Autotuning BLAS for GPUs: Matrix Multiplication for Fermi and Kepler

Jakub Kurzak
Piotr Luszczek
Stan Tomov
Jack Dongarra

VECPAR 2012
10th International Meeting on High-Performance Computing for Computational Science, Kobe, Japan, July 17-20, 2012

http://web.eecs.utk.edu/~kurzak/presentations/
Overview

- Motivation for an academic GPU autotuning software (BLAS and beyond)
- Development of tunable CUDA GEMM stencil (and search space generation)
- Pruning of the search space
- Fermi architecture vs Kepler architecture (GTX 680)
- Results on Fermi
- Results on Kepler (GTX 680)
Motivation

vendors autotune

Cray Fermi ZGEMM autotuning sweep

272 auto-tuned kernels

no complex pruning
slow kernels tested

272 samples
Motivation
academics autotune

UTK Fermi ZGEMM autotuning sweep

- powerful pruning
- fast kernels tested

211 samples
Motivation
academically sustainable development
Motivation

special cases

*not the main focus of optimization efforts in commercial packages*

“Misshaped” matrices

The GEMM routine is often optimized for the case where all inputs are large square matrices. In dense linear algebra packages (e.g. MAGMA), in most cases, at least one dimension is small.

Transposed inputs

The GEMM routine is often optimized for the $A \times B$ case (LINPACK). In dense linear algebra packages (e.g. MAGMA), in many cases, at least one input is transposed.

Complex arithmetic

The double precision DGEMM is always the best optimized one due to the LINPACK Benchmark. Many real scientific applications actually require the complex ZGEMM.
Motivation
“exotic” arithmetic

arithmetic not typically supported in commercial packages

Quadruple precision

D. Mukunoki, D. Takahashi
Implementation and Evaluation of Triple Precision BLAS Subroutines on GPUs
26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW 2012),
13th Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-12), pp. 1372–1380, Shanghai, China, 2012

D. Mukunoki, D. Takahashi
Implementation and Evaluation of Quadruple Precision BLAS Functions on GPUs
State of the Art in Scientific and Parallel Computing, Reykjavík, Iceland, 2010
DOI: 10.1007/978-3-642-28151-8_25

N. Nakasato
A fast GEMM implementation on the cypress GPU
ACM SIGMETRICS Performance Evaluation Review,
Special Issue on the 1st International Workshop on Performance Modeling,
DOI: 10.1145/1964218.1964227

Interval arithmetic

S. Collange, J. Flórez, D. Defour
Interval Arithmetic in CUDA
GPU Computing GEMS Jade Edition
Applications of GPU Computing Series, Morgan Kaufmann, 2011
ISBN: 978-0123859631

similar methodology
high computational intensity
Motivation
unorthodox data layout

data layout not typically supported in commercial packages

Tile Matrix Layout

F. Gustavson, L. Karlsson, B. Kågström,
Parallel and Cache-Efficient In-Place Matrix Storage Format Conversion,
DOI: 10.1145/2168773.2168775

J. Kurzak, P. Luszczek, M. Faverge, J. Dongarra
Programming the LU Factorization for a Multicore System with Accelerators
International Meeting on High-Performance Computing for Computational Science, Kobe, Japan, 2012
http://nkl.cc.u-tokyo.ac.jp/VECPAR2012/

tile layout on the CPU (CCRB)
tile layout on the GPU (CRRB)
Motivation

development complexity

- Thread divergence
- Warp serialization
- Cache line access
- Partition camping

- Occupancy
- Register pressure
- Shared memory pressure
- Device memory pressure
Motivation
optimization complexity

- no access to proprietary low-level hardware specs
  - no access to assembly (PTX is the lowest level)
  - proprietary warp scheduling, thread block scheduling, etc.

