Declarative computations

Carl Nettelblad

2019-01-16
What is the bottleneck for large computations?

- Compute
- Memory
  - Size, bandwidth, latency?
- Give me some arguments!
- “It depends”
Memory bandwidth

• A new dual socket 56-core Xeon system can manage 165 GB/s in extreme benchmarks
  – That’s 3 GB/s sustained per core
Compute bandwidth

- Such a machine can perform 32 double precision floating point operations per clock per core at 1,900 MHz
  - Fused multiply add $A = B \times C + D$ (2 flops)
  - At the very least we expect to read $B$ and $C$ from memory
  - Each operand is a scalar of 8 bytes
  - $16 \times 2 \times 8 – 256$ bytes per core per clock
  - That’s 25,367 GB/s
  - The machine could read 165 GB/s
Compute bandwidth

- This is using vector instructions
- Non-FMA scalar code can do 2 double precision floating point operations per clock per core
- Still 1,600 GB/s, 10x memory bandwidth
Why?

• Is that compute useless?
• No
  – If we use the same data many times, it can be read from cache
  – **Caches** are just about fast enough to keep up
• This requires **data locality**
  – A feature of the problem
    • A scalar product between two vectors that are only used once is never data local)
  – A feature of the algorithm
    • Make sure that global data locality becomes local/temporal data locality
      – Make sure the data is reused while it is still in cache
Dense matrix multiplications

• “If you can turn your problem into DGEMM, you’re good”

• DGEMM is a BLAS routine doing $C = a \times A \times B + b \times C$

• If the A, B, C are large enough, this gives plenty of data locality
  – You only have to trust that the DGEMM implementation is efficient
  – If most of your time is spent here, it almost doesn’t matter what framework or language you are using
• Only some computations are dense matrix multiplications of modestly large matrices

• How do we make it easy to write performant code in other cases?
  – Promoting data locality, allowing for parallelism
Example expression

C = max(0, min(1, B.^3)) + D .* E

• How will MATLAB, or numpy, probably handle this?
  – Compute power, store in temporary F (1 R + 1 W)
  – Compute a min matrix, store in temporary G (1 R + 1 W)
  – Compute a max matrix, store in temporary H (1 R + 1 W)
  – Compute an elementwise product, store in temporary I (2 R + 1 W)
  – Compute a sum, store in temporary J (2 R + 1 W)
  – Copy J to C (1 R + 1 W)

• 8 matrix read operations, 6 writes
• Ideally, this could be done as 3 reads (B, D, E) and 1 write (C)

• Some of these steps can be eliminated, probably not all
• We *can* write this as an explicit loop in our language of choice

```plaintext
for i = 1:numel(B)
    C(i) = max(0, min(1, B(i)^3)) + D(i) * E(i);
end
```

• In e.g. Matlab and Python, this is slow
• Possibly quite complicated if not all operations are simple per-element
• Conflating *intent* and *sequence*
Imperative programming

- In imperative programming, it is all about sequence of operations
  - \((A + B) - C\) means
    - Compute \(A + B\), then subtract \(C\)
      - Or “as-if” principle
    - Natural implementation will lead to several scans over the data
  - for loop means
    - Perform each iteration, sequentially
What do we want?

- Both of these models are focused on the sequential order of execution
- Exactly the thing that is the bottleneck of current computers

- Sometimes, we have to identify parallelism ourselves
  - Sometimes, a shift in abstraction can be enough
What do we want?

- What if we could express “compute this expression, in whatever order makes sense, on whatever execution units makes sense”?
- What if we could express “perform this piece of code for every element, in some order, on some execution units”?
- What if we could express “this is the input, these are the expressions, be ready at some point to compute the results of some of these expressions”?
What do we want?

- What if we could express “compute this expression, in whatever order makes sense, on whatever execution units makes sense”?  
  - Expression templates in Eigen
- What if we could express “perform this piece of code for every element, in some order, on some execution units”?  
  - Various primitives in Thrust
- What if we could express “this is the input, these are the expressions, be ready at some point to compute the results of some of these expressions, and also perform automatic differentiation, just for fun”?  
  - Computation graphs in TensorFlow
Eigen

