bytes with a single instruction, but in turn it requires implementing a vector string comparison. On earlier processor generations this involved considerable assembly programming, whereas the recently released Core i7 implements specific instructions for string comparisons.

Our first prototype implementation of parallel search in early 2007 used the OpenGL and Cg graphics APIs, which required mapping string data to two-dimensional textures and writing vertex and fragment programs. Although the basic approach presented here does not yield competitive performance, it illustrates the effort necessary to leverage GPGPU during its early stages. As a comprehensive description of all necessary steps to invoke GPU computation would go beyond the scope of this article, I will only highlight critical ones. For obvious reasons, the following examples require a 1:1 pixel-to-texture element (texel) ratio, unless you prefer Scrabble results.

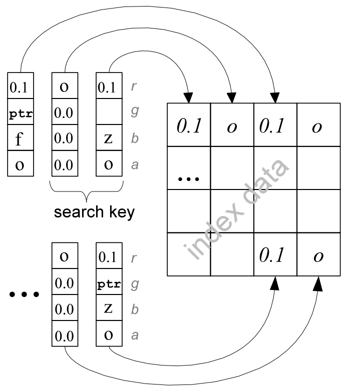

FIGURE 4: MAPPING INDEX DATA TO A TEXTURE

A simple way to store character data in a texture is to map the ASCII character set to floating-point values between 0.0 and 255.0 and use the rgba color information of each pixel to store up to four characters (Figure 4). Strings are null-terminated (0.0), and their starting point is marked as well (0.1). The marking is necessary to make sure we do not report partial string matches, since parallelism is transparent, meaning pixels are processed independently. Dependent on string length, this approach might result in numerous idle processors due to the GPU's SIMD operation (see "GPGPU" section, above).

```
float* data = malloc(sizeof(float)*1200*1200*4);
...
data[pos++] = 0.1;
data[pos++] = *(float*)&docindex;
for (i=0;i<=strlen(currentString);i++) {
      data[pos++] = (float)currentString[i];
}
...
glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB,
                         0,0,0, // detail level, x-, y- offset
                         1200, 1200, // size
                         GL_RGBA, // texture format
                         GL_FLOAT, // data format
                         data); // data pointer
```

Although this encoding requires four bytes per character and an additional four bytes for marking the beginning of a string, it greatly simplifies the identification of an individual string and implementation of string comparison. Floating-point numbers can be directly compared using "=", and string boundaries are aligned with the colors of a texel or pixel. Given the small range of numbers and that "string" comparisons performed during a search operation do not require data manipulation, errors due to lack of precision or rounding are not a concern. To further simplify the implementation of a search algorithm, the beginning of a string can be aligned with the pixel boundaries, which in the worst case wastes another three bytes of space per string. While character strings required an explicit mapping in order to be comparable, the document index pointer docindex, referencing a list of documents containing this search key, is only required for the result. Therefore, copying its bit pattern using some pointer gymnastics is sufficient.

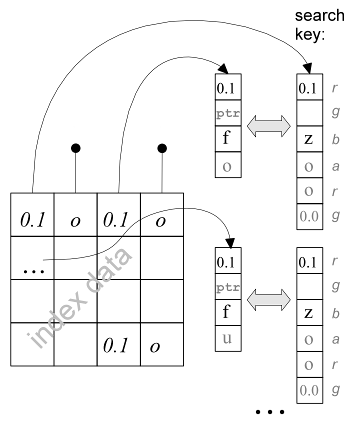

FIGURE 5: STRING SEARCH ON THE GPU USING TEXTURES

With the search key and the data stored in a texture, a naive search can be implemented as a two-step process, using two fragment programs. The first fragment program looks for a match with the search key and marks it (Figure 5), while the second one performs a reduction such that only the marked value is returned (Figure 6). Although this sounds fairly straightforward, the implementation requires some explanation:

```
float4 search(float2 coords: WPOS,
              uniform samplerRECT texCgFrag) : COLOR {
    float2 data_coords = coords;
    float2 searchkey_coords = float2(0.5,0.5);
    float4 data = texRECT(texCgFrag, data_coords );
    float4 searchkey = texRECT(texCgFrag, searchkey_coords);
    float done =0.0;
    if (data.r == 0.1) {
      if (done == 0.0) {
        if (data.b != searchkey.b) done = -1.0;
        if (data.b == searchkey.b)
          if (data.b== 0.0) done = 1.0;
      }
      if (done == 0.0) {
        if (data.a != searchkey.a) done = -1.0;
        ...
```

