Applied Mathematics & Information Sciences 5 (2) (2011), 129S-138S
– An International Journal

Data Layout Pruning on GPU

Xinbiao Gan, Zhiying Wang, Li Shen and Qi Zhu

School of Computer, National University of Defense Technology, Changsha 410073, China
xinbiaogan@163.com

This work is based on NVIDIA GTX 280 using CUDA (Computing Unified Device Architecture). We classify Dataset to be transferred into CUDA memory hierarchy into SW (shared and must write) and SR (shared but only read), and existing memory spaces (including shared memory, constant memory, texture memory and global memory) supported on CUDA-enabled GPU memory hierarchy are adopted to probe into best memory space for specified dataset. Conclusions from experimental results are that shared memory is proposed for SW; constant memory is advisable for SR and texture memory for SR with structured-grid dataset, especially for 2D, 3D regular grid.

Keywords: memory optimization, data layout pruning, GPU.

1. Introduction

GPU architecture provides high peak performance but maximal performance available is bound in data layout on memory hierarchy. So, best memory optimization on GPU is that programmer must orchestrate data locality and GPU memory architecture, which would increase GPU programmer's workloads to decide which memory space is best for specified dataset. This paper is pruning data layout using CUDA benchmarks to guide programming efficient GPU programs with ease.

Graphics Processing Units (GPUs) Computing era is coming due to massive highly parallel computing resources1. However, how to make best use of GPUs' powerful computing capacity is still a challenge. In reality, modern GPU is a memory-bound architecture. Therefore, understanding memory hierarchy on GPU and then deploy particular dataset into specified memory space is very important for maximizing performance. There are many works focusing on memory optimization to improve performance.

Ryoo and Hong both emphasized importance of data layout and memory optimization from performance evaluation [2-3]. Differently, Ryoo focused on generalizing optimization principles and strategies [2], while Hong highlighted MLP (Memory-Level parallelism) to maximizing memory bandwidth [3].

Siegel proposed loop unrolling and memory access patterns for Gravit Simulator on GPU [4]. His contributions focused on shared memory and coalesced access using AoS (Array of

Structure) rather than SoA (Structure of Array) for Gravit Simulator. Our data layout pruning is benchmarking CUDA application suites to acquire architectural experiences on memory hierarchy supported on CUDA-enabled GPU for general computing-intensive applications.

Sung developed a data layout transformation methodology that can speed up structured-grid codes on GPUs [5]. But, above solution cannot address holistic data layout transformations, especially for irregular data structures. While our solution focus on holistic data layout including regular data and irregular data to explore data locality.

Previous works either emphasize the importance of memory optimization or present optimization techniques to avoid non-coalesced access and access conflicts. The question this work address is that which memory space is the best residence for particular dataset. The followings are the main contributions of this paper.

1. A modification and implementation of CUDA benchmark application suites using different memory space model including shared memory, constant memory, texture memory and global memory, respectively.

2. Classify dataset processed on GPU into SW (shared and must write) and SR (shared but only read).

3. Suggestions concluded from experiments on CUDA benchmark suites include that: (1) shared memory is the best residence proposed for SW; (2) constant memory is advisable for SR and texture memory for SR with structured-grid dataset.

The rest of this paper is organized as follows: CUDA-enabled GPU architecture, programming model and memory hierarchy on CUDA are introduced in Section 2. Section 3 proposes methodology to pruning data layout and presents benchmarks. Section 4 discusses experimental results for data layout pruning on CUDA benchmark suites. Conclusions and suggestions for future work are concluded in Section 5.

2. Architecture

Each CUDA-enabled GPU is a large set of processor cores with the ability to directly address a global memory. This makes it easier for developers to implement data-parallel kernels using a more general and flexible programming model than previous GPUs. Recently, there have been several new programming languages that aim to program GPU such as Brook+, CUDA and OpenCL [3].

