16 Comments (http://blogs.nvidia.com/blog/2012/09/12/how-tesla-k20-speeds-up-quicksort-a-familiar-comp-sci-code/#disqus_thread)
3007

(http://blogs.nvidia.com/blog/2012/09/12/how-tesla-k20-speeds-up-quicksort-a-familiar-comp-sci-code/)

HOW TESLA K20 SPEEDS QUICKSORT, A FAMILIAR COMP-SCI CODE (http://blogs.nvidia.com/blog/2012/09/12/how-tesla-k20-speeds-up-quicksort-a-familiar-comp-sci-code/)

By Stephen Jones (http://blogs.nvidia.com/blog/author/stephenjones/) on September 12, 2012

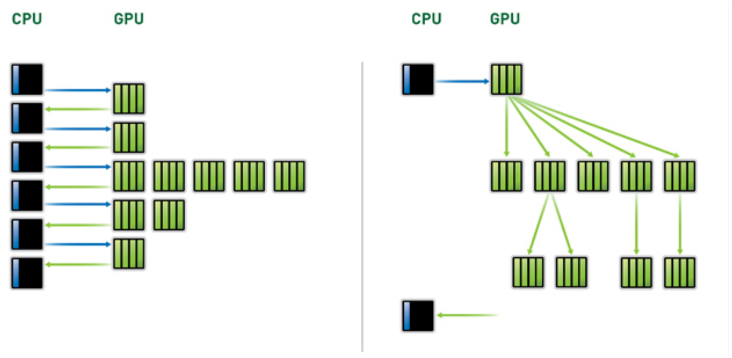
We promised that the Kepler-based NVIDIA Tesla K20 GPU (http://www.nvidia.com/object/tesla-servers.html) – first unveiled in May at the GPU Technology Conference (GTC) (http://www.gputechconf.com/page/home.html) – would be the highest-performance processor the HPC industry has ever seen. One reason: support for a technology called "Dynamic Parallelism," that can speed up a wide range of tasks.

Dynamic Parallelism allows the GPU to operate more autonomously from the CPU by generating new work for itself at run-time, from inside a kernel. The concept is simple, but the impact is powerful: it can make GPU programming easier, particularly for algorithms traditionally considered difficult for GPUs such as divide-and-conquer problems.

To showcase its potential, I will use Quicksort—a universal requirement for all Computer Science 101 students—to show how Dynamic Parallelism cuts the lines of code needed for Quicksort in half while improving performance by 2x.

Under the Hood

Let's begin with a bit of background. On GPUs based on the current Fermi architecture (http://www.nvidia.com/object/fermi-architecture.html), there exists a one-way, fixed execution flow from the host CPU to the cores in the GPU. This is illustrated on the left side of the chart below.



(http://blogs.nvidia.com/2012/09/how-tesla-k20-speeds-up-quicksort-a-familiar-comp-sci-code/firstpicture/)

(Left): Without Dynamic Parallelism, (Right): With Dynamic Parallelism

With Dynamic Parallelism, the GPU is able to generate new work for itself without involving the CPU at all. This permits dynamic run-time decisions about what to do next, enabling much more complex algorithms than previously were possible (illustrated on the right side of the chart), while simultaneously releasing the CPU to conserve power or perform other work.

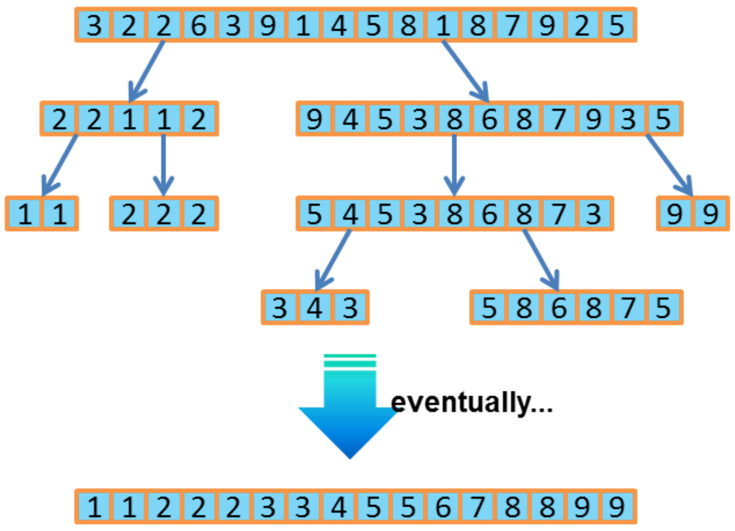
To handle this dynamic work, NVIDIA created a new hardware technology in Tesla K20 GPUs called the Grid Management Unit (GMU). This manages the complexities of dynamic execution at hardware speed – launching, suspending and resuming kernels, as well as tracking dependencies from multiple sources. A layer of system software running on the GPU interacts with the GMU, enabling the CUDA (http://www.nvidia.com/object/cuda_home_new.html) Runtime application-programming interface (API) to be used from within a kernel program.