In order to avoid handling multiple textures, we placed the search key at the beginning of the texture, which has the coordinates (0.5,0.5), the center of the first texel. This might appear odd for conventional arrays, but for graphics this actually makes sense, since a texel does not necessarily mean a pixel on the screen, e.g., when scaling images. Although the comparisons between search key and data elements appear repetitive, they are inevitable, since logical operators and else constructs did not work reliably. If the red color marks the beginning of a string (0.1), this code successively compares the other colors for a match or a terminal symbol (0.0). To support longer strings it can be placed in a while loop that adds coordinate offsets and needs to handle line wraps. In case we find a match, we mark the beginning of the word with another magic number, e.g., red=0.9 and store the index pointer as subsequent color, e.g., green.

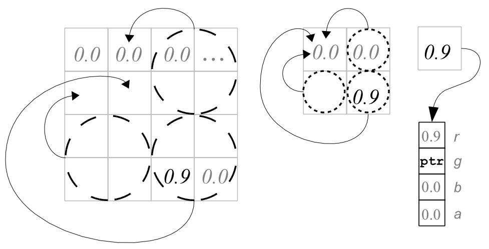

FIGURE 6: REDUCING THE SEARCH RESULT TO A SINGLE RETURN VECTOR, CONTAINING THE INDEX POINTER

In order to execute the search function described above, we actually have to draw the scene, which is accomplished by drawing a rectangle (*quad*) of the texture size:

```
drawQuad(1200,1200);
```

Since the fragment processor does not support memory scatter, i.e., writing the results to a computed location, we implement a reduction function, which after multiple iterations yields a 1x1 texture (Figure 6). In graphics terms a *reduction* consists of multiple rendering passes, which are simply repeated calls of the same function while reducing the texture size, in this case by a factor of two.

```
numPasses = (int)(log((double)width)/log(2.0));
for (i=0; i<numPasses; i++) {
    ...
    outputWidth = outputWidth / 2;
    drawQuad(outputWidth,outputWidth);
    ...
```

For the fragment program this means comparing four pixels whose coordinates are multiples of the current one, in which results are always gathered in the top left fourth of the texture. For fragment programs, the return value is stored at the current coordinate, preferably in another texture to avoid overwriting the original data. On a side note, multiple return points are not supported such that we need another local variable for the result.

```
float4 reduce (float2 coords: WPOS,
               uniform samplerRECT texCgFrag2) : COLOR {
    float2 topleft = ((coords-0.5)*2.0)+0.5;
    float4 val1 = texRECT(texCgFrag2, topleft);
    float4 val2 = texRECT(texCgFrag2, topleft+float2(1,0));
    float4 val3 = texRECT(texCgFrag2, topleft+float2(1,1));
    float4 val4 = texRECT(texCgFrag2, topleft+float2(0,1));
    float4 result = (0.0,0.0,0.0,0.0);
    if (val4.r == 0.9) result = val4;
    if (val3.r == 0.9) result = val3;
    if (val2.r == 0.9) result = val2;
    if (val1.r == 0.9) result = val1;
    return result;
}
```

Eventually, the search result will be located in the top left pixel and can be read back using glReadPixels(0,0,...).