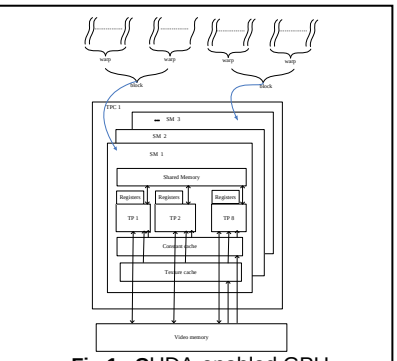

Fig 1 CUDA-enabled GPU architecture

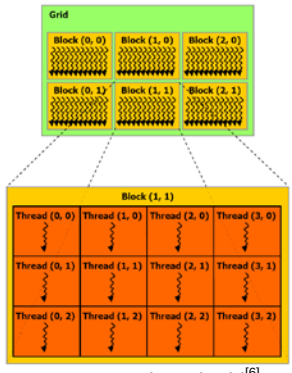

Fig 2 CUDA thread grid[6]

2.1. GPU Architecture and Execution Model

Figure 1 depicts architecture of NVIDIA GTX 280, which consists of 10 TPCs (Threading Processor Cluster), each containing 3 SMs (Streaming Multiprocessors). Each SM has 8 TPs (Threading Processor), and has 16KB shared memory [6].

During execution, thousands of parallel threads are running in SPMD (Single Program Multiple Data) model, in which threads are organized in thread blocks; threads within a block are organized into warps of 32 threads. Each warp executes in SIMD fashion, issuing in 4 cycles on eight TPs in one SM.

2.2. CUDA Programming Model

CUDA programming model is ANSI C extended by several keywords and constructs. The developer supplies a single source program encompassing both host (CPU) and device (GPU) code. The source program would be separated and compiled, in which device code compiled to CUDA device instruction set and it becomes a parallelized new program that is termed “kernel”. The kernel is downloaded into GPU device that acts as a coprocessor to the host (CPU). It is executed in thousands of “threads”; and threads are organized in thread block; thread blocks that execute the same kernel can be batched together to form a grid of blocks, as figure 2.

Threads within a thread block can co-work with each other through shared memory and can synchronize their execution using keyword “syncthreads” to coordinate their memory access. But, threads in different thread blocks are unable to access same shared memory and they run independently.

Table 1 Property of CUDA Memory Spaces

Memory Space	Permissions	Scope	Capacity	Latency	Special Features
Global	Read/Write	All threads	DRAM capacity	High	Requires aligned, contiguous simultaneous accesses for best bandwidth.
Texture	Read-Only	All threads	Up to DRAM capacity	High but cached	Hardware interpolation, indexed by real-valued indexes, and other features for image processing.
Constant	Read-Only	All threads	64KB	Low and cached	Single-banked cache with broadcast capability to multiple threads.
Local	Read/Write	Single thread	DRAM capacity	High	Space for register spilling, most often promoted to private registers, which are shared between threads. Values not promoted to registers have long latency access.
Shared	Read/Write	Thread-block	16KB	Low	Local scratchpad memory that can be shared among threads in a thread block. Organized into 16 banks. More shared memory used per thread block means fewer thread blocks can be simultaneously active

Understanding CUDA memory hierarchy is a prerequisite for data layout pruning on CUDA-enabled GPU before benchmarking CUDA benchmark suites under the guidance of comprehensive methodology.

CUDA threads may access data from multiple memory spaces during their execution such as register, shared memory, constant memory, texture memory and global memory or video memory, but different memory space has different property and special purpose, as shown in table 1. So that data layout pruning among memory hierarchy to hunt for best memory space for specified dataset is to accelerate applications on GPU with ease.

3.1. Benchmarking Suites

The following benchmarking applications are selected from Parboil benchmarking suite [7].

◆ *Coulombic Potential*

Coulombic Potential (CP) is computing electric potential in a volume containing point charges based on direct Coulomb summation, in which a rectangular lattice is defined around the atoms with specified boundary padding, and fixed lattice spacing is used in all three dimensions.

For each lattice point i located at position r_i, the coulomb potential V_i is given in equation (1).

$$V_i = \sum_j \frac{q_j}{4\pi\varepsilon_0\varepsilon\left(r_{ij}\right)r_{ij}} \tag{1}$$

With the sum taken over the atoms, where atom j is located at r_j and has partial charge q_j,

and the pair-wise distance is $r_{ij} = | r_j - r_i |$. The function $\varepsilon(r)$ is a distance-dependent dielectric coefficient, and in the present work will always be defined as either $\varepsilon(r) = \kappa$ or $\varepsilon(r) = r\kappa$, with κ constant. For a system of N atoms and a lattice consisting of M points, the time complexity of the direct summation is $O(MN)$. Actually, the potential map is easily decomposed into planes or slices, which can translate conveniently to a CUDA grid and can be independently computed in parallel [8].