Quick and Dirty with the Quicksort Algorithm

So now let's move on to the Quicksort algorithm, which provides a great example of the power of Dynamic Parallelism.

First, a quick reminder of how it works. The goal is to sort an array of numbers, and I begin by picking a "pivot" value which I use to partition my array into two smaller arrays: one with values less than the pivot, and one with values equal or greater.

In the diagram below, I'm simply using the first element of each array as its pivot:



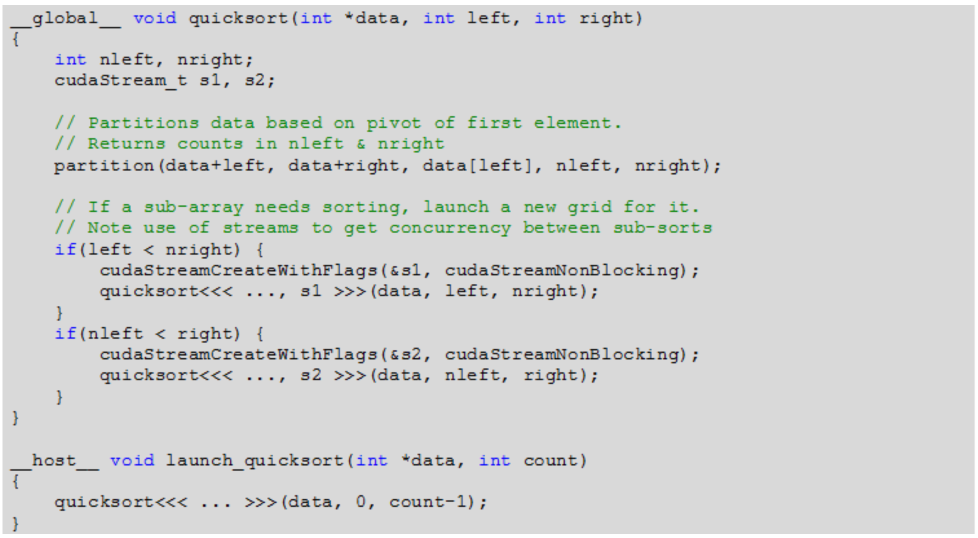
(http://blogs.nvidia.com/2012/09/how-tesla-k20-speeds-up-quicksort-a-familiar-comp-sci-code/secondimage/)

After partitioning the initial array, the algorithm then launches two new quick sorts on the two new arrays, producing four sub-arrays and so on until each sub-array contains just a single value; the result is put together and you're done. It's a classic "divide-and-conquer" algorithm because it breaks the problem into ever smaller pieces and solves them recursively.

Quicksort Made Easy – Cutting Lines of Code in Half

Now let's take a look at the actual CUDA code for Quicksort, with and without Dynamic Parallelism.