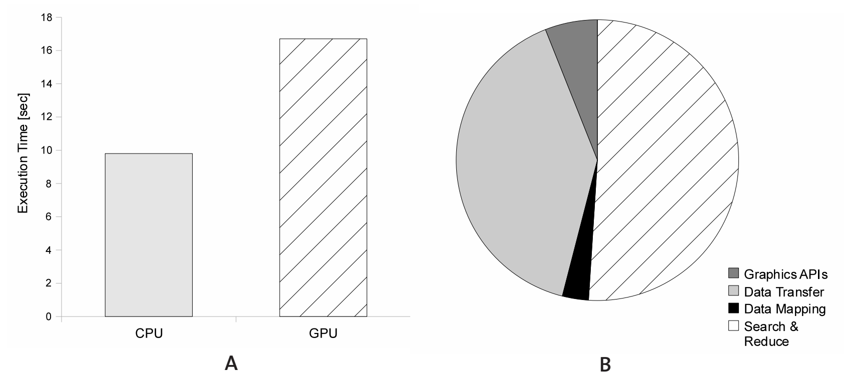

FIGURE 7: A GPU SEARCH IMPLEMENTATION USING GRAPHICS APIS WITHIN BERKELEY DB. (A) EXECUTION TIME OF 10K INSERT/DELETE OPERATIONS, EACH REQUIRING AN INDEX SEARCH. (B) BREAKDOWN OF GPU EXECUTION TIME.

All things considered, the poor performance of this approach does not come as a surprise. Searching for 10,000 values in a few megabytes of data is 60% more time-consuming than computing the same results on the CPU (Figure 7a). Considering that more than 40% of the total GPU execution time is spent on copying data between main and video memory (Figure 7b), a more efficient mapping from data to textures will significantly improve transfer times. For example, we could map substrings to floating-point values or pack multiple characters into one floating-point value. While the former approach might run into issues with rounding errors, in particular on GPUs not implementing full 32-bit precision, the latter requires bit masking and all comparison operations to be performed on bit masks, since floating-point values like “not a number” cannot be compared directly.

While debating with my colleagues how to improve the performance of this first prototype, CUDA 1.0 was released, allowing us to program the GPU directly, natively supporting integer data types. This made any attempts to map data to graphics objects and computation to drawing operations obsolete. Given the poor performance of this implementation and that any new code using graphics APIs for general-purpose implementations would be doomed legacy very soon, we decided to start over with a CUDA implementation.

GPU PROGRAMMING WITH CUDA

As opposed to graphics APIs, CUDA allows programming the GPU directly, using mostly standard C constructs, with all the strings attached. The programmer is in charge of memory management, mode of execution, parallelism, etc.

Since GPU and CPU do not share the same memory address space (see “GPGPU” section, above), CUDA adds a memory copy function, `cudamemcopy()`, that allows copying data to and from the video memory. The GPU does not (yet) support dynamic memory allocation at runtime, and `cudamalloc()` has to be invoked on the CPU(host) side to allocate memory before copying data and/or calling a GPU function accessing data.

Function type qualifiers determine where the code is executed: `global` denotes functions that provide an entry point to GPU code, callable by any CPU code, and `device` functions are only accessible from GPU code. Variable type qualifiers determine their location: `device` denotes variables residing in global memory accessible by all GPU code, while shared variables are located in shared memory (Figure 2), private to each thread block.

An *execution configuration*, placed between function name and parameter list of a call to a GPU function, determines the level of parallel execution. The main configuration options are grid and block dimension. While they are three-dimensional vectors, in the simplest case using only one dimension, they represent the number of thread blocks launched and the number of threads within each block. For example, to run 240 search queries, we could partition them using 30 blocks with block size of 8 to leverage all 30 SMs with 8 PEs each, on a GTX285.

```
dim3 Dg = dim3(30,0,0);
dim3 Db = dim3(8,0,0);
searchGPU< < < Dg,Db > > >(...
```

Besides a little extra memory set-up and copying data, implementating a basic search application with CUDA is fairly straightforward: First, we have to allocate memory for data and for search keys, and, since there is no dynamic memory allocation, also for the results. Then we can transfer the data and search keys to the video card.

```
cudaMalloc((void**)&dataGPU, sizeof(char)*wordlength*words);
cudaMemcpy(dataGPU, dataCPU, sizeof(char)*wordlength*words,
                cudaMemcpyHostToDevice);
cudaMalloc((void**)&searchkeysGPU, ...
```

Transferring larger amounts of data can take a while (e.g., copying the 512MB data set we use for our experiments takes approximately 90ms). In case of read-only operations like search, this is only required at startup.

