FlashAttention: an interesting CUDA application

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Overview

- mathematical algorithm
  - particular focus on softmax function
- naive implementation, and why it performs poorly
  - how many operations performed?
  - how much data transferred?
- re-examining implementation
  - how to minimise data transferred?
  - how to implement softmax?

FlashAttention

Three references:

- FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness (PDF) – 1750 citations
- FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning (PDF) – 750 citations
- FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision (PDF) – 50 citations

The code is available on Github but I have not looked at the code, just read the papers.

I am not concerned with why this “transformer” is important for machine learning (original Google Research “Attention is All You Need” paper has over 150k citations!) – I’m only focussed on how to achieve good CUDA performance.

Mathematics

The objective is to evaluate

$$\overbrace{O}^{N\times d} = \mathrm{softmax}\left(\overbrace{Q}^{N\times d} \times \overbrace{K^T}^{d\times N}\right) \times \overbrace{V}^{N\times d}$$

where typical values for N and d are $N = 1024$, $d = 64$.

Note that if we didn't have the $\mathrm{softmax}$ operation, then we would organise the matrix product as

$$O = Q \times \left(K^T \times V\right)$$

since $N \gg d$. This would minimise floating point operations, $O(d^2 N)$, and data movement, $O(d\,N)$.

softmax

The $\mathrm{softmax}$ operation applied to a row vector u_j, $1 \le j \le N$ is defined as

$$v_j = \mathrm{softmax}(u)_j = \exp(u_j) \;/\; \sum_{k=1}^{N} \exp(u_k).$$

so that $0 < v_j < 1$ and $\displaystyle\sum_{k=1}^{N} v_k = 1$.

When applied to the matrix $S = Q\,K^T$, it is applied row-wise, i.e. applied separately to each row of S.

softmax

There is an important practical complication.

When using low precision, variables have a very restricted range. e.g. for fp16, the range is roughly $10^{-4} - 10^5$.

This means that there is a likelihood that $\exp(u_j)$ will lead to overflow (Inf) or underflow (0). A single overflow is a problem; it's also a problem if all of them underflow.

Consequently, the implementation uses the following:

$$m = \max_j u_j, \quad v_j = \exp(u_j - m) \ / \ \sum_{k=1}^{N} \exp(u_k - m).$$

Avoids overflow, and if some underflow that's not a problem.

Naive implementation

The naive implementation is fairly simple:

- calculate $S = Q\,K^T$, using cuBLAS which loads Q and K by blocks to compute and store blocks of S;
- calculate $P = \mathrm{softmax}(S)$, loading in rows of S, doing warp/block reductions as required, and then storing P;
- calculate $O = P\,V$, using cuBLAS which loads P and V by blocks to compute and store blocks of O.

It's possible that PyTorch can do all of this at a very high level, making the implementation really easy.

Unfortunately, the performance is poor because S and P are $N\!\times\!N$ matrices.

Performance estimate

With any new application, before doing any coding I do a back-of-the-envelope estimate of the performance to identify whether the application is compute-bound or bandwidth-limited.

i.e. is the limit on performance the Gflops of the GPU, or the GB/s bandwidth between GPU memory and GPU?

Quite often it is the memory bandwidth which is the limiting feature, and that is the case here with the naive implementation.

Naive implementation

Remembering that $N \gg d$, the main data transfers are

- storing S
- loading S
- storing P
- loading P

Total data moved for $N = 1024, d = 64$, assuming fp32 (4 Bytes per variable) is

$$1024^2 \times 4 \times 4\,\text{Bytes} = 0.016 \text{ GB}$$

The A100 GPU is capable of 2 TB/s, so this equates to 8 μs.

Naive implementation

On the compute side we have two major elements which we will consider separately.

The first is the dN^2 multiply-adds to compute S, and another dN^2 multiply-adds to compute O.

With $N = 1024, d = 64$, this requires

$$64 \times 1024^2 \times 2 \times 2 = 0.256 \text{ GFlop}$$

The A100 tensor cores are capable of 156 TFlops (TF32 / float), so this equates to 1.6 μs.

Naive implementation

The second element is the N^2 exponentials to compute P.

With $N=1024, d=64$, and noting that the exponential is equivalent to 8 floating point operations, (see section 5.4.1 in the CUDA C/C++ Programming Guide) this requires

$$1024^2 \times 8 = 0.008 \text{ GFlop}$$

The A100 standard cores are capable of 19.5 TFlops (FP32), so this equates to 0.4 μs.

Conclusion: both computing times are much smaller than the data transfer time, so the application is bandwidth-limited.

Performance estimate

In general, the back-of-the-envelope performance estimate influences the way in which you write your code.

