Geosci. Model Dev. Discuss., 7, C3691–C3699, 2015
www.geosci-model-dev-discuss.net/7/C3691/2015/

Geoscientific Model Development Discussions
Open Access

Interactive comment on "gpuPOM: a GPU-based Princeton Ocean Model" *by* X. Huang et al.

X. Huang et al.

hxm@tsinghua.edu.cn

Received and published: 3 May 2015

Dear David:

First of all, we would like to express our sincere appreciation to your valuable feedback. Your comments are highly insightful and enable us to significantly improve the quality of our manuscript and our program. The following pages are our point-by-point responses to each of your comments.

(1) "After a fairly standard introduction, the key description of the Nvidia K20X unit and the CUDA low level programming model, involving warps and streaming multiprocessors, is poorly written and confusing. There is also very little on the K20X memory and cache hardware although these will always have a major impact on the structure of the optimum code." "The authors need to improve their description of the hardware and software models."

[Response]:

We plan to rewrite the relevant parts in section 3 as you suggested in the revised manuscript. We will add a paragraph to describe the GPU architecture overview, the CUDA programming model involving the warp and streams, and the hardware execution model involving the stream multiprocessors(SM) and scheduling policy.

(2) “I am also concerned that there is no proper discussion about how best to deal with the large ocean model arrays in a cache based system. The code continues to use the east-west index as the innermost array index, although with a cache it may be more efficient to use the vertical index. Although not mentioned in the paper, the code shows that many of the innermost loops have been changed to vectorise in the vertical. “

“They also need a proper quantitative discussion of how the ocean model is fitted into memory and cache, and where the bottlenecks are when running the model.”

[Response]:

We will make a comprehensive comparison between mpiPOM and gpuPOM and describe how they benefit from the cache system architecture. In general, the biggest difference between GPU and CPU is that, in GPU, programmers can artificially choose which array to store in cache. Moreover, GPU provides various on-chip caches, such as L1/L2 cache, shared memory, texture cache. Thus, according to how the arrays are used, we can put different arrays in different caches. We focus on exploring a better data placement on different caches for different terms, rather than conventional cache blocking optimizations. We will detail our optimizations for cache optimizations as follows:

“For the loop/subroutine fusion optimization, global memory is cached in registers for later reuse. Take the Algorithm 2 in Section 3.1 as an example. After loop fusion, each thread accesses the z-axis of “drhox” once, rather than twice.

For the local memory blocking optimization, such as Algorithm 1 in Section 3.1. Before we use local memory to block ee and gg, these two arrays in global memory cannot be resident in L1 cache, which is a feature of K20X. After blocking ee and gg in local memory, each thread can get its own ee_new and gg_new from L1 cache, instead of accessing global memory. However, since there are usually more than 500 active threads in each SM to keep enough parallelism/occupancy, ee_new and gg_new of some threads may not completely be resident in L1 cache(48KB). But, compared with raw implementation that every array is accessed in global memory, such method is more acceptable.

For the read-only data cache optimization, global memory is cached in read-only data cache (also called texture cache). It makes use of the spatial locality that data accessed by one thread will be accessed by adjacent threads. This cache acts much like the conventional cache on CPU, with the only difference that we can choose which array will be cached in it."

We will enrich the first paragraph in Section 3 to discuss our consideration on the innermost array index of gpuPOM. Moreover, we will add a citation on mpiPOM's design for multiprocessor system(Masumoto, 1999). We agree that it is advantageous to use vertical index as the innermost array index for ocean models on CPU, while the mpiPOM uses east-west index as the innermost array index. However, for gpuPOM, since we use CUDA programming model, we have to adopt 2-D block decomposition (i and j) to guarantee enough parallelism. Then we have to make east-west (i) as innermost index to satisfy GPU memory coalescing. Each GPU thread does all the computation along z axis as described in Section 3, thus the whole domain is actually calculated one horizontal layer by one horizontal layer.

In terms of the bottleneck, we think it is the memory access rather than floating point execution that determine the performance. That is, POM is memory-bounded. In addition, we will add a paragraph to demonstrate it: "To demonstrate the memory-bound problem, the Performance API(PAPI) is used to estimate floating point operations count and memory access(store/load) instructions count. Results show that the computational intensity(flops/byte) of the mpiPOM is about 1:3.3, while it is 7.5:1 for that provided by SandyBridge E5-2670 CPUs. Moreover, data is mostly streamed from memory and shows little locality. According to the roofline model(Williams, 2009), the whole POM is mainly memory-bounded. However, the mpiPOM suffers from an average profiling results, with even the most time-consuming subroutine just occupying 20% of the total execution time. Thus, only porting some subroutines does not help for the overall performance, and it has to be the whole model to be ported."

(3)"The amount of effort spent converting a Fortran code to C and CUDA-C is also odd, given that a CUDA-Fortan compiler has been available since 2009 and that in future POM is likely to stay a Fortran code - if for no other reason than the simplicity of loop optimisation with this compiler."