Adding a global qualifier to the CPU search code above is not sufficient, as standard C library functions are not available. However, there is no shortage of C source code for binary search and string comparison, which can be used without further modification, by simply adding a device prefix. For example, using the original BSD source, a GPU implementation of strcmp is as simple as:

```
__device__ int strcmpGPU(const char* s1, const char* s2){
    while (*s1 == *s2++) {
        if (*s1++ == 0) return 0;
    }
    return (*s1 - *(s2 - 1));
}
```

Given the divided address space, pointers returned by a binary search operation refer to addresses in video memory. Using the base address of the data, they can be easily converted into an offset which is platform-independent. Alternatively, we can implement binary search with base-index addressing. In any case, GPU implementations have to be iterative, since the GPU does not support recursion. The GPU also does not support function pointers, so that function calls to strcmpGPU() have to be explicit.

Retrieving results uses the same mechanism as copying data to the video card, except for the last parameter determining the direction of the memory copy, cudaMemcpyDeviceToHost.

When comparing CPU and GPU query performance, for the GPU we include the time to copy the search keys to video memory and to retrieve the results, but not the time required to copy the data set. It can be reliably placed in video memory for the long term. Although there have been discussions about the absence of error correction [12], we did not experience any discrepancies between CPU and GPU query results across all our experiments, including long runs and large data sets.

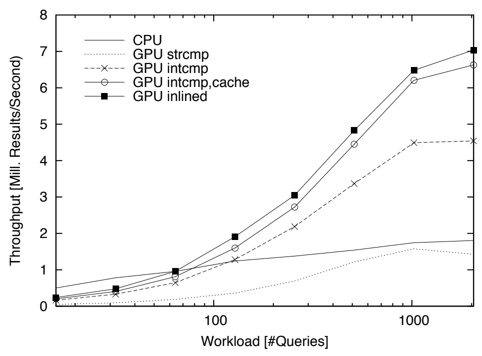

FIGURE 8: PERFORMANCE OF DIFFERENT OPTIMIZATIONS OF GPU SEARCH IN COMPARISON TO A BASIC CPU IMPLEMENTATION

Comparing the performance of this simple approach with a CPU implementation on recent hardware (Core i7 & GTX285) reveals that the GPU cannot keep up (Figure 8). Considering the impressive performance of other database functions implemented on the GPU, e.g., sorting [6], the performance gains or, rather, losses of the above search implementation do not seem very promising. However, given the simple approach we chose for this first implementation, the poor performance is somewhat expected.

Our research on memory performance [8] has shown that small memory accesses can significantly impact memory and, therefore, overall performance of memory-bound applications. Like most database operations, search falls into this category [1]. Without caching, all accesses to search key(s) and pivot element(s) incur full memory latency. This also pertains to consecutive sub-string accesses, as there is no prefetching. Thus we expect that use of vector data types and "manual" caching will increase performance significantly.

Improving memory accesses. As on the CPU, vector data types on the GPU can be used to aggregate small, linear memory accesses. Unlike the CPU, the GPU does not offer byte-wise accessible vectors, but its 4x32-bit integer vectors can be used to load up to 16 bytes at once. For multi-byte words, for example 32-bit integers, the byte order or *endianness* is machine-dependent. Little endian architectures like x86 and NVIDIA GPUs will reverse the byte order, so that integer comparisons applied to character strings produce incorrect results. For example, the character string "dcba" is alphabetically ordered after "abcd." Loaded as a little endian 32-bit integer, an integer comparison would tell us that it is the other way round, `1633837924 < 1684234849`. While x86 CPUs provide the `bswap` instruction to reverse byte order, on the GPU we have to do this manually, e.g., by a macro:

```
#define BSWP(x);\
temp = x << 24;\
temp = temp - ((x << 8) & 0x00FF0000);\
temp = temp - ((x >> 8) & 0x0000FF00);\
x = temp - (x >> 24);
```