◆ *Magnetic Resonance Image FHD*

Magnetic Resonance Imaging FHD (MRI-FHD) is an advanced reconstruction algorithm, which operates directly on non-uniformly sampled data and uses the least-squares optimality criterion. The algorithm uses an iterative linear solver to solve equation (2).

$$F^H F \rho = F^H d \qquad (2)$$

$$Q(x_n) = \sum_{m=1}^{M} | \phi(k_m) |^2 e^{(i2\pi k_m x_n)} \qquad (3)$$

Where ρ is the desired image, $F^H F$ is a matrix that depends only on the scan trajectory, and $F^H d$ is a vector that depends both on the scan trajectory and the acquired data. Element (j,k) of $F^H F$ is defined as $Q(x_j - x_k)$, where $Q(x)$ is given by equation (3) [9], which is the convolution kernel that facilitates multiplication operations involving $F^H F$, and $\phi(x)$ is the Fourier transform of the voxel basis function. There is M k-space sampling locations, with k_m denoting the location of the m^{th} sample. Likewise, there are N voxel coordinates, with x_n denoting the coordinates of the n^{th} voxel. Because Q depends only on the scan trajectory (not the scan data) and the size of the image, it can be computed before the scan occurs and can be reused during any reconstruction that shares the same scan trajectory and image size. Second, the algorithm computes the vector $F^H d$, which is defined as equation (4) [10].

$$\left[F^H d\right]_n = \sum_{m=1}^{M} \phi^*(k_m) d(k_m) e^{(i2\pi k_m x_n)} \qquad (4)$$

◆ *Rys Polynomial Equation Solver*

Rys Polynomial Equation Solver (RPES) is to calculate 2-electron repulsion integrals which represent the Coulomb interaction between electrons in molecules.
The electron repulsion integrals be evaluated exactly by an N-point numerical Quadrature formula defined as equation (5)[1-12]

$$(\eta_i \eta_j \square \eta_k \eta_l) = 2(\rho/\pi)^{\frac{1}{2}} \times \sum_{\alpha=1}^{N} I_x(u_\alpha) I_y(u_\alpha) I_z(u_\alpha) W_\alpha \qquad (5)$$

Where W_α is a weight factor; I factors may be used for many different integrals. And

efficient recursion formulas are given for calculation I by a two-step process involving the "transfer" equations as follows.

$$I_x\left(n_i, n_j, m, 0\right) = I_x\left(n_i + 1, n_j - 1, m, 0\right) + \left(x_i - x_j\right) I_x\left(n_i, n_j - 1, m, 0\right) \tag{6}$$

$$I_x\left(n_i, n_j, n_k, n_l\right) = I_x\left(n_i, n_j, n_k + 1, n_l - 1\right) + \left(x_i - x_j\right) I_x\left(n_i, n_j, n_k, n_l - 1\right) \tag{7}$$

◆ *Two Point Angular Correlation Function*

The Two Point Angular Correlation Function (TPACF) is a mathematical equation, which is used as a way to measure the probability of finding an astronomical body at a given angular distance from another astronomical body.

$$w(\theta) = \frac{DD(\theta) - 2\frac{\sum DR(\theta)}{N} + \frac{\sum RR(\theta)}{N}}{\frac{\sum RR(\theta)}{N}} \tag{8}$$

The TPACF is calculated according to the equation (8). The three most computationally intensive parts of the equation are the computations of the correlation functions: DD, DR, and RR. DD is the auto-correlation of the actual dataset. DR is the cross-correlation of the actual data set to a given random set. RR is the auto-correlation of the random sets. Note that there will be only one DD to calculate, but there will be multiple DRs and RRs to calculate. There will be as many DRs and RRs as there are random sets used. Each of the correlation functions consists of a series of dot products between coordinate pairs [13]. Each of these calculations is an independent floating point operation that can be done in parallel. So that TPACF is suitable for accelerating on CUDA-enabled GPU.

3.2. Methodology

To prune data layout, inputs for above benchmarking applications are purposefully deployed into different memory spaces detailed in table 1, respectively. The data layout pruning process is shown in figure 3.

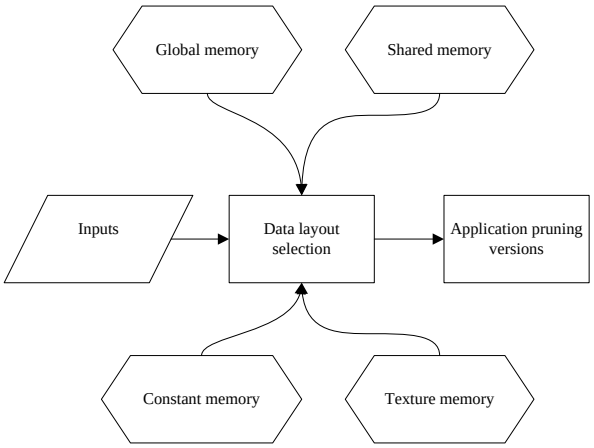

Fig 3 Data layout pruning strategy

As shown in figure 3, memory spaces including global memory, shared memory, constant memory and texture memory supported in CUDA-enabled GPU are configured to probe into best memory space for specified dataset.