- always development and optimization of parallel code
  (no serial optimization followed by parallelization)

- many tuning parameters are discrete
  (e.g. number of threads should be divisible by the warp size)
BLAS GEMM

target workload

\[ C = \alpha A \times B + \beta C \]

```c
void cblas_xgemm (  
    const enum CBLAS_ORDER Order,  
    const enum CBLAS_TRANSPOSE TransA,  
    const enum CBLAS_TRANSPOSE TransB,  
    const int M,  
    const int N,  
    const int K,  
    const SCALAR alpha,  
    const TYPE *A,  
    const int lda,  
    const TYPE *B,  
    const int ldb,  
    const SCALAR beta,  
    TYPE *C,  
    const int ldc)
```
BLAS GEMM
canonical form

FOR each element
FOR each element
FOR each element

\[ C = \alpha A \times B + \beta C \]
**BLAS GEMM**

loop tiling

FOR each tile
  FOR each tile
    FOR each tile
      FOR each element
        FOR each element
          FOR each element

  $C = \alpha A \times B + \beta C$
BLAS GEMM
loop unrolling

FOR each tile
  FOR each tile
    FOR each tile
      FOR each element
        FOR each element
          \[ C = \alpha A \times B + \beta C \]

canonical technique
BLAS GEMM
device level

\[ \begin{align*}
\text{A} & : M_{\text{dev}} \\
\text{B} & : K_{\text{dev}} \\
\text{C} & : N_{\text{dev}}
\end{align*} \]

completely parametrized
CUDA GEMM
flow of data through the memory system

Read-only data pushed through texture caches does not pollute L2 and L1 caches.

This is not really a cache, but more of a vector register file allowing for permutations of vector elements.
BLAS GEMM
doctor buffering

canonical technique
double buffered GEMM loop
CUDA GEMM
shared memory skewing

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CUDA GEMM
universal stencil

- All types of input (BLAS standard)
  - single / double
  - real / complex
  - A: NoTrans / Trans / ConjTrans
  - B: NoTrans / Trans / ConjTrans

- Three types of memory reads
  - no textures / 1D textures / 2D textures

- $2 \times (4 + 9) \times 3 = 78$ cases total
CUDA GEMM
universal stencil

- 9 tuning parameters
  - M, N, K – tiling of A, B, C
  - $X_A, Y_A$ – shape of thread grid for reading A
  - $X_B, Y_B$ – shape of thread grid for reading B
  - $X_C, Y_C$ – shape of thread grid for computing C

- Basically $256^9$ search space, ca. $10^{21}$ (sextillion)
Pruning hardware specs can be queried no need for complex and time consuming probing.

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>Compute Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a grid of thread blocks</td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>8</td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>768</td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
</tr>
<tr>
<td>Maximum amount of shared memory</td>
<td>16 KB</td>
</tr>
</tbody>
</table>
Pruning
rigid hardware constraints

represent parameter configurations that cause the code to exceed hardware limitations

Examples
- creating more threads / warps / thread blocks than supported
- allocating more shared memory than available

Results of violation
- failure to compile
- failure to launch during runtime

Every kernel that passes through pruning should successfully compile and launch.
Pruning
rigid software constraints

represent parameter configurations that render the implementation invalid

Examples
- not supported corner cases
- not supported problem dimensions

Results of violation
- division by zero
- out-of-bound memory access
- crashing, hanging, incorrect results

Accelerator is not a stand-alone device by definition. Therefore, only offload what can go fast. Leave the rest to the host processor.

Every kernel that passes through pruning should be free of such problems.
Pruning
flexible hardware constraints

represent parameter configurations that are bound to render the code slow due to the characteristics of the hardware (due to wasted hardware resources)

Examples
- low occupancy
- partially filled warps
- partially utilized cache lines

Results of violation
- idle threads
- wasted memory bandwidth

Every kernel that passes through pruning should be free of major performance blunders from the hardware utilization perspective.
Pruning flexible software constraints

represent parameter configurations that are bound to render the code slow due to the nature of the implementation (they are algorithm and implementation dependent)

Examples
- high register pressure, cache pressure, memory pressure
- too much branching / loop overhead

Results of violation
- register, cache, memory starvation
- poor instruction scheduling