The macro can be applied on the fly as it will take only a few cycles, with `x` stored in a register. Unlike on the CPU, there are no hardware instructions available to directly compare vectors, such that we have to resort to a sequential approach:

```
__device__ int intcmp(uint4 *st1, uint4 *st2) {
    int r =1;
    if (BSWP((*st1).x) < BSWP((*st2).x) r=-1;
    else if (BSWP((*st1).x) == BSWP((*st2).x) {
      if (BSWP((*st1).y) < BSWP((*st2).y) r=-1;
      ...
```

The individual components of a vector are compared with decreasing significance until a decision can be made if one of them is larger than the other or if they are equal. Although these are CUDA integer vectors, they still use coordinates addressing x, y, z, w. After de-referencing the pointers, the vector elements are stored in registers, such that even this branch-intensive comparison will only take a few cycles.

Using this approach, we reduced the number of memory requests by a factor of four, at the cost of a few additional instructions to handle the data in integer format. As a result, performance increases by nearly a factor of four (Figure 8).

Caching. Although the GPU does not employ caches in the traditional sense, its shared memory with only a few latency cycles can be used as a user-managed cache. For example, caching the search key and the pivot element before calling the comparison functions further reduces the number of memory accesses by a factor of four to a total of two 128-bit global memory requests per search iteration.

```
__shared__ uint4 cache[2*BLOCKSIZE] ;
...
cache[threadidx.x*2] = *searchkey;
cache[threadidx.x*2+1] = *pivotelement;
res = intcmp(&cache[threadidx.x*2], &cache[threadidx.x*2+1]);
...
```

While using shared memory as a cache to alleviate the memory bottleneck significantly increases overall performance (Figure 8), it comes with strings attached. The amount of local memory used by a thread block determines the number of blocks that can be handled by a single SM (occupancy). However, in our case the amount of shared memory required for caching is small enough (304 bytes/block) that it does not impact occupancy (Table 1) but reduces the number of global memory accesses by more than a factor of three.

Algorithm	Occupancy	Shared Memory per Block	Registers per Thread	Global Memory Accesses
strcmp	25%	48 bytes	19	7,012,536
intcmp	25%	48 bytes	19	688,046
intcmp cached	25%	304 bytes	19	200,476
inlined	33%	48 bytes	14	198,310

TABLE 1: CUDA PROFILER RESULTS FOR DIFFERENT SEARCH IMPLEMENTATIONS, RUNNING 65K SEARCH QUERIES AGAINST A 512MB DATA SET

Further optimizations. Although structuring code by using functions and pointers to reduce parameter overhead are good coding practices, they are not necessarily optimal from a performance point of view. Each function invocation comes with a large overhead: allocating a new stack frame, saving registers, etc. Since the GPU does not implement dynamic memory allocation, each function invocation will use up additional registers, similar to the way shared memory impacts occupancy. Pointers intended to reduce register usage are not very helpful in environments like the GPU with thousands of registers available. The absence of caching makes pointer resolution for consecutive addresses particularly painful due to repeated round trips to memory; the use of registers would eliminate this issue.

The core functions of this application, binary search and string comparison, are small enough to inline them into a single global function with 35 lines total. This step eliminates any use of shared memory and decreases register usage and global memory accesses (Table 1). Since this approach also eliminates function call overheads, it provides the best overall performance using well-known algorithms (Figure 8).