- Matrix and vector arithmetics library for C++

- Based on expression templates
  - Each expression becomes a unique type
  - $A + B$ becomes something like
    $\text{plus}<A\_\text{type}, B\_\text{type}>$
  - When you assign such an expression to a concrete matrix, it is evaluated
    * Evaluation solution chosen at compile-time
• Possible to combine small (e.g. up to 16-256 elements) vectors/matrices of fixed size and arbitrary dense and sparse matrices
  – Generating efficient code even for small cases
  – Can use LAPACK/BLAS/Intel MKL as backend where it makes sense
  – Performs vectorization and sometimes unrolling
    • Recompile for the relevant architecture
Our example

```cpp
using BigMatrix = Matrix<
double,
Dynamic,
Dynamic>; 

BigMatrix B = MatrixXd::Random(4096, 4096);
BigMatrix D = MatrixXd::Random(4096, 4096);
BigMatrix E = MatrixXd::Random(4096, 4096);
BigMatrix C = MatrixXd(4096, 4096);

C = B.array().pow(3.0).min(1).max(0) + D.array() * E.array();
```

- `array()` is the equivalent to dot operators in Matlab, instructing Eigen to perform elementwise operations.
• pow turned out to be a real limitation here
• Logical, pow, exp, sin, and even (somewhat) division and sqrt are more expensive than mul, add by far
• This applied to many of the technologies tried here, including Matlab

• In the Eigen case, pow was not vectorized, resulted in individual scalar calls for each element
• Vector pow does exist in the CPU, a bit strange

• So we will try to avoid this in all environments
Our example

```cpp
using BigMatrix = Matrix<double, Dynamic, Dynamic>;
BigMatrix B = MatrixXd::Random(4096, 4096);
BigMatrix D = MatrixXd::Random(4096, 4096);
BigMatrix E = MatrixXd::Random(4096, 4096);
BigMatrix C = MatrixXd(4096, 4096);

C = B.array().cube().min(1).max(0) + D.array() * E.array();
```

- Significant difference
Generated code

```assembly
mov  rax,qword ptr [rsp+328h]
mov  rcx,qword ptr [rsp+318h]
vmovupd ymm0,ymmword ptr [rcx+r8]
vmulpd ymm4,ymm0,ymmword ptr [rax+r8]
vbroadcastsd ymm3,mmword ptr [rsp+300h]
vbroadcastsd ymm2,mmword ptr [rsp+2f0h]
mov  rax,qword ptr [rsp+2e0h]
vmovupd ymm1,ymmword ptr [rax+r8]
vmulpd ymm0,ymm1,ymm1
vmulpd ymm1,ymm0,ymm1
vminpd ymm2,ymm1,ymm2
vmaxpd ymm0,ymm2,ymm3
vaddpd ymm1,ymm0,ymm4
mov  rax,qword ptr [rsp+1c0h]
vmovupdp ymmword ptr [rax+r8],ymm1
lea  r8,[r8+20h]
sub  rdx,1
jne  main+970h (07FF7CF6C1B40h)
```
using BigMatrix = Matrix<
  double,
  Dynamic,
  Dynamic>
};

BigMatrix B = MatrixXd::Random(4096, 4096);
BigMatrix D = MatrixXd::Random(4096, 4096);
BigMatrix E = MatrixXd::Random(4096, 4096);
BigMatrix C = MatrixXd(4096, 4096);

C = B.array().cube().eval().min(1).eval().max(0).eval() + (D.array() * E.array()).eval();
#pragma loop(no_vector)

```c
for (int j = 0; j < C.size(); j++)
{
    C.data()[j] = max(0., min(1., B.data()[j] * B.data()[j] *
        B.data()[j])) + D.data()[j] * E.data()[j];
}
```
Results

- Pow Eigen
- Good Eigen
- Forced eval Eigen
- Scalar loop
- numpy
- pow numpy
- Multi-core Matlab
- Single-core Matlab
- pow Matlab

- Which one is fastest? By how much?
### Results

<table>
<thead>
<tr>
<th>Version</th>
<th>Rel. time usage (lower is better)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-core Matlab</td>
<td>0.88</td>
</tr>
<tr>
<td>Good Eigen</td>
<td>1</td>
</tr>
<tr>
<td>Scalar</td>
<td>1.11</td>
</tr>
<tr>
<td>Single-core Matlab</td>
<td>1.36</td>
</tr>
<tr>
<td>Forced eval Eigen</td>
<td>8.74</td>
</tr>
<tr>
<td>numpy</td>
<td>10.9</td>
</tr>
<tr>
<td>pow Eigen</td>
<td>16.1</td>
</tr>
<tr>
<td>pow Matlab</td>
<td>20.0</td>
</tr>
<tr>
<td>pow numpy</td>
<td>21.8</td>
</tr>
</tbody>
</table>

Good Eigen managed roughly 14 GB/s (counting 3 reads, 1 write) on a single desktop Skylake core.
"As fast as C"

- We had a huge span of results for C(++) code here.
- If some framework claims to be as fast as C, what does that mean?