If compute-bound:

- don't worry (too much) about cache efficiency
- minimise integer index operations
- if using double precision, think whether it's needed

If bandwidth-limited:

- ensure efficient cache use – may require extra coding
- may be better to re-compute some quantities rather than fetching them from device memory
- if using double precision, think whether it's needed

FlashAttention

Coming back to FlashAttention, the back-of-the-envelope analysis shows that the challenge is to reduce the amount of data moved between the GPU memory and the GPU.

Looking again at the naive implementation

- calculate $S = Q\,K^T$
- calculate $P = \mathrm{softmax}(S)$
- calculate $O = P\,V$

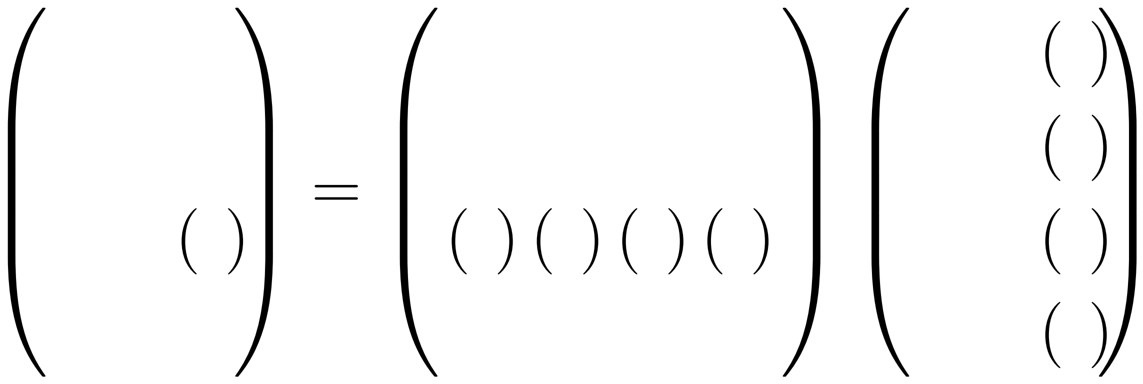
it means we need to avoid storing/loading S and P, which in turn means that we can't do the above three steps sequentially; they have to be overlapped somehow.

The only arrays we should be loading in are Q, K and V, each of which is $N\!\times\!d$ so much smaller than S and P.

FlashAttention

Let’s start with the final step.

As discussed previously in Lecture 5, the output matrix O can be computed in blocks (or “tiles”).



$$O_{ij} = \sum_k P_{ik} V_{kj}$$

The blocks V_{kj} are easily loaded in; the challenge is generating the blocks P_{ik} on-the-fly.

FlashAttention

For the first block, P_{i1}, we can compute the corresponding block S_{i1} on the fly using the same block/tile approach:

$$\begin{pmatrix} \\ \\ (\ ) \\ \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \\ (\ )(\ ) \\ \\ \\ \end{pmatrix} \begin{pmatrix} (\ ) & & & \\ (\ ) & & & \end{pmatrix}$$

$$S_{i1} = \sum_k Q_{ik} K_{1k}$$

The blocks Q_{ik} and K_{1k} are easily loaded in to perform this calculation.

FlashAttention

If there is only one block, P_{i1}, then we would compute it as

$$P_{i1} = \mathrm{softmax}(S_{i1})$$

This would require a warp reduction (assuming the block is handled by a single warp) to first compute

$$m_i = \mathrm{rowmax}(S_{i1})$$

a column vector with each element being the max of the row of S_{i1}. Then we need a second reduction to do a row-sum of the scaled exponentials, and finally the scaling of the row elements to give P_{i1}.

We can then compute $P_{i1}V_{1j}$ which gives O_{ij}, if there's only one block P_{i1}.

FlashAttention

The tricky bit, and the hardest thing to understand in the whole FlashAttention algorithm/implementation, is what to do if there are two blocks, P_{i1} and P_{i2}.

To make things a little simpler, let's suppose we don't have to do the re-scaling of the exponentials using the m_i rowmax's. Hence, all we have to do for each row n is compute

$$\exp(S_{nj}) \;/\; \sum_{k=1}^{N} \exp(S_{nk})$$

If we treat both blocks as described on the last slide, then the problem is that each block is dividing by a partial sum of exponentials, not a full row sum.