[Response]:

We agree that using PGI CUDA Fortran is indeed the most convenient way for model porting as a lot of efforts can be saved. And we will add a paragraph describing our choice in the revised manuscript. Actually exploring the performance potential using CUDA Fortran is also part of our plans after this work based on CUDA-C. But we choose to use CUDA-C in the current version of gpuPOM because

a) CUDA C is free of charge while CUDA-Fortran for one workstation costs more than $1000.(https://www.pgroup.com/pricing/bcwsa.htm)

b) Previous work(Henderson, 2011) showed that, during the porting of Nondydrostatic Icosahedral Model(NIM), the resulting executable program of the commercial CUDA-Fortran compiler did not perform as well as the manually converted CUDA-C version.

c) The read-only data cache is not supported in CUDA-Fortran, which is the key optimization of Section 3.1.1

d) We already have a lot of previous experiences for deep optimizations with CUDA-C

(4) “However because the main subroutines updating the tracer and velocity fields have been split into a number of small GPL kernels, many opportunities to make better use of the cache and to reduce cache loads have been missed.” “Finally I think the code needs to be rewritten to drastically reduce the number of independent kernels. This is because once a cache contains the temperature, salinity, velocity and grid arrays for a small region of ocean, it appears senseless not to update the values of all of these variables.”

[Response]:

We agree that there are many GPU kernels in gpuPOM and kernel fusion can surely improve data locality usually. In fact, we have adopted kernel fusion in the current gpuPOM, as described in Section 3.1.4. More aggressive kernel fusion is a part of future work.

The main reason of such many kernels in gpuPOM is that there exist a large number of subroutines in mpiPOM. Since we port the entire model one subroutine by one subroutine, which is a convenient way to debug the gpuPOM and to guarantee its bit-by-bit identical results to mpiPOM, we have to write a large number of gpu kernels.

In version 1.0 of gpuPOM, we have broken several subroutines in mpiPOM into multi GPU kernels in gpuPOM in 3 cases:

a) when subroutine B is invoked in subroutine A, A is broken into 2 small kernels, as shown in Fig.1.

b) when a MPI function call is invoked in subroutine A, A is broken into 2 small kernels, as shown in Fig.2.

c)when interior array is first writed by one thread and later read by adjacent threads, where caching this array in shared memory can not benefit, subroutine A is broken into 2 small kernels, as shown in Fig.3. We plan to add a paragraph in the revised manuscript describing the reason of such many kernels in gpuPOM.

We really appreciate your highly constructive comments. We hope our responses will address your concerns.

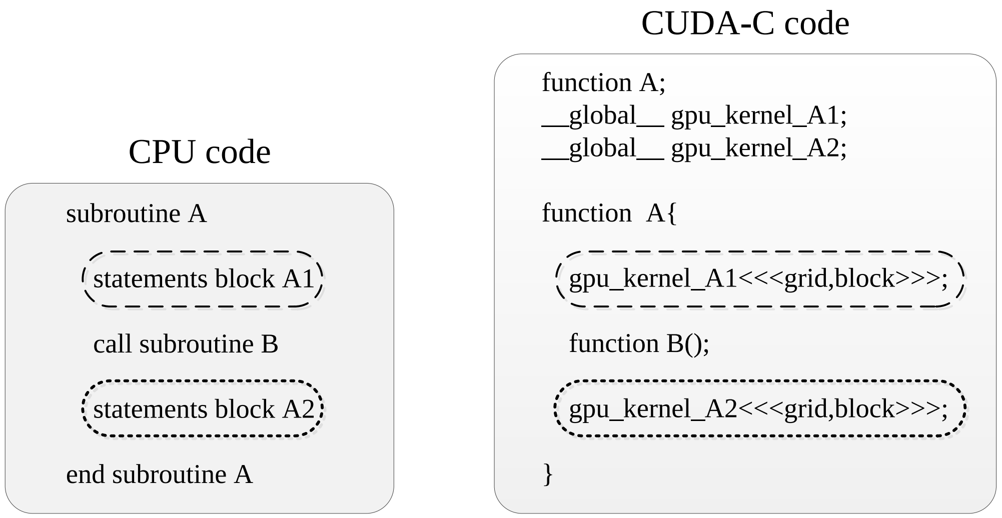
Best wishes, Xiaomeng Huang

References

Masumoto, Yukio, Takashi Kagimoto, Toshio Yamagata, Masahiro Yoshida, Masahiro Fukuda, and Naoki Hirose. "Simulated circulation in the Indonesian archipelago from a high resolution global ocean general circulation model on the numerical wind tunnel." In Proceedings of the 1999 ACM/IEEE conference on Supercomputing, p. 35. ACM, 1999.

Henderson, Tom, J. Middlecoff, J. Rosinski, M. Govett, and P. Madden. "Experience applying Fortran GPU compilers to numerical weather prediction." In Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on, pp. 34-41. IEEE, 2011.