- Why was the scalar code almost as fast as the vectorized code?
  - What happens if we work with 32x32 matrices rather than 4096x4096?
  - Or 64x64?
  - 128x128?
Why is eval slow

- Multiple passes over the data
  - *and*
- Repeated allocations of temporary data
  - Actually seems like a weakness in Eigen (on MSVC)
  - Repeated allocations of objects of the exact same size should not kill performance
- On the other hand, faster than numpy anyway
Results

<table>
<thead>
<tr>
<th>Matrix size</th>
<th>Rel. time usage (lower is better)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32x32</td>
<td>4,5</td>
</tr>
<tr>
<td>64x64</td>
<td>4,0</td>
</tr>
<tr>
<td>128x128</td>
<td>2,8</td>
</tr>
<tr>
<td>4096x4096</td>
<td>1,1</td>
</tr>
</tbody>
</table>

What’s going on?
Matrix sizes and Matlab

• How does Matlab fare for single core computations with smaller matrices?
Matlab results

<table>
<thead>
<tr>
<th>Matrix size</th>
<th>Rel. time usage (lower is better)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32x32</td>
<td>6,4</td>
</tr>
<tr>
<td>64x64</td>
<td>5,1</td>
</tr>
<tr>
<td>128x128</td>
<td>3,6</td>
</tr>
<tr>
<td>4096x4096</td>
<td>1,4</td>
</tr>
</tbody>
</table>
• OpenCL and CUDA are "raw" interfaces to GPU programming
  – CUDA vendor-specific

• Thrust is a C++ wrapper for host and GPU programming
  – Only works with CUDA, since OpenCL has very few C++ features
## Eigen vs. Thrust

<table>
<thead>
<tr>
<th>Eigen</th>
<th>Thrust</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-focused, some GPU support</td>
<td>GPU-focused, CPU implementation as reference</td>
</tr>
<tr>
<td>Building expression templates for “closed” set of supported operations</td>
<td>Heavy use of functor objects, represent operations as code</td>
</tr>
<tr>
<td>Based on mathematical matrices and vectors</td>
<td>Based on computer science iterators</td>
</tr>
<tr>
<td>Linear algebra and related higher level operations</td>
<td>Transformations, reductions, mapping, sorting</td>
</tr>
</tbody>
</table>
Iterators

- Rather than specifying data sets as vectors, Thrust uses iterator
- Can simply be pointer to first and next after last element in a vector
- Can also be any other object that behaves the same way
  - i.e. compare positions, easily get value for arbitrary position $i$, go to next/previous
  - This is a quite convenient abstraction
Special iterators in Thrust

- **constant_iterator**
  - Dummy for giving the same value for any i
- **counting_iterator**
  - Return a linear sequence of values
Special iterators in Thrust

- **zip_iterator**
  - Combine two iterators to create an iterator over tuples of values (pairs, triples, etc)
  - Easy way to represent iterating over multiple vectors
- **permutation_iterator**
  - Really more of indexing iterator, takes two iterators and uses the indices returned from one of them to access specific elements from the other
- **transform_iterator**
  - Access elements from an input iterator, pass those to a functor, return the resulting value
(Some) algorithms in Thrust

- **transform**
  - Apply functor to each element in input, save in output
- **any_of**
  - Does passed-in functor return true for any value from input iterator
- **sort**
  - Sort values in place by natural order, by comparison functor, or by specified keys
(Some) algorithms in Thrust

- set_intersection
  - Accept two sorted sets, output to new iterator those elements that appear in both
- count
  - Return number of occurrences of specified value
Why is this useful?

• Something like `transform` can be quite easy to write yourself in CUDA or OpenCL
• Even writing something as simple as `count` or `any_of` is hard
  – You want parallelism on the order of thousands of threads
  – How do you keep track of the total sum (`count`)?
  – How do you determine when to return (`any_of`)?
• Thrust allows you to ignore how the parallelism is achieved, you provide iterators pointing to or generating data, and functors acting on the data

• Say **what** you want done on **what data**
  – Thrust will arrange the computations
Functors and lambdas

- A functor is a function object
  - If a C++ class overloads the () operator, it can be called just like a function
  - But also carry some data in the members, if it wants to

- It gets rather clunky to write separate small classes outside of your main function

- Lambda expressions solve this
  - Allows us to write code where it makes sense
  - What should happen in a transform call is specified in the function call, as code
auto begin = make_zip_iterator(make_tuple(B.begin(), D.begin(), E.begin()));
transform(begin,
    begin + SIZE * SIZE,
    C.begin(),
    [] __device__ __host__(tuple < T, T, T> t) -> T {
        return min((T)1, max((T)0, get<0>(t) * get<0>(t) * get<0>(t))) + get<1>(t) * get<2>(t);
    });
Components used