Every kernel that passes through pruning should be free of major performance blunders from the software perspective.
Occupancy Pruning
occupancy limited by register usage

\[
\begin{align*}
\text{regs}_\text{per}_\text{thread} &= \text{estimated usage in the innermost (completely unrolled) loop} \\
\text{regs}_\text{per}_\text{block} &= \text{regs}_\text{per}_\text{thread} \times \text{threads}_\text{per}_\text{block} \\
\text{blocks}_\text{per}_\text{sm} &= \frac{\text{TOTAL}_\text{REGS}_\text{PER}_\text{SM}}{\text{regs}_\text{per}_\text{block}} \\
\text{blocks}_\text{per}_\text{sm} &> \text{MAX}_\text{BLOCKS}_\text{PER}_\text{SM} ? \\
\text{warps}_\text{per}_\text{sm} &> \text{MAX}_\text{WARPS}_\text{PER}_\text{SM} ? \\
\text{threads}_\text{per}_\text{sm} &= \text{blocks}_\text{per}_\text{sm} \times \text{threads}_\text{per}_\text{block}
\end{align*}
\]
Occupancy Pruning
occupancy limited by shared memory usage

\[
\text{shmem\_per\_block} = \text{as declared in the kernel body}
\]

\[
\text{blocks\_per\_sm} = \frac{\text{TOTAL\_SHMEM\_PER\_SM}}{\text{shmem\_per\_block}}
\]

\[
\begin{align*}
&\text{blocks\_per\_sm} > \text{MAX\_BLOCKS\_PER\_SM} \quad ? \\
&\text{warps\_per\_sm} > \text{MAX\_WARPS\_PER\_SM} \quad ? \\
\end{align*}
\]

\[
\text{threads\_per\_sm} = \text{blocks\_per\_sm} \times \text{threads\_per\_block}
\]
Pruning
Fermi ZGEMM

- no constraints
  - 10^{21}
- hard constraints
  - 99,519
- soft constraints
  - occupancy \geq \frac{1}{3} (512)
  - 492
- soft constraints
  - max 2 loads / FMA
  - min 2 blocks / SM
  - 211

300 GFLOPS → 340 GFLOPS
Fermi vs Kepler
Kepler GK104 (GTX680)

16 SMs, 32 CUDA cores each

8 SMXs, 192 CUDA cores each
### Fermi vs Kepler
**Kepler GK104 (GTX680)**

<table>
<thead>
<tr>
<th></th>
<th>GF100(Fermi)</th>
<th>GK104(Kepler)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTIPROCESSORS</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>CUDA CORES PER MULTIPROCESSOR</td>
<td>32</td>
<td>192</td>
</tr>
<tr>
<td>CUDA CORES TOTAL</td>
<td>512</td>
<td>1536</td>
</tr>
<tr>
<td>CLOCK RATE</td>
<td>1544 MHz</td>
<td>1006 MHz</td>
</tr>
<tr>
<td>SINGLE PRECISION FLOATING POINT PEAK</td>
<td>1581 GFLOPS</td>
<td>3090 GFLOPS</td>
</tr>
<tr>
<td>REGISTERS PER MULTIPROCESSOR</td>
<td>32 K</td>
<td>64 K</td>
</tr>
<tr>
<td>SHARED MEM / L1 PER MULTIPROCESSOR</td>
<td>64 K</td>
<td>64 K</td>
</tr>
<tr>
<td>L2 CACHE SIZE</td>
<td>768 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>L2 CACHE BANDWIDTH</td>
<td>384 B per clock</td>
<td>512 B per clock</td>
</tr>
<tr>
<td>MEMORY BANDWIDTH PEAK</td>
<td>192 GBPS</td>
<td>192 GBPS</td>
</tr>
<tr>
<td>TRANSISTORS COUNT</td>
<td>3.0 B</td>
<td>3.5 B</td>
</tr>
<tr>
<td>THERMAL DESIGN POWER</td>
<td>244 W</td>
<td>195 W</td>
</tr>
</tbody>
</table>
## Fermi vs Kepler