4. Experiments and Analysis

In this section, we will benchmark above applications to prune data layout on NVIDIA GTX280 with Intel Core2 Quad 2.33 GHz CPU and 4GB main memory using Visual Studio 2005 + CUDA toolkit and SDK 3.1 with NVIDIA Driver version number 257.21 for Microsoft Windows XP.

4.1. Experiments

In order to keep testing suites running at the same configuration, data layout pruning only re-deploys inputs into different memory spaces without any modifications or optimizations on benchmark suites. Separate experiments have shown that deploying inputs (input data) into different memory space has major impact on application performance.

Figure 4, Figure 5, Figure 6, and Figure 7 show pruning results on benchmarking applications using different memory spaces model including global memory, constant memory and texture memory. In order to avoid introducing other optimization techniques, inputs for CP are not deployed into shared memory because inputs should be partitioned before placing into shared memory; inputs for RPES are not deployed into texture memory because inputs need update.

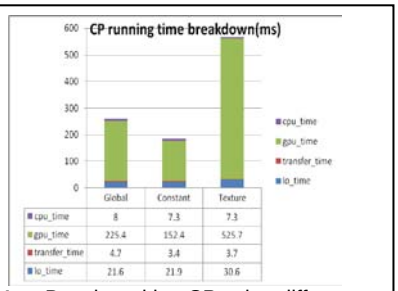

Fig 4 Benchmarking CP using different memory spaces

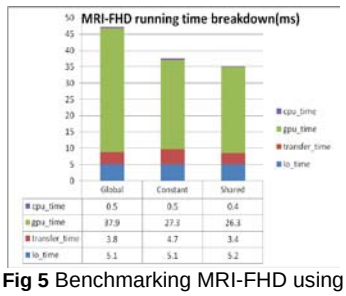

Fig 5 Benchmarking MRI-FHD using different memory spaces

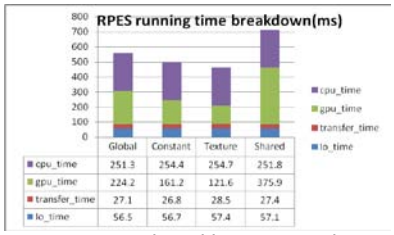

Fig 6 Benchmarking RPES using different memory spaces

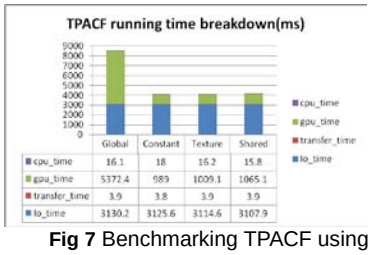

Fig 7 Benchmarking TPACF using different memory spaces

4.2. Analysis

To summarize, above separate experiments all have shown that deploying inputs (input data) into different memory space has major impact on application performance, especially for the kernel time (gpu_time).

Figure 8 shows the speedup of CUDA benchmark suites using different memory model, in which performance under global memory is normalized as baseline. When speedup is zero it denotes that such memory model is unavailable to testing suite. So, there are no shared memory model for CP and no texture memory model for MRI-FHD.

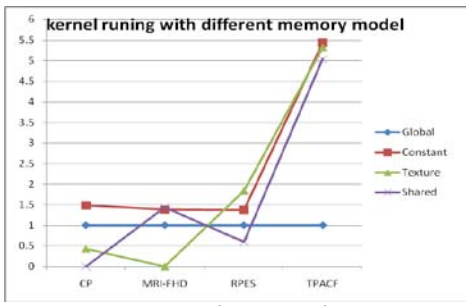

Fig 8 Data layout pruning

It is obvious that deploying inputs into different memory space has major impact on application kernels; an appropriate memory model can attain up to 5.5X speedup. From

Figure 8, we can also find that not all special memory space including shared memory, constant memory and texture memory can outperform the global memory. Especially, CP using texture memory is worse than using global memory because of high texture cache miss. However, texture memory can outperform global memory on other testing suites; RPES using shared memory is also worse than using global memory because of bank conflict. However, shared memory can outperform global memory on other testing suites. Obviously, constant memory space always outperforms global memory but it is read-only.

