Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes

J. Langguth[a,*], N. Wu[a,b], J. Chai[a,b], X. Cai[a,c]

[a] Simula Research Laboratory, Fornebu, Norway
[b] National University of Defense Technology, Changsha, China
[c] University of Oslo, Oslo, Norway

HIGHLIGHTS

- Multicore and GPU code optimization for finite volume computation.
- Numerical experiments investigating performance relative to irregularity.
- Detailed performance modeling based on CPU and GPU architecture.
- Generalized performance model for identifying bottlenecks in irregular applications.

ARTICLE INFO

Article history:
Received 31 March 2014
Received in revised form
4 October 2014
Accepted 6 October 2014
Available online 16 October 2014

Keywords:
Nvidia K20 GPU
Finite volume method
Unstructured tetrahedral mesh
CUDA programming
OpenMP
Multicore
Performance modeling

ABSTRACT

Finite volume methods are widely used numerical strategies for solving partial differential equations. This paper aims at obtaining a quantitative understanding of the achievable performance of the cell-centered finite volume method on 3D unstructured tetrahedral meshes, using traditional multicore CPUs as well as modern GPUs. By using an optimized implementation and a synthetic connectivity matrix that exhibits a perfect structure of equal-sized blocks lying on the main diagonal, we can closely relate the achievable computing performance to the size of these diagonal blocks. Moreover, we have derived a theoretical model for identifying characteristic levels of the attainable performance as a function of hardware parameters, based on which a realistic upper limit of the performance can be predicted accurately. For real-world tetrahedral meshes, the key to high performance lies in a reordering of the tetrahedra, such that the resulting connectivity matrix resembles a block diagonal form where the optimal size of the blocks depends on the hardware. Numerical experiments confirm that the achieved performance is close to the practically attainable maximum and it reaches 75% of the theoretical upper limit, independent of the actual tetrahedral mesh considered. From this, we develop a general model capable of identifying bottleneck performance of a system's memory hierarchy in irregular applications.

1. Introduction

For any computer program that implements a numerical computation over an unstructured mesh, irregular accesses to the data structures are unavoidable. Such irregular data accesses put significant pressure on the different levels of the memory hierarchy. The common wisdom for CPU programming is thus to strive for good spatial and temporal locality of data on all cache levels. This strategy becomes even more important on GPUs, because the discrepancy between a GPU's global memory bandwidth and its floating-point capabilities is even wider.

Finite volume methods are widely used numerical strategies for solving partial differential equations. Advantages of using finite volumes include built-in support for conservation laws and applicability for unstructured computational meshes. The cell-centered finite volume method is the most common variant, where the degrees of freedom lie in the center of each computational cell. In this paper, we study the most representative 3D scenario where the computational mesh is irregularly made up of tetrahedra which constitute the computational cells. Our objective is to obtain a

* Corresponding author.
E-mail addresses: langguth@simula.no (J. Langguth), nanwu@nudt.edu.cn (N. Wu), chaijun200306@nudt.edu.cn (J. Chai), xingca@simula.no (X. Cai).

http://dx.doi.org/10.1016/j.jpdc.2014.10.005
0743-7315/© 2014 Elsevier Inc. All rights reserved.

quantitative understanding of the impact of spatial and temporal data locality on the speed of computations when executed on the Kepler GPU by Nvidia [19] and on a typical dual-socket HPC node equipped with Intel CPUs.

For an unstructured tetrahedral mesh, the only viable data structure for storing the tetrahedron-center values is a 1D array. There is no universally ideal ordering of the tetrahedra. However, a completely random ordering often means poor computing speed, because of the consequent random jumps back and forth in the 1D array. Another important observation for cell-centered finite volume computations over tetrahedral meshes is that each computational cell (except for the boundary cells) is directly coupled with exactly four neighboring cells. This is so because the coupling from one tetrahedron to its neighboring tetrahedra arises from flux-type computations across the four triangular faces. Without losing generality, it suffices to study the following computation:

$$y(i) = \sum_{j=1}^{4} A(i,j)\,(x(\mathcal{I}(i,j)) - x(i))\,, \tag{1}$$

where x and y denote two 1D arrays that store two sets of tetrahedron-center values. Index i loops from 1 to the total number of tetrahedra n, where $\mathcal{I}(i,\cdot)$ gives the face-to-face connection from tetrahedron i to all its four neighboring tetrahedra. The values of A are weights representing the pairwise tetrahedron–tetrahedron couplings. In terms of data structure, all the entries of A are typically stored in a 2D array, where the first dimension equals the total number of tetrahedra, and the second dimension is four. We also remark that for each tetrahedron, the associated amount of computation is 11 floating-point operations (FLOPS), namely 4 subtractions, 4 multiplications and 3 additions.

