High Performance Matrix-matrix Multiplication of Very Small Matrices

Ian Masliah , Marc Baboulin, ICL people

University Paris-Sud - LRI

Sparse Days
Cerfacs, Toulouse, 1/07/2016

universite
PARIS-SACLAY

université PARIS-SACLAY

Context

Tensor Contractions

- High-order FEM hydrodynamics code from BLAST
- Tensor contractions results in batch of small matrix-matrix products

Contraction by the first index :

$C_{i_1,i_2,i_3} = \sum_{k_1} A_{k_1,i_1} B_{k_1,i_2,i_3}$

Can be written as :

$Reshape(C)^{nd_1 \times (nd_2 nd_3)} = A^T \; Reshape(B)^{nq_1 \times (nd_2 nd_3)}$.

Context

Use modern C++11

- Constexpr
- Integral constants
- Variadic templates

Write efficient code

- Use hardware anaylsis tools : PAPI, Intel memory checker , CUPTI...
- Make sure the algorithm fits the hardware - multicore or GPUs

C++ features

auto : automatic type inference

```
auto x = 5;
```

Constexpr : compile time functions and variables

```
constexpr int factorial(int n) { return n<2 ? 1 : n*factorial(n-1); }
```

Integral constant : Represent Integers as types

```
typedef std::integral_constant<int, 2> two_t;
```

Variadic templates : variable parameter size functions

```
template<typename... Values> class tuple;
```

Modeling a problem for a multicore

Register Data Reuse and Locality

Data Access Optimizations and Loop Transformation Techniques

Effect of the Multi-threading

Effect of the NUMA-socket and Memory Location

universite PARIS-SACLAY

Register Data Reuse and Locality

We use a Intel Xeon Processor E5-2650 v3 - 10 cores

- Supports AVX-2 - 256 bit SIMD -16 Registers
- Measured bandwidth : 44 GB/s

We also ran test on a ARM Cortex A57 (Tegra X1) - 4 cores

- Supports NEON advanced SIMD - 128 bit SIMD - 32 Registers
- Measured bandwidth : 13 GB/s

A modern CPU has 32KB of L1 cache

- Matrix-Matrix products of up to 32... fit in L1 cache

Data Access Optimizations and Loop Transformation Techniques

We focus on reusing data as much as possible

- Minimize the number of Store and Load operations

Prefetch parts of B and reuse them

- Before starting the matrix-matrix product

Unrolling inner loops

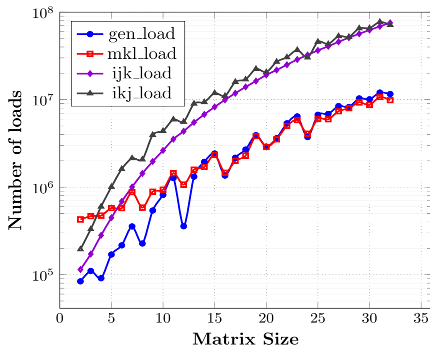
- Better control on memory access patterns