Accordingly, re-deploying inputs into special memory space seamlessly can attain remarkable improvements but they all have limits detailed in table 1. In CUDA kernel, data are all shared by threads. For example, built-in data type variables are duplicated in register affinity to each thread; memory spaces for structured variables such as array space are shared by all thread. Therefore, dataset structures for GPU applications are classified into SW and SR to make best use of CUDA memory hierarchy. Furthermore, dataset for above applications are analyzed and suggestions concluded that shared memory is proposed for SW; constant memory is advisable for SR and texture memory for SR with structured-grid dataset, especially for 2D, 3D regular data structures.

5. Conclusions and Future work

In this work, selective CUDA benchmark suites are tested using different memory spaces on CUDA-enabled GPU NVIDIA GTX 280 for probing into best memory space for special dataset structure. In order to make best use of CUDA memory hierarchy, dataset structures for GPU applications are classified into SW and SR. Furthermore, suggestions and conclusions from experiments is that shared memory is proposed for SW; constant memory is advisable for SR and texture memory for SR with structured-grid dataset, especially for 2D, 3D regular grid. We are currently performing research on automated data layout pruning on extension of CUDA (xCUDA), which would transform an OpenMP-like explicit parallel source program into a standard CUDA program with memory optimizations including data layout pruning.

Acknowledgements : We would like to thank the IMPACT Research Group Research Group at the University of Illinois at Urbana-Champaign for the Parboil Benchmark suite they released online. This work is supported by the National Grand Fundamental Research Foundation of China under Grant No. 2007CB310901, the National Natural Science Foundation of China under Grant No. 60803041, and the Innovation Program for Excellent

Graduates Foundation of national University of Defense Technology of China under Grant No. B090603.

References

[1] J. Nickolls, William J. Dally. The GPU Computing era. IEEE Micro, 2010:31(2), pp. 1–10.
[2] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, et al. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In PPoPP, pp. 73–82, 2008.
[3] S. Hong, H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In ISCA, pp. 152–163, 2009.
[4] J. Siegel, J. Ributzka, X.M. Li. CUDA Memory Optimizations for Large Data-Structures in the Gravit Simulator. Proceedings of the International Conference on Parallel Processing Workshops, pp.174–181, 2009.
[5] I-Jui Sung, Wen-Mei Hwu. Data Layout Transformation for Structured-Grid Codes on GPU. Proceedings of the Workshop on Language, Compiler, and Architecture Support for GPGPU, pp.85–94, 2010.
[6] NVIDIA Corporation. CUDA Programming Guide 2.0, July 2008.
[7] IMPACT Research Group. Parboil benchmark suite. http://impact.crhc.illinois.edu/ parboil. php.
[8] J. E. Stone, J. C. Phillips, P. L. Freddolino, et al. Accelerating molecular modeling applications with graphics processors. Journal of Computational Chemistry, 2007:28(16), pp. 2618 – 2640.
[9] S. S. Stone, H. Yi, J. P. Haldar, et al. How GPUs Can Improve the Quality of Magnetic Resonance Imaging. Proceedings of the 5th International Conference on Computing Frontiers, pp. 1–9, 2008.
[10] S. S. Stone, J. P. Haldar, S. C. Tsao, et al. Accelerating Advanced MRI Reconstructions on GPUs. Proceedings of the Sixth International Conference on Computing Frontiers, pp. 1283–1286, 2009.
[11] J. Rys, M. Dupuis, H. F. King. Computation of Electron Repulsion Integrals Using the Rys Quadrature Method. Journal of Computational Chemistry, 1983:4(2), pp. 154–157.
[12] M. Dupuis and A. Marquez. The Rys Quadrature revisited: A novel formulation for the efficient computation of electron repulsion integrals over Gaussian functions. Journal of Chemical Physics, 2001:114(5), pp.2067–2078.
[13] V. V. Kindratenko, A. D. Myers, R. J. Brunner. Implementations of the two-point angular correlation function on a high-performance reconfigurable computer. Journal of Scientific Programming, 2009:17(3), pp. 247–259.

Xinbiao Gan received MS degree in computer system architecture from National University of Defense Technology of China in 2008. He is currently pursuing PhD degree in computer system architecture from National University of Defense Technology of China, His research interests are in the areas of computer architecture, High performance Computing, Compiler and optimization.