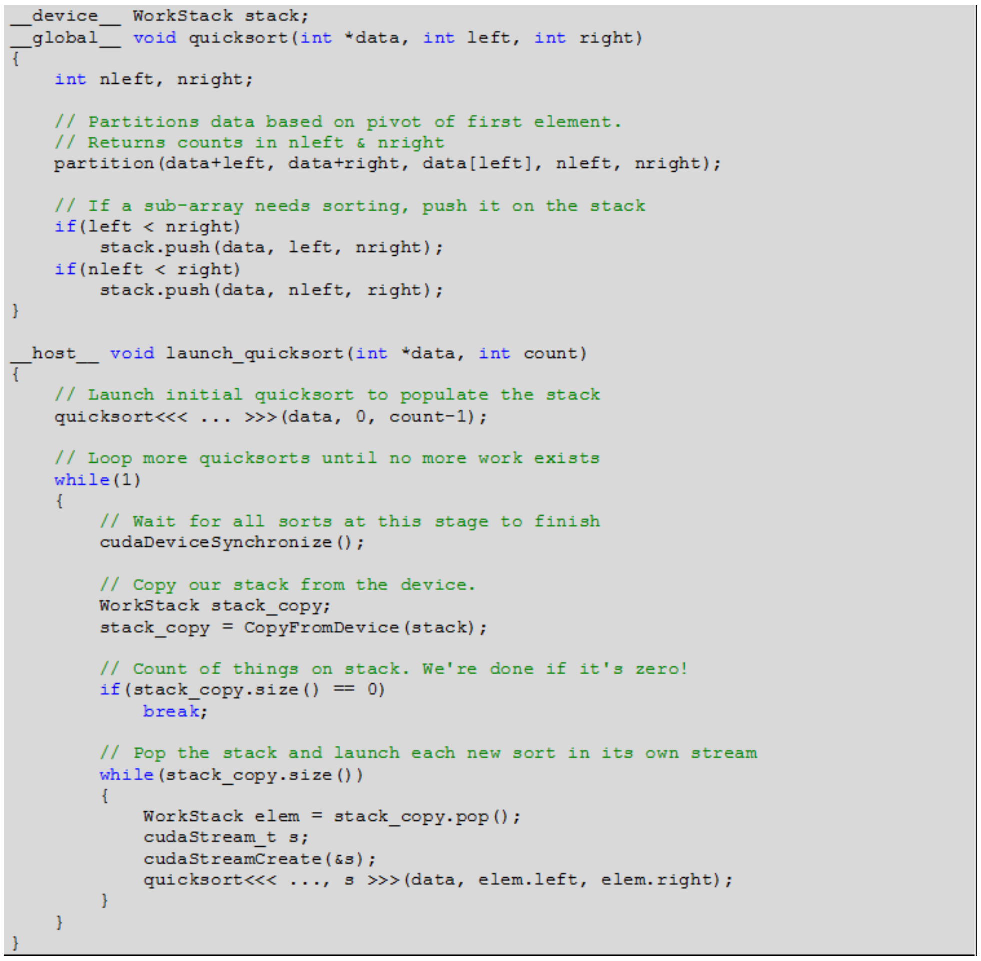
Quicksort with Dynamic Parallelism



```
__global__ void quicksort(int *data, int left, int right)
{
    int nleft, nright;
    cudaStream_t s1, s2;

    // Partitions data based on pivot of first element.
    // Returns counts in nleft & nright
    partition(data+left, data+right, data[left], nleft, nright);

    // If a sub-array needs sorting, launch a new grid for it.
    // Note use of streams to get concurrency between sub-sorts
    if(left < nright) {
        cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
        quicksort<<< ..., s1 >>>(data, left, nright);
    }
    if(nleft < right) {
        cudaStreamCreateWithFlags(&s2, cudaStreamNonBlocking);
        quicksort<<< ..., s2 >>>(data, nleft, right);
    }
}

__host__ void launch_quicksort(int *data, int count)
{
    quicksort<<< ... >>>(data, 0, count-1);
}
```

(http://blogs.nvidia.com/2012/09/how-tesla-k20-speeds-up-quicksort-a-familiar-comp-sci-code/texta-4/)

Quicksort without Dynamic Parallelism



```
__device__ WorkStack stack;
__global__ void quicksort(int *data, int left, int right)
{
    int nleft, nright;

    // Partitions data based on pivot of first element.
    // Returns counts in nleft & nright
    partition(data+left, data+right, data[left], nleft, nright);

    // If a sub-array needs sorting, push it on the stack
    if(left < nright)
        stack.push(data, left, nright);
    if(nleft < right)
        stack.push(data, nleft, right);
}

__host__ void launch_quicksort(int *data, int count)
{
    // Launch initial quicksort to populate the stack
    quicksort<<< ... >>>(data, 0, count-1);

    // Loop more quicksorts until no more work exists
    while(1)
    {
        // Wait for all sorts at this stage to finish
        cudaDeviceSynchronize();

        // Copy our stack from the device.
        WorkStack stack_copy;
        stack_copy = CopyFromDevice(stack);

        // Count of things on stack. We're done if it's zero!
        if(stack_copy.size() == 0)
            break;

        // Pop the stack and launch each new sort in its own stream
        while(stack_copy.size())
        {
            WorkStack elem = stack_copy.pop();
            cudaStream_t s;
            cudaStreamCreate(&s);
            quicksort<<< ..., s >>>(data, elem.left, elem.right);
        }
    }
}
```