**Kepler GK104 (GTX680)**

<table>
<thead>
<tr>
<th></th>
<th>SM (Fermi)</th>
<th>SMX (Kepler)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA CORES</td>
<td>32</td>
<td>192</td>
<td>6</td>
</tr>
<tr>
<td>REGISTERS</td>
<td>32 K</td>
<td>64 K</td>
<td>2</td>
</tr>
<tr>
<td>SHARED MEMORY / L1</td>
<td>64 K</td>
<td>64 K</td>
<td>1</td>
</tr>
</tbody>
</table>
Fermi GEMMs

number of kernels passing the pruning phase

<table>
<thead>
<tr>
<th>GEMM</th>
<th>$A \times B$</th>
<th>$A \times B^T$</th>
<th>$A^T \times B$</th>
<th>$A^T \times B^T$</th>
<th>$A^T \times B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEMM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CGEMM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DGEMM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZGEMM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fermi GEMMs
performance for all cases

- CGEMM: 800 GFLOPS
- SGEMM: 650 GFLOPS
- ZGEMM: 340 GFLOPS
- DGEMM: 300 GFLOPS
GTX680 GEMMs

number of kernels passing the pruning phase

![Graph showing the number of kernels passing the pruning phase for different GEMM operations: SGEMM, CGEMM, DGEMM, and ZGEMM. The x-axis represents the type of operation, and the y-axis represents the number of kernels. The operations are color-coded as follows: A × B (light yellow), A × B^T (green), A^T × B_T (light green), and A^T × B (dark green).]
GTX680 DGEMM
ASTRA vs CUBLAS

CUDA SDK 4.2.6
(April 2012)
GTX680 ZGEMM
ASTRA vs CUBLAS

<table>
<thead>
<tr>
<th></th>
<th>CUBLAS</th>
<th>ASTRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A × B</td>
<td>124</td>
<td>127</td>
</tr>
<tr>
<td>A × Bᵀ</td>
<td>123</td>
<td>127</td>
</tr>
<tr>
<td>Aᵀ × Bᵀ</td>
<td>122</td>
<td>127</td>
</tr>
<tr>
<td>Aᵀ × B</td>
<td>124</td>
<td>128</td>
</tr>
</tbody>
</table>

CUDA SDK 4.2.6
(April 2012)
GTX680 SGEMM
ASTRA vs CUBLAS

CUBLAS
ASTRA
1101  1125
1254  1151
  896  1097
1230  1146

CUDA SDK 4.2.6
(April 2012)
GTX680 CGEMM
ASTRA vs CUBLAS

![Bar chart comparing A × B, A × B^T, A^T × B^T, A^T × B for ASTRA and CUBLAS.

- A × B: ASTRA: 1383, CUBLAS: 1693
- A × B^T: ASTRA: 1419, CUBLAS: 1687
- A^T × B^T: ASTRA: 1332, CUBLAS: 1725
- A^T × B: ASTRA: 1090, CUBLAS: 1663

CUDA SDK 4.2.6 (April 2012)
ASTRA
Automatic Stencil TuneR for Accelerators

Diagram:
- Stencil
- Constraints
- Software Abstraction Layer
- Hardware Abstraction Layer
- Generation & Pruning Engine
- Top Performers Database
- Search Coverage Map
- Distributed Timing Engine
- Tuned Kernel for a Specific Accelerator
ASTRA
making a case for massively parallel tuning

The TSUBAME 2.0 supercomputer at the Tokyo Institute of Technology includes 4,224 Fermi GPUs. By using 365 GPUs for 24 hours, an autotuning run can be done that would take a year on a single GPU; 365 GPUs are less than 9% of the machine.

In order to test the speed of the TSUBAME 2.0 machine (and determine its TOP500 ranking) an HPL run was made that took 144 minutes. If that time was used for an autotuning run, one could tune in parallel the equivalent of 422 days of tuning on a single GPU.
どうもありがとうございます。
質問ありますか？