In order to perform this computation, 4 entries of A and 5 entries of x must be provided. Throughout this paper, we assume that A is stored in the *ELLpack* matrix format [10]. This means that an additional 4 entries of $\mathcal{I}$ are required. The ELLpack format was shown to work well on GPUs in [2]. Since most tetrahedra have exactly four neighbors, it is more efficient to assume this and use padding if the number of neighbors is lower, rather than to store the number of neighbors explicitly. Furthermore, we assume 64-bit floating point values and 32-bit integers, which implies a computational intensity (i.e. FLOP per byte ratio) of 0.125, since 88 bytes must be read to perform 11 FLOPS. However, as each entry of x is accessed five times – four times by the four neighboring tetrahedra and once by its tetrahedron owner – proper caching can reduce the number of memory accesses on x from 5 to an average of 1 per tetrahedron. This results in an effective computational intensity relative to memory of 0.196, while the computational intensity relative to the cache traffic remains at 0.125. Because data is moved to cache in entire cache lines, the effective intensity can be even lower. We study this effect in Sections 7 and 8.

In addition to reading the data, one y value of 8 bytes must be written back to memory for each tetrahedron, which consumes additional bandwidth on most systems and thus reduces the above intensity to 0.1712 and 0.115 respectively. Depending on the architecture and the size of its cache line, overfetching can occur which in effect further reduces the above values.

In any case, the kernel is severely memory bound and the maximum achievable performance in FLOPS will be far below the theoretical peak performance. In the following, we will investigate the achievable computational intensity relative to the memory traffic and caches on the Nvidia K20 Kepler GPU and on Sandy Bridge Intel CPUs.

The remainder of the paper is organized as follows: in Sections 2 and 3, we describe the hardware and the implementations used for our experiments. Section 4 details the setup of the experiments while Sections 5 and 6 describe the experimental results. Based on these, our performance model for the GPU is detailed in Section 7 and that for the CPU in Section 8. Based on this, we develop a general performance model in Section 9. Finally, we give a brief discussion of related work and our conclusions.

2. GPU platform and implementation

The heart of our GPU test implementation consists of two CUDA kernel functions that compute Eq. (1) in a straightforward manner while using the memory hierarchy efficiently. These are detailed in Figs. 2 and 3 below.

In order to highlight the details, let us first take a look at the architecture and memory hierarchy of the Nvidia GK110 GPU, labeled K20m "Kepler". On each of its 13 streaming multiprocessors, the K20 possesses 48 kB of read-only cache, as well as 64 kB of on-chip storage that is divided between shared memory and level 1 cache (L1). In our experiments 48 kB are assigned to shared memory.

In order to obtain the best performance, it is crucial to optimize the use of this on-chip storage. Accesses to shared memory are fast, but data has to be placed there explicitly. Read-only cache is comparably fast [20] and requires only flagging of variables, but eviction of data from it cannot be controlled by the programmer. In previous generations of Nvidia GPUs, the read-only cache was designated as texture cache and could only be accessed for computation by using unwieldy commands. In CUDA 5.x using devices with compute capability 3.5 however, variables can easily be flagged for read-only caching using the *const __restrict* qualifier, or the *__ldg* instruction. Since *const __restrict* is essentially only a hint for the compiler to use read-only cache, using *__ldg* is preferable for obtaining more predictable performance.

Data that is not fetched into read-only cache or coalesced using shared memory will be accessed by reading 128 bytes at a time, i.e. the SIMD width of the GPU, which leads to significant overfetching since our application loads at most 88 bytes of data per tetrahedron, and only up to 32 contiguous bytes per array. This overfetching leads to severely reduced performance. On the other hand, L2 and read-only cache is read at 32 bytes at a time, and thus does not suffer from this problem.

We therefore avoid using direct access, which leaves us with two viable memory placement strategies. In the first option, A and $\mathcal{I}$ are placed in the shared memory, while read-only cache is used exclusively for caching the x vector. In the second option, all three arrays are cached in read-only cache. Placing parts of the x vector into shared memory is not viable, since the elements that will be accessed are not known beforehand.

Meanwhile, accesses to A and $\mathcal{I}$ are regular which allows explicitly preloading all required elements into shared memory. When implemented in a straightforward manner, this incurs the same overfetching problems as direct accesses. However, because all threads in a thread block can see the same shared memory contents, it is possible to coalesce these accesses, thereby avoiding overfetching. This is achieved by spreading the memory accesses of an entire thread block in such a way that every thread reads one value at a time, irrespective of which thread will actually use the value. See Kernel 1 in Fig. 2 for implementation details.

Together, L1, read-only cache, and shared memory can be thought of as the first level of cache on the K20 GPU. From the technical point of view, it works in a manner substantially different from a CPU, where L1 cache does not need to be managed explicitly. On the other hand, the second level of cache (L2) is rather similar. The K20m has 1280 kB of L2 cache which is shared among the 13 multiprocessors. All accesses to and from the GPU's global memory are cached via L2 automatically. Thus, its function is similar to the shared third level cache on current multicore CPUs. Accesses to