(http://blogs.nvidia.com/2012/09/how-tesla-k20-speeds-up-quicksort-a-familiar-comp-sci-code/textb/)

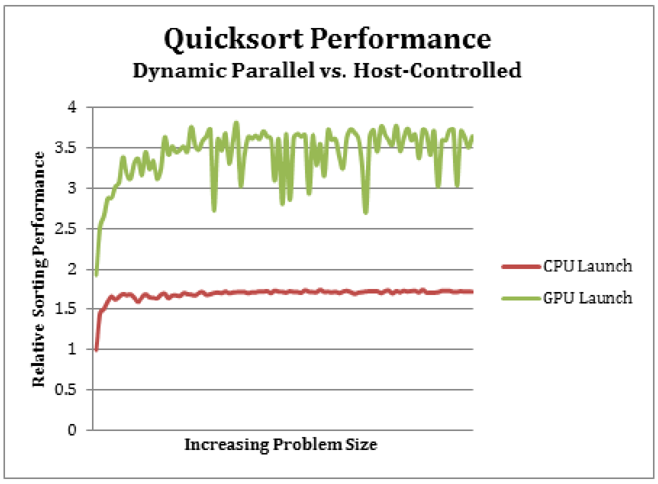
Even if you aren't a programmer you'll notice that Quicksort with Dynamic Parallelism is half the size of the code without it. And it's much easier to follow. Here's why.

In Quicksort, the information needed to sort each stage depends on the stage before it. Without Dynamic Parallelism all of the launches must take place from the CPU, which means that the details of what to launch next must be passed back to the host after each stage. For simplicity, the example encapsulates this communication in a CPU/GPU work stack; this can be highly complex in its own right, requiring atomics, data management, and as much code as the Quicksort algorithm itself.

But, with Dynamic Parallelism the GPU performs its own launches on-the-fly, enabling each Quicksort to launch its two sub-sorts as soon as it has finished. There are no complex overheads like the CPU/GPU stack exchange, and no need for all the host code which manages the launches. The whole thing is shorter, easier to understand and as we shall see next, faster.

Dynamic Parallelism Boosts Performance

We benchmarked the above two approaches on the same Tesla K20 GPU, and the results are shown in the graph below: Quicksort with Dynamic Parallelism delivered a 2x speed-up compared to the code without Dynamic Parallelism.



(http://blogs.nvidia.com/2012/09/how-tesla-k20-speeds-up-quicksort-a-familiar-comp-sci-code/thirdimage/)

The reason for the speedup is closely connected to launch strategy. The CPU-controlled code must wait for each stage to complete before launching into the next stage, requiring a *cudaDeviceSynchronize()* (http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf) call at each stage. Not only is this a heavy-duty operation, but it forces all sorts in a stage to finish before any sub-sort can begin – in effect, each stage goes as slowly as its longest operation.

By contrast, the dynamic parallel code simply launches work as and when it is needed. There's no need to transfer data between GPU and CPU. There's no need to wait for each stage to complete before starting the next. We get much better overlap of work with much lower management overhead.

As you can see, not only is the code much easier to write—and read—but it's also significantly faster.

Limitless Possibilities

I believe Dynamic Parallelism will revolutionize GPU computing by delivering three powerful benefits:

1. Programming the GPU will be easier than ever;
2. Algorithms previously considered difficult will now accelerate easily on GPUs;
3. The GPU depends significantly less on the CPU, enabling both to operate more efficiently.

With Dynamic Parallelism, the possibilities are endless. Over the next few weeks, I will write about two more of these powerful use cases: implementing complex algorithms by calling parallel libraries directly from the GPU, and maximizing GPU utilization by easily batching lots of small jobs together.

