CUDA code generation

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Code generation

Computers are much better than humans at performing tedious repetitive tasks, such as large matrix-matrix multiplications.

Sometimes, this includes writing code!

The purpose of this presentation is to give an example, and show how simple and effective code generation can be.

The mathematical task

The objective was to compute

$$O_{ij} = \sum_{k,l} A_{ik}\, B_{il}\, C_{jkl}$$

where A, B, O are $N \times D$ matrices, with $D \leq 32 \ll N$,
and C is a $D \times D \times D$ tensor with all elements known
at compile time, and only a fraction $0.1 < s < 0.25$ non-zero.

Initial assessment:

- too much sparsity to use tensor cores?
- want to load in the elements of A and B only once
- want to use only one thread for each output O_{ij}

The mathematical task

$$O_{ij} = \sum_{k,l} A_{ik}\, B_{il}\, C_{jkl}$$

where A, B, O are $N \times D$ matrices, with $D \leq 32 \ll N$, and C is a $D \times D \times D$ tensor with sparsity s.

Initial assessment (continued):

- could do only one O_{ij} per thread, with different threads in a warp having same i, different j – warp loads in i^{th} row of A and B
- if N is sufficiently big, and/or there are multiple such products to be computed, better for a single thread to do whole i^{th} row of O – this is what I chose to implement

The mathematical task

$$O_{ij} = \sum_{k,l} A_{ik}\, B_{il}\, C_{jkl}$$

where A, B, O are $N \times D$ matrices, with $D \leq 32 \ll N$, and C is a $D \times D \times D$ tensor with sparsity s.

Initial assessment (continued):

- each thread has to
  - load $2\,D$ elements of A, B
  - do D^2 products for $A_{ik}B_{il}$, and then $s\,D^3$ FMAs
  - store D elements of O
- when $D = 25, s = 0.25$, about 30 FMAs per load/store
- needs less than $3\,D$ registers

CUDA kernel code

```
//
// include files
//
#include <stdio.h>
#include <string.h>
#include <math.h>
//
// kernel routine
//
__global__ void O_calc(int N, int D,
                const float* __restrict__ d_A,
                const float* __restrict__ d_B,
                      float* __restrict__ d_O)
{
  int tid = threadIdx.x + blockDim.x*blockIdx.x;
  float prod1, prod2;
```

CUDA kernel code

```
//
// load in A and B into registers
//
   float A00 = d_A[tid +  0*N];
   float B00 = d_B[tid +  0*N];
   float A01 = d_A[tid +  1*N];
   float B01 = d_B[tid +  1*N];
   float A02 = d_A[tid +  2*N];
   float B02 = d_B[tid +  2*N];
   float A03 = d_A[tid +  3*N];
   float B03 = d_B[tid +  3*N];
   ...
   ...
```

CUDA kernel code

```
//
// initialise O in registers
//
  float O00 = 0.0f;
  float O01 = 0.0f;
  float O02 = 0.0f;
  float O03 = 0.0f;
  ...
  ...
  ...
```

CUDA kernel code

```
//
// perform calculations
//
  prod1 = A00*B00;
  O00   = O00 + 0.174377f*prod1;
  O20   = O20 + 0.081610f*prod1;

  prod2 = A00*B01;
  O05   = O05 + 0.074208f*prod2;
  O10   = O10 + 0.139085f*prod2;
  O12   = O12 + 0.030585f*prod2;
  O13   = O13 + 0.136700f*prod2;
  O14   = O14 + 0.237413f*prod2;
  O24   = O24 + 0.204053f*prod2;
  ...
  ...
```

CUDA kernel code

```
//
// write out O to device array
//
   d_O[tid +  0*N] = O00;
   d_O[tid +  1*N] = O01;
   d_O[tid +  2*N] = O02;
   d_O[tid +  3*N] = O03;
   ...
   ...
   ...
   ...

}
```

CUDA kernel code generation

Total code length: 5375 lines for $D = 25$, 25% sparsity

This was generated by about 60 lines of python

Python generator code

```
import numpy as np
def code_gen(C):
  if( C.ndim != 3):
      raise ValueError('C has wrong number of dimen
  if (max(C.shape)!= min(C.shape)):
      raise ValueError('C has unequal dimensions')
  D = C.shape[1]

  print('//                                          ')
  print('// include files                            ')
  print('//                                          ')
  print('                                            ')
  print('#include <stdlib.h>                         ')
  print('#include <stdio.h>                          ')
  print('#include <string.h>                         ')
  print('#include <math.h>                           ')
```

Python generator code

```
print('//                                                   ')
print('// kernel routine                                    ')
print('//                                                   ')
print('                                                     ')
print('__global__ void O_calc(int N, int D,                 ')
print('             const float* __restrict__ d_A,          ')
print('             const float* __restrict__ d_B,          ')
print('                   float* __restrict__ d_O)          ')
print('{                                                    ')
print('  int tid = threadIdx.x + blockDim.x*blockIc
print('  float prod1, prod2;                                ')
```

Python generator code

```
print('//                                   ')
print('// load in A and B into registers    ')
print('//                                   ')

for d in range(D):
  print(f'   float A{d:02d} = d_A[tid+{d:2d}*N]; ')
  print(f'   float B{d:02d} = d_B[tid+{d:2d}*N]; ')

print('//                                   ')
print('// initialise O in registers         ')
print('//                                   ')

for d in range(D):
  print(f'   float O{d:02d} = 0.0f;           ')
```

Python generator code

```
print('//                                          ')
print('// perform calculations                     ')
print('//                                          ')

for k in range(D):
  for l in range(D):
    if (k+l > 0):
      print()
    m = np.mod(l + k*D, 2) + 1
    print(f'  prod{m:d} = A{k:02d}*B{l:02d};     ')
    for j in range(D):
      if ( C[j,k,l] != 0.0):
        print(f'  O{j:02d}    = O{j:02d}
                       + {C[j,k,l]:f}f*prod{m:d}; ')
```

Python generator code

```
print('//                                                          ')
print('// write out O to device array                              ')
print('//                                                          ')

for d in range(D):
  print(f'  d_O[tid + {d:2d}*N] = O{d:02d};                        ')

print('}                                                           ')
return
```

Extension

Here we knew the values of all elements of C at compile time.

If instead we knew the sparsity pattern (i.e. which elements are zero) but not the values of the non-zero elements, then we could load the non-zero values into shared memory, and then all threads could load them in from there when needed – would need just minor changes to the generator code

(The constant cache is only 8KB so might not be big enough to hold all of the non-zeros)

Conclusion

Code generation is surprisingly easy.

I don’t use it often in my research, but I have used it previously on a major project (OP2 – a separate talk) and one other small project.