Algorithm 1 Generic matrix-matrix product applied to matrices of size 16×16

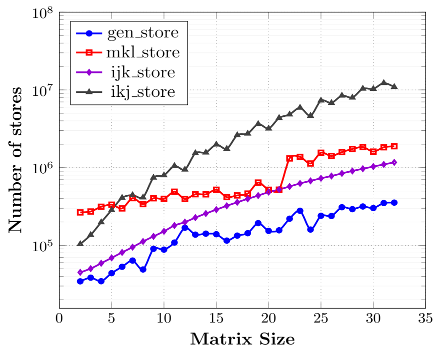
```
1:  Load B0, B1, B2, B3
2:  Load α, β
3:  S = 16
4:  for i = 0, 1, ... , S-1 do
5:      Load A[i*S]
6:      Mi0 = A[i*S] * B0; ... Mi3 = A[i*S] *B3
7:      for u = 1, 2, ... , S-1 do
8:          Load A[i*S + u]
9:          Load Bu0, Bu1, Bu2, Bu3
10:         Mi0 += A[i*S+u] * Bu0; ... Mi3 += A[i*S+u] *Bui3
11:     end for
12:     Mi0 = α Mi0 + β (Load Ci0); ... Mi3 = α Mi3 + β (Load Ci3)
13:     Store Mi0, Mi1, Mi2, Mi3
14: end for
```

```cpp
template <unsigned long block_num = 1>
inline void batch_Mult( const double * A , const double * B , double * C , double alpha, double
      beta, std::integral_constant<unsigned long,16>) {

constexpr int ind = 16 * block_num;
auto v_b = _mm256_loadu_pd( &B[0 ]);
auto v_b_ = _mm256_loadu_pd( &B[4 ]);
auto v_b__ = _mm256_loadu_pd( &B[8 ]);
auto v_b___ = _mm256_loadu_pd( &B[12]);

auto alpha_ = _mm256_set1_pd(alpha);
auto beta_ = _mm256_set1_pd(beta);

for (int iA = 0; iA < 16; iA++){

  auto tmp = _mm256_set1_pd( A[iA*ind] );
  auto v_c = tmp * v_b ;
  auto v_c1 = tmp * v_b_;
  auto v_c2 = tmp * v_b__;
  auto v_c3 = tmp * v_b___;

 for(int u = 1 ; u < 16 ; ++u ){
   tmp = _mm256_set1_pd( A[iA*ind +u ]);
   auto v_bt = _mm256_loadu_pd( &B[u*ind]);
   auto v_b_t = _mm256_loadu_pd( &B[u*ind+ 4]);
   auto v_b__t = _mm256_loadu_pd( &B[u*ind+ 8]);
   auto v_b___t = _mm256_loadu_pd( &B[u*ind+ 12]);
   v_c += tmp * v_bt ;
   v_c1 += tmp * v_b_t ;
   v_c2 += tmp * v_b__t ;
   v_c3 += tmp * v_b___t;
  }

  v_c = _mm256_loadu_pd(&C[iA*ind]) * beta_ + v_c * alpha_ ;
  v_c1 = _mm256_loadu_pd(&C[iA*ind+4]) * beta_ + v_c1 * alpha_ ;
  v_c2 = _mm256_loadu_pd(&C[iA*ind+8]) * beta_ + v_c2 * alpha_ ;
  v_c3 = _mm256_loadu_pd(&C[iA*ind+12]) * beta_ + v_c3 * alpha_ ;


  _mm256_storeu_pd( &C[iA*ind] , v_c );
  _mm256_storeu_pd( &C[iA*ind+4] , v_c1 );
  _mm256_storeu_pd( &C[iA*ind+8] , v_c2 );
  _mm256_storeu_pd( &C[iA*ind+12], v_c3 );
 }
}
```