If you have any suggestions for codes that would benefit from Dynamic Parallelism please share them below. We are in the process of putting sample codes into the CUDA Toolkit (https://developer.nvidia.com/cuda-toolkit), and would like to hear your ideas.

For more on Kepler and some of its cool features follow @NVIDIATesla (http://www.twitter.com/nvidiatesla).

Categories: Supercomputing (http://blogs.nvidia.com/blog/category/enterprise/supercomputing/)

Tags: CUDA (http://blogs.nvidia.com/blog/tag/cuda/), developer (http://blogs.nvidia.com/blog/tag/developer/), Dynamic Parallelism (http://blogs.nvidia.com/blog/tag/dynamic-parallelism/), GPU Computing (http://blogs.nvidia.com/blog/tag/gpu-computing/), GPUs (http://blogs.nvidia.com/blog/tag/gpus/), high performance computing (http://blogs.nvidia.com/blog/tag/high-performance-computing/), hpc (http://blogs.nvidia.com/blog/tag/hpc/), Hyper-Q (http://blogs.nvidia.com/blog/tag/hyper-q/), k20 (http://blogs.nvidia.com/blog/tag/k20/), Kepler (http://blogs.nvidia.com/blog/tag/kepler/), Quicksort (http://blogs.nvidia.com/blog/tag/quicksort/), Supercomputing (http://blogs.nvidia.com/blog/tag/supercomputing/), Tesla (http://blogs.nvidia.com/blog/tag/tesla/)

Similar Stories

What the Heck Is a GRID vGPU? (http://blogs.nvidia.com/blog/2015/03/02/what-

Mapping the World with GPUs Reveals Po

How GPUs Are Helping Pinpoint Toxins in Everyday Household

GPUs Help Unfold the Loops in Genome's Graceful Structure

Comments for this thread are now closed. ×

16 Comments The NVIDIA Blog Login

Sort by Best Recommend Share

Jack Jones • 2 years ago
It would be nice to have the partition source code...
3 • Share ›

Jake • 10 months ago
There are several errors in the Dynamic Parallelism code.

The logic to check if there is a left sub-array to sort is currently:

if(left < nright)

but it should be

if(left < nleft)

because when left == nleft, this implies there is no left sub-array to sort, the if-check returns false, and there is no quicksort call on the left sub-array.

Likewise, the recursive quicksort call for the left sub-array is incorrect. It should be

quicksort<<<...s1>>> (data, left, nleft);

as the left sub-array's bounds go from [left,....,nleft].

It's a bit laughable that you include the partition function with no comment on how it is

see more

1 • Share ›

david macpherson • 2 years ago
Stephen,

Thanks for the quicksort example. Perhaps including a link to the partition function & a makefile would be helpful.
1 • Share ›

Jackson Beatty • 2 years ago
One or more code samples using Thrust in the context of dynamic parallelism would be very helpful.
1 • Share ›

Dave Ojika → Jackson Beatty • a year ago
well said!! exactly what I am looking for. did you find anything on that?
• Share ›

tpofofnt • 2 years ago
Stephen,

In a recent talk, you made the comment that the overhead for launching a kernel on the device is precisely the same as the overhead for launching a kernel on the host. You went on to say that if device kernel launches were batched, say in a batch of 250 launches, the overhead for each kernel launch would be 1/250 the overhead of a host kernel launch.

What is this batch kernel launch from the device you speak of? Are you referring to the situation where every thread from a parent kernel launches the same child kernel?

Thanks in advance!

Mitch Horton
• Share ›

Jonh Rain • 2 years ago
Hello. I just got a GTX660 thinking that i could use dynamic parallelism, but it seems only the GTX TITAN can do it? is that correct?
• Share ›

Sagar Rawal • 2 years ago
Congratulations on a fantastic demonstration of the hardware prowess of Tesla K20!