FlashAttention

However, changing to the full row-sum simply involves a multiplicative factor so with two blocks what we get for a particular row n is

$$\alpha_n (P_{i1} V_{1j})_n + (P_{i2} V_{2j})_n$$

where the re-scaling of row n of P_{i2} uses

$$\sum_{(1)+(2)} \exp(S_{nk}) = \sum_{(1)} \exp(S_{nk}) + \sum_{(2)} \exp(S_{nk})$$

and

$$\alpha_n = \sum_{(1)} \exp(S_{nk}) \; / \sum_{(1)+(2)} \exp(S_{nk}).$$

FlashAttention

This can be generalised to multiple blocks, repeatedly adding to each row sum, and rescaling what has been computed previously.

It can also be generalised to include the re-scaling of the exponentials to keep things in range – read the original FlashAttention paper to see the details.

FlashAttention

Two little “hacks” for improved performance?

1) the single precision warp maximum m can be computed in a single instruction by

```
m_max = __int_as_float(
      __reduce_max_sync(-1,__float_as_int(S) ) );
```

treating the bits of the `float` as if they are an `int`

(Currently CUDA only has this warp max for signed and unsigned integers)

FlashAttention

2) m does not need to be precisely the row-max of S; all that is needed is that $\exp(S_j - m)$ is in range

This means that an alternative is to use $m = n \log 2$ where n is an integer given by

$$n = \lfloor \max_j S_j / \log 2 \rfloor$$

The point of this is that the re-scaling factors

$$\exp(\Delta n \log 2) = 2^{\Delta n}$$

can be evaluated very cheaply without doing a floating point exponential.

FlashAttention

One thing I have not discussed so far is the size of the blocks.

Ideally, would like all of the blocks of Q, K, V to fit in shared memory, so that they all get loaded in just once.

In practice, the shared memory is not big enough, so instead the block sizes are chosen as big as possible to minimise the number of times the data has to be reloaded.

Reverse pass

For the reverse pass (back-propagation) there are two key mathematical ingredients.

- forward pass $A = B\,C$ leads to reverse pass

$$\overline{B} = \overline{A}\,C^T, \qquad \overline{C} = B^T\,\overline{A}$$

  where $\overline{A}, \overline{B}, \overline{C}$ are the adjoint/dual variables (“gradients” in the Flash paper, denoted by $\mathrm{d}A, \mathrm{d}B, \mathrm{d}C$)

- forward pass $v = \mathrm{softmax}(u)$ leads to reverse pass

$$\begin{aligned} \overline{u}_j = \overline{\mathrm{softmax}}(u, \overline{v})_j &= \frac{\exp(u_j)\,\overline{v}_j}{\sum_k \exp(uk)} - \frac{\exp(u_j)\,\sum_k \exp(uk)\,\overline{v}_k}{(\sum_k \exp(uk))^2} \\ &= v_j\,\overline{v}_j - v_j\,(v^T\overline{v}) \end{aligned}$$

Naive implementation

The naive implementation is again fairly simple:

- calculate $\overline{P} = \overline{O}\, V^T;\ \ \overline{V} = P^T\, \overline{O}$
- calculate $\overline{S} = \overline{\mathrm{softmax}}(S, \overline{P})$
- calculate $\overline{Q} = \overline{S}\, K;\ \ \overline{K} = \overline{S}^T Q$

using stored values for S and P from the forward pass.

This again requires $O(N^2)$ data transfer for $S, P, \overline{S}, \overline{P}$ which leads to poor performance.

FlashAttention

The key to the design of the reverse pass in FlashAttention is working out mathematical expressions for the three outputs $\overline{V}, \overline{Q}, \overline{K}$.

These are then implemented blockwise, in a similar way to the forward pass, using stored information on

$$m_i, \quad L_i = \sum_k \exp(S_{ik} - m_i)$$

from the forward pass. The reverse pass re-computes S_{ik} along the way, but it reduces the data transfer and that's what matters for performance.

(It is often better to re-compute things rather than storing them and re-loading them from GPU memory.)

FlashAttention-3

FlashAttention-3 achieves a very high percentage of the peak compute capability of the H100 GPUs.

Extra performance is achieved by using new instructions which enable the asynchronous loading of data directly into shared memory, without passing through a register – a new highly specialised feature presumably intended primarily for ML applications.

An increasing concern is the exponentials and other operations not using the tensor cores. As d increases the cost of these becomes less important because there are relatively more tensor operations.

Final comments

An interesting example of how to approach a new CUDA application:

- do a rough performance assessment to decide whether the application is compute-bound or bandwidth-limited
- this tells you which aspects to focus on in the implementation
- sometimes best to re-compute things to reduce data transfer
- this is an example of more general technique of “loop fusion”, overlappng the execution of two algorithm parts to reduce the storing/loading of intermediate variables
- if it’s an important enough application, seek assistance from NVIDIA DevTechs – two worked on FlashAttention-3