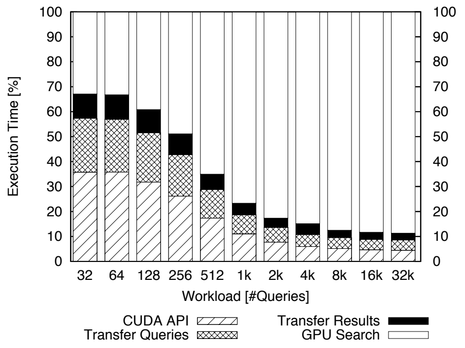

FIGURE 9: TIMING BREAKDOWN FOR OFFLOADING BATCHES OF SEARCH OPERATIONS TO THE GPU

The poor performance for small workloads is the result of inefficient resource utilization and the overhead involved in starting GPU computation (Figure 9). Small workloads do not invoke sufficient threads to leverage the GPU's seamless multi-threading to hide memory latency. To measure the execution time of each individual step, we run exactly the same batch of queries multiple times, each time adding another step in the offloading process. We obtain the time required for a step by computing the difference to the previous run. For example, the API launch time is determined by executing an empty program. The time for transferring a batch of queries to the GPU is determined by subtracting the time required to launch an empty program from the time required for launching the program and copying the queries to the video card, and so on.

In order to achieve maximum performance on parallel architectures like video cards, not only in terms of throughput but also in terms of response time, parallel algorithms are required. For a parallel search algorithm I would like to refer the reader to our recent HotPar publication which introduces p-ary search [9]. I am currently working on a p-ary search implementation for multi-core CPUs and expect a head-to-head race between similarly priced CPUs and GPUs.

GPGPU: Quo Vadis?

To answer the question of where GPU programming, and parallel programming in general, is heading, I would like to refer to the numerous presentations by major chip manufacturers at HotChips '08. While GPUs clearly evolve in terms of programmability, the core/thread count in CPUs is continuously increasing. For example, NVIDIA announced a CUDA debugger and a profiler [2], while Sun announced the Niagara successor, named Rock, with 16 cores, each of them supporting four hardware threads [4]. Intel's Larrabee architecture represents the next logical step for CPU and GPU architectures, combining many cores with x86 programmability [3].

If you were to ask me what I would like to see next, I would say a fully integrated, fully programmable, many-core chip—i.e., plugging into a standard CPU socket, sharing the memory with all other processors, and offering full OS support. As far as programmability is concerned, I am looking forward to evaluating OpenCL [13], which claims to be a transparent programming API for multi- and many-core environments and is backed by major manufacturers (e.g., Intel, AMD, IBM, NVIDIA). The two together could eliminate the bitter taste of explicit co-processor programming and distributed memory architectures.

REFERENCES

[1] A. Ailamaki, D.J. DeWitt, M.D. Hill, and D.A. Wood, "DBMSs on a Modern Processor: Where Does Time Go?" *VLDB '99*.

[2] I. Buck, "CUDA Tutorial," *Hot Chips '08*.

[3] D. Carmean. "Larrabee: A Many-Core x86 Architecture for Visual Computing," *Hot Chips '08*.

[4] S. Chaudhry, "Rock: A SPARC CMT Processor," *Hot Chips '08*.

[5] M.J. Flynn, "Very High-speed Computing Systems," *Proceedings of the IEEE* 54(12), 1966.

[6] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management," *SIGMOD '06*.

[7] GPGPU.org, "General-Purpose Computation on Graphics Hardware," 2009: http://www.gpgpu.org.

[8] T. Kaldewey, A.D. Blas, J. Hagen, E. Sedlar, and S.A. Brandt, "Memory Matters," WiP session of *RTSS '08*.

[9] T. Kaldewey, J. Hagen, A. Di Blas, and E. Sedlar, "Parallel Search on Video Cards," *HotPar '09*.

[10] NVIDIA, CUDA Zone, 2009: http://www.nvidia.com/cuda.

[11] K. Schlegel, "Emerging Technologies Will Drive Self-Service Business Intelligence," Gartner Report #G00152770, 2008.

[12] J. W. Sheaffer, D.P. Luebke, and K. Skadron, "A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors," *Graphics Hardware '07*.

[13] The Khronos Group, "OpenCL—The Open Standard for Parallel Programming of Heterogeneous Systems," 2009: http://www.khronos.org/opencl.