The anticipation builds in everyone for the release of such a groundbreaking product!
• Share ›

Guest • 2 years ago
one small question, you first call the kernel with one block of 1 thread? not sure about that.
• Share ›

Biao Wang Guest • 2 years ago
I have the same question, what is the grid configuration which is substituted by three dot in each kernel call?
1 • Share ›

@heg53 • 2 years ago
I have a new algorithm of my authorship. it is based on the principle of divide and conquer and I have programmed in C language. I think that could be calculated in this architecture. Do you could program for K-20 if I send the code?.
regards
• Share ›

nvjones @heg53 • 2 years ago
We're always interested to hear about algorithms where dynamic parallelism might apply - could you be more specific about what you are working on? I'm afraid I won't personally be able to help with porting your code, but I've often found the people on www.stackoverflow.com to offer good advice and help.
• Share ›

@heg53 nvjones • 2 years ago
Very nice for taking the time to answer. With pleasure I send an article and source code in C language for you to see if you can run it in this architecture. I hope I can be useful and we can use it as an example of numerical efficiency. The numerical complexity of this algorithm is exponential and if all goes well may be reduced to polynomial complexity. If everything turned out well, I would expect us to publish something and you could advertise the algorithm as an example.Could you send a mail where to send this safely?

regards
• Share ›

oscar barenys • 2 years ago
First congratulations for sharing such early experiences on GK110 with all people..

Hope you have time to answer some questions/suggestions (and hope aren't much stupid after all):

*to me the code is not the same as without dyn parallelism as you could also use 2 different streams and then should concurrent kernel execution (which should be working with Hyper-Q avoiding false dependencies).. should with that optimizations be even 2x speedup?

*Is this sample scaling over GK104 I mean is code without dyn parallelism scaling over GK104 nearly proportionaly to #SMs GK110 / #SMs GK104 assuming same SM clocks?

*Seems this "simple quick sort code" is not the most efficient.. I mean it would not achieve peak speed of Duane Merill sort code integrated into Thrust.. Do you expect using dynamic parallelism code in the highest perf codes such as Duane Merill or MSORT to bring also such high speedups (2x) over current code on GK110.. i.e. can we expect thanks to that very high speedups in sorting in GK110 HW..

*Seems currently best sorting rates for "small" arrays are by a wide margin achieved using

see more

• Share ›

nvjones oscar barenys • 2 years ago
Thanks for the detailed response! You've asked a lot of questions, so I'll work through them one at a time (I'm numbering based on your * comments above).

1. The "without dynamic parallelism" code does use separate streams for each launch of a stage (see the last 3 lines of the code sample), so this should be a fair comparison. The limiter is not concurrency between kernels of a given stage, but rather that each stage must finish before the CPU can launch the next one

rather that each stage must finish before the GPU can launch the next one.

2. The graph shows both runs performed on a K20. I have not compared the host-launch algorithm performance between K10 and K20, although I would expect to see equivalent behaviour because the limiting factor is the inter-stage synchronisation overhead.

3. Quicksort is a comparison sort, which allows sorting of arbitrary data, so it cannot be compared with Radix Sort which can only compare bitfield sorts (for example, you could not radix-sort complex numbers based on modulus). The time complexity of the two algorithms is different because they perform different tasks. There is no need to use dynamic parallelism for bitfield sorts such as radix sort, because there is no intrinsic data dependence and the GPU already performs very well at these

see more

• Share ›

oscar barenys nvjones • 2 years ago

Really thanks for your time and detailed response.. it shows I have more excitement than knowledge in fast GPU sorters :-)

I won't promise you but I think this is the last mega post here so I won't take more of your time.. I think I have some good suggestions for 5.0 SDK I have been thinking lately.. perhaps sorry posting here but as said using Nvidia forums is no option now..

Before that only say I'm happy NV is well recieving the suggestion on starting somthing like Intel Many Core testing lab.. hope it materializes soon after time of Tesla K20s release..

Here are the suggestions for CUDA 5.0 SDK:
(mostly I'm interested in graphics/CUDA interop for questions clearer at the bottom)

*Seems CUDA 5.0RC ships with CUDA BLAS device library for GK110 altough no documentation present currently and perhaps even not support in CUBLAS headers for using it.. Hope this gets fixed and a simple simple using

see more

1 • Share ›

Subscribe Add Disqus to your site Privacy

Next Post »

« Previous Post (http://blogs.nvidia.com/blog/2012/09/11/post-tsunami-rebuilding-one-sweet-seaweed-cake-at-a-time/)