Intel CPU performance analysis - 1

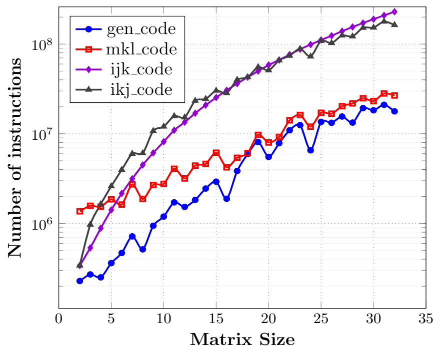


(a) # of load instructions

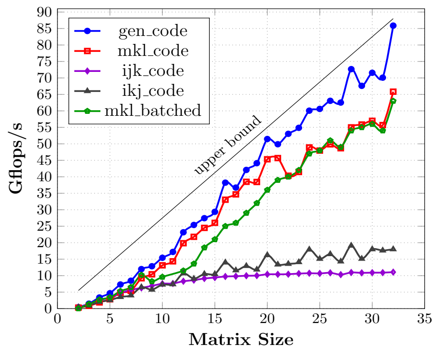


(b) # of store instructions

Intel CPU performance analysis - 2

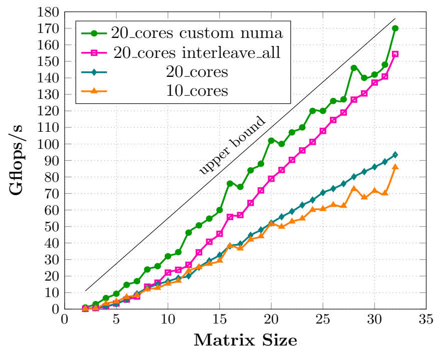


(c) Total CPU instruction count

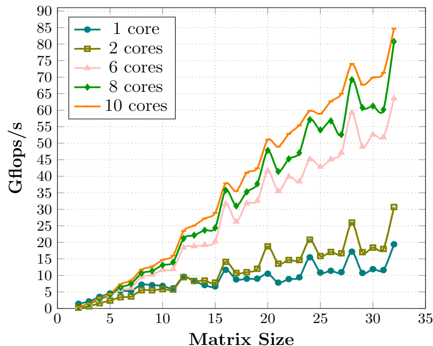


(d) CPU Performance comparison

Intel CPU performance analysis - 3

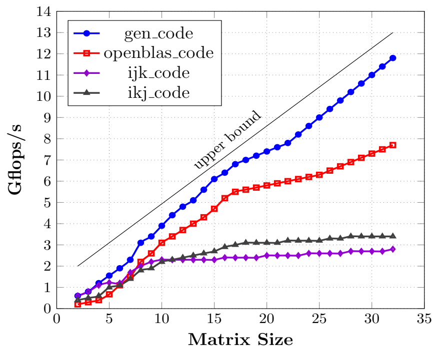


(e) Effect of the NUMA memory management



(f) Effect of the number of CPU cores

ARM CPU performance analysis



Modeling a problem for a GPU - CUDA

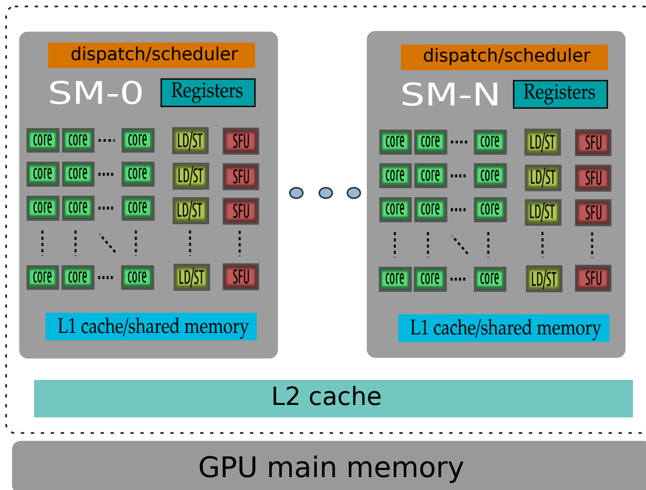
Take advantage of the architecture threads, thread blocks (TB) and Streaming Multiprocessors (SM)

- threads - lots of them
- thread blocks (TB) - shared memory
- Streaming Multiprocessors (SM) - shared scheduler
  - 64 Double Precision units per SM
  - 32 Load/Store (LD/ST) units per SM
  - 64KB of shared memory/L1 cache

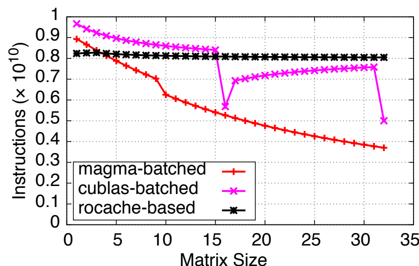
Use a hierarchical blocking model

Prefetch blocks of A and B

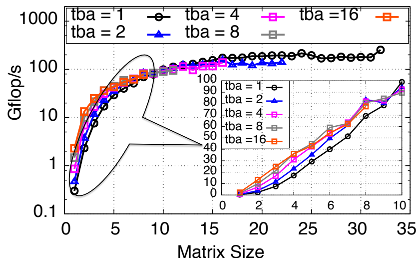
Overview of Kepler architectures



NVIDIA GPU performance analysis - 1

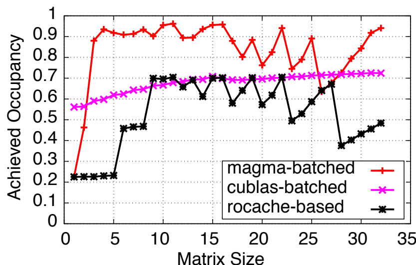


(g) Fraction of integer instructions

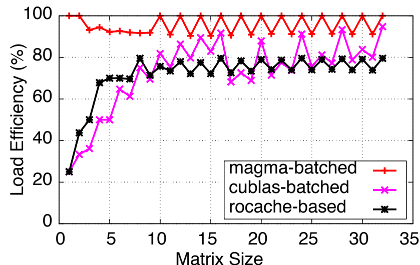


(h) Impact of TB aggregation

NVIDIA GPU performance analysis - 2

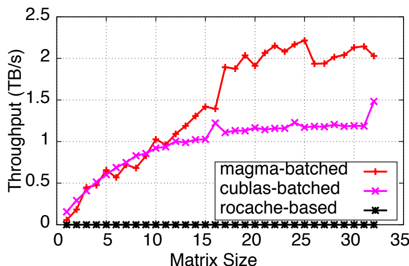


(i) GPU Achieved Occupancy

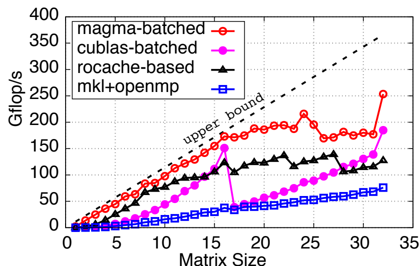


(j) GPU Global Memory Load Efficiency

NVIDIA GPU performance analysis - 3



(k) GPU shared memory throughput



(l) GPU Performance on K40c

To Sum it up

Different architectures require different optimizations

But optimization for CPUs or GPUs is generally similar

- CPU : SIMD instruction set vary between constructors/architectures
- GPU : Tend to be more different - textures, bandwidth, compute units

Concepts however remain the same

- If you don't write assembly, help the compiler !
- Architecture are hierarchical - Go step by step

Thanks for your attention