Williams, Samuel, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures." Communications of the ACM 52, no. 4 (2009): 65-76.



```
subroutine A
    statements block A1
    call subroutine B
    statements block A2
end subroutine A
```

```
function A;
__global__ gpu_kernel_A1;
__global__ gpu_kernel_A2;

function  A{
    gpu_kernel_A1<<<grid,block>>>;
    function B();
    gpu_kernel_A2<<<grid,block>>>;
}
```

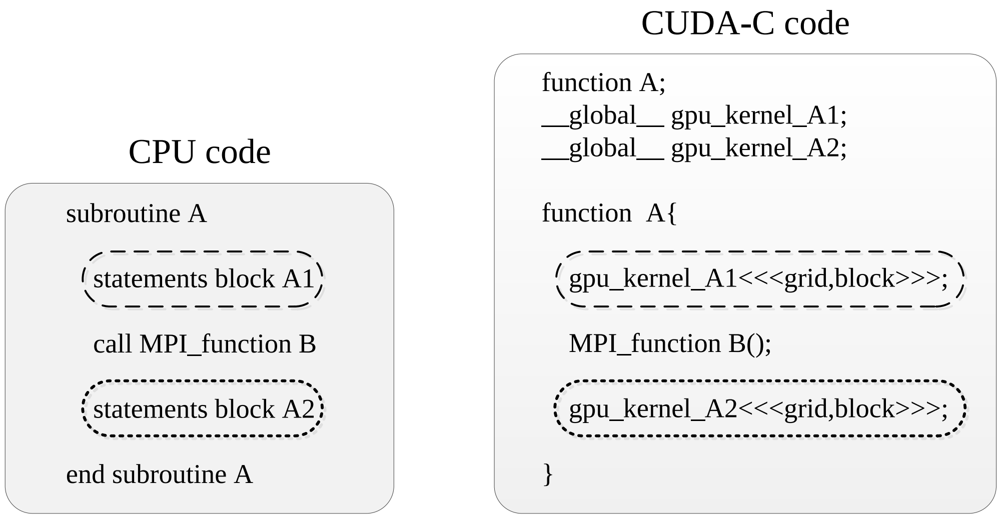
Fig. 1. when subroutine B is invoked in subroutine A, A is broken into 2 small kernels.

Full Screen / Esc

Printer-friendly Version

Interactive Discussion

Discussion Paper



```
subroutine A
    statements block A1
    call MPI_function B
    statements block A2
end subroutine A
```

```
function A;
__global__ gpu_kernel_A1;
__global__ gpu_kernel_A2;

function  A{
    gpu_kernel_A1<<<grid,block>>>;
    MPI_function B();
    gpu_kernel_A2<<<grid,block>>>;
}
```

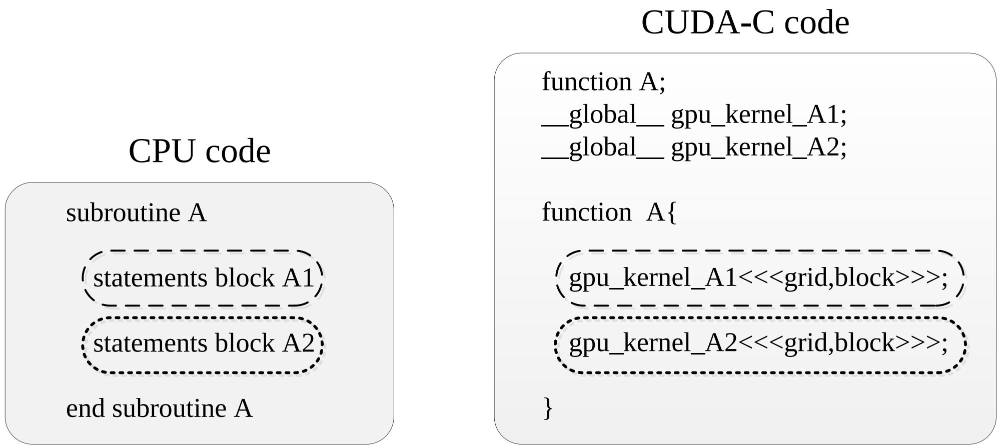
Fig. 2. when a MPI function call is invoked in subroutine A, A is broken into 2 small kernels.

Full Screen / Esc

Printer-friendly Version

Interactive Discussion

Discussion Paper



```
subroutine A
  statements block A1
  statements block A2
end subroutine A
```

```
function A;
__global__ gpu_kernel_A1;
__global__ gpu_kernel_A2;

function  A{
  gpu_kernel_A1<<<grid,block>>>;
  gpu_kernel_A2<<<grid,block>>>;
}
```

Fig. 3. when interior array is first writed by one thread and later read by adjacent threads, where caching this array in shared memory can not benefit, subroutine A is broken into 2 small kernels.