• Combine all three input matrices into one zip iterator
  – This iterator will have three-value tuple elements
• Specify the end of this iterator (indicates size)
• Specify the first element of the output iterator

• Specify a lambda expression for the calculation as a functor
Generating random numbers

template<typename T> struct prg
{
    T a, b;

    __host__ __device__
    prg(T _a=0.f, T _b=1.f) : a(_a), b(_b) {};

    __host__ __device__
    float operator()(const unsigned int n) const
    {
        thrust::default_random_engine rng;
        thrust::uniform_real_distribution<float> dist(a, b);
        rng.discard(n);

        return dist(rng);
    }
};
Generating random numbers

```cpp
transform(make_counting_iterator(0),
    make_counting_iterator(SIZE * SIZE), B.begin(), rg);
transform(make_counting_iterator(SIZE * SIZE),
    make_counting_iterator(SIZE * SIZE * 2), D.begin(), rg);
transform(make_counting_iterator(SIZE * SIZE * 2),
    make_counting_iterator(SIZE * SIZE * 3), E.begin(), rg);
```
We don’t need no input

- Rather than generate random numbers using transform
- Use a transform_iterator
- *If* the random number generator is fast enough, this can be faster than reading three matrices
  - But many “good” RNGs are pretty slow
- Basic concept still useful
  - Easy way to create matrix-free/”virtual matrix” code
auto make_rg_iterator = [rg](int v) {
    return
    make_transform_iterator(make_counting_iterator(v),
    []__device__ __host__(T x) { return rg(x); }); });
auto begin = make_zip_iterator(make_tuple(make_rg_iterator(0),
make_rg_iterator(SIZE * SIZE), make_rg_iterator(SIZE * 2)));
transform(begin,
    begin + SIZE * SIZE,
    C.begin(),
    [] __device__ __host__(tuple < T, T, T> t) -> T {
        return min((T)1, max((T)0, get<0>(t) * get<0>(t) * get<0>(t)) + get<1>(t) * get<2>(t);
    });}
Results

• Relative time usage 0.32 compared to single core Eigen
• 3x speedup, but using full GPU
  – On the other hand, our single core CPU performance could not scale (bandwidth constrained)

• This was on a Quadro P2000 mobile chip
  – Double precision performance 1/32 of floating point
  – Still, floating point version hardly 2x faster
  – Indicates *still* memory-bound
TensorFlow

- Eigen has even lower latency/overhead than BLAS
  - Free mixing of general CPU code and efficient vectorized expression templates
- Thrust allows arbitrary code in functors
  - But to run efficiently, it has to run on the GPU
- There are bandwidth and latency limitations for issuing commands and transferring data to the GPU
  - We want to express a large chunk of computations and data
  - Transfer of data between subtasks/steps should happen transparently on the GPU
- In Thrust, we can explicitly control flow of data using multiple function calls and special iterators
  - Not very declarative anymore
TensorFlow

- In deep learning, there are deep computational graphs
  - Outputs in one step are passed on to the next step
- Manual GPU implementations got extremely tedious
- Especially when we also want to compute gradients for all computational steps

- TensorFlow (and other libraries) provide a way to express computational graphs (including deep neural networks)
  - Specify data and variables
  - Reuse the graph multiple times
  - Execute the full graph on one or multiple GPUs
  - The normal use case is then to optimize some variables for training neural networks
TensorFlow vs. other abstractions

• Some other libraries to access GPUs focus more on coordinating the specific operations
• Again, compare Eigen and numpy
• You give TensorFlow the full graph first, all dependencies are known
Crude example

import numpy
import tensorflow as tf

hB = numpy.random.rand(4096,4096)
hD = numpy.random.rand(4096,4096)
hE = numpy.random.rand(4096,4096)

B = tf.constant(hB)
D = tf.constant(hD)
E = tf.constant(hE)

C = tf.maximum(tf.minimum(B*B*B, 1), 0) + D * E

with tf.Session() as sess:
    sess.run(C)
Performance?

• Real benefits when stacking multiple complex operations

• But you also get an advanced (accelerated) gradient-descent optimizer minimizing/maximizing that expression
Summary

• Parallelism has to address the proper bottleneck
• Movement of data and synchronization is at the core of parallelization
• Models that abstract common patterns allowing us to specify only what operations need to be performed can be convenient

• This can of low-level, tight parallelism can be combined with higher level threading, distributed computing
  – On low level, maximize temporal sharing
  – On high level, minimize spatial sharing