Transformer From Scratch

Thitrin Sastarasadhit[1], Prof. Kenjiro Taura[2]

[1]Chulalongkorn University, Bangkok, Thailand
[2]University of Tokyo, Tokyo, Japan

Agenda

- Methodology
- Small-Size Model on Single Thread
- Full-Size Model on Single and Multiple Threads
- Productivity
- Conclusion

Methodology

Models

There are 4 models:

- C++	Implemented from scratch
- Chapel	
- PyTorch A	From Transformer-from-Scratch
- PyTorch B	PyTorch A with the transformer layer replaced with `torch.nn.Transformer`

**This project does performance tests on CPU, single thread, and multiple threads*
**The Chapel and C++ implementations were very similar; all variables could be mapped from one to the other.*

Test Environments

Property	Machine A	Machine B
CPU	AMD Ryzen 7 4800H with Radeon Graphics	Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz
RAM	6.67 GB	204.45 GB
Clang	Ubuntu clang version 19.1.1 (1ubuntu1) Target: x86_64-pc-linux-gnu Thread model: posix	clang version 19.1.3 Target: x86_64-unknown-linux-gnu Thread model: posix
Chapel	chpl version 2.4.0 built with LLVM version 19.1.1 available LLVM targets: xtensa, m68k, xcore, x86-64, x86, wasm64, wasm32, ve, systemz, sparcel, sparcv9, sparc, riscv64, riscv32, ppc64le, ppc64, ppc32le, ppc32, nvptx64, nvptx, msp430, mips64el, mips64, mipsel, mips, loongarch64, loongarch32, lanai, hexagon, bpfeb, bpfel, bpf, avr, thumbeb, thumb, armeb, arm, amdgcn, r600, aarch64_32, aarch64_be, aarch64, arm64_32, arm64	chpl version 2.4.0 built with LLVM version 19.1.3 available LLVM targets: amdgcn, r600, nvptx64, nvptx, aarch64_32, aarch64_be, aarch64, arm64_32, arm64, x86-64, x86
Python	Python 3.11.13 PyTorch : 2.3.0 Numpy : 2.3.0	Python 3.11.13 PyTorch : 2.5.1 Numpy : 2.0.1

Model Configuration

Parameter	Machine A	Machine B	Description
dModel	32	512	Dimension of embedding layer of the encoder and decoder
sequenceLength	128	256	Maximum length of input sequence
dFF	256	2048	Dimension of the feed-forward layer inside the encoder and decoder
N	6	6	Number of transformer encoder, decoder layers (stacked).
head	8	8	Number of attention heads in multi-head attention layer
secVocab	15700	15700	Size of source vocabulary (number of unique tokens).
tgtVocab	22470	22470	Size of target vocabulary

The model architecture are based on the Attention Is All You Needed paper

attention is all you needed paper: https://arxiv.org/abs/1706.03762

Performance Measurement

- Timers are inserted into each layers
- The Models were trained on English-Italian machine translation task, the dataset were taken from opus_book
- The time of each iteration of each was gathered, trimming 10% fastest and slowest iterations
- The average and standard deviation were calculated

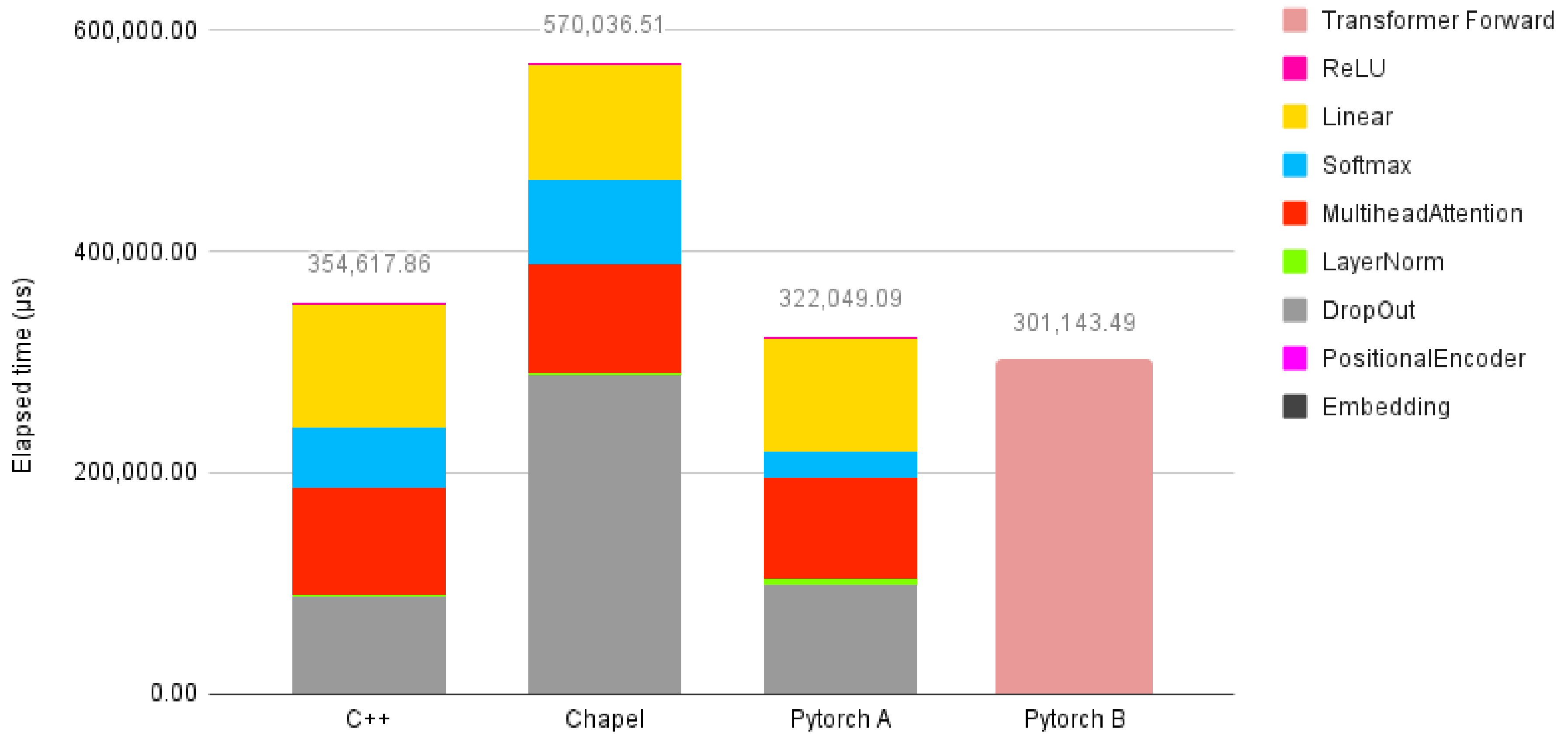
The Hugging Face link for opus_book: https://huggingface.co/datasets/Helsinki-NLP/opus_books
The model was executed for 500 and 40 iterations on Machines A and B, respectively.

Small-Size Model on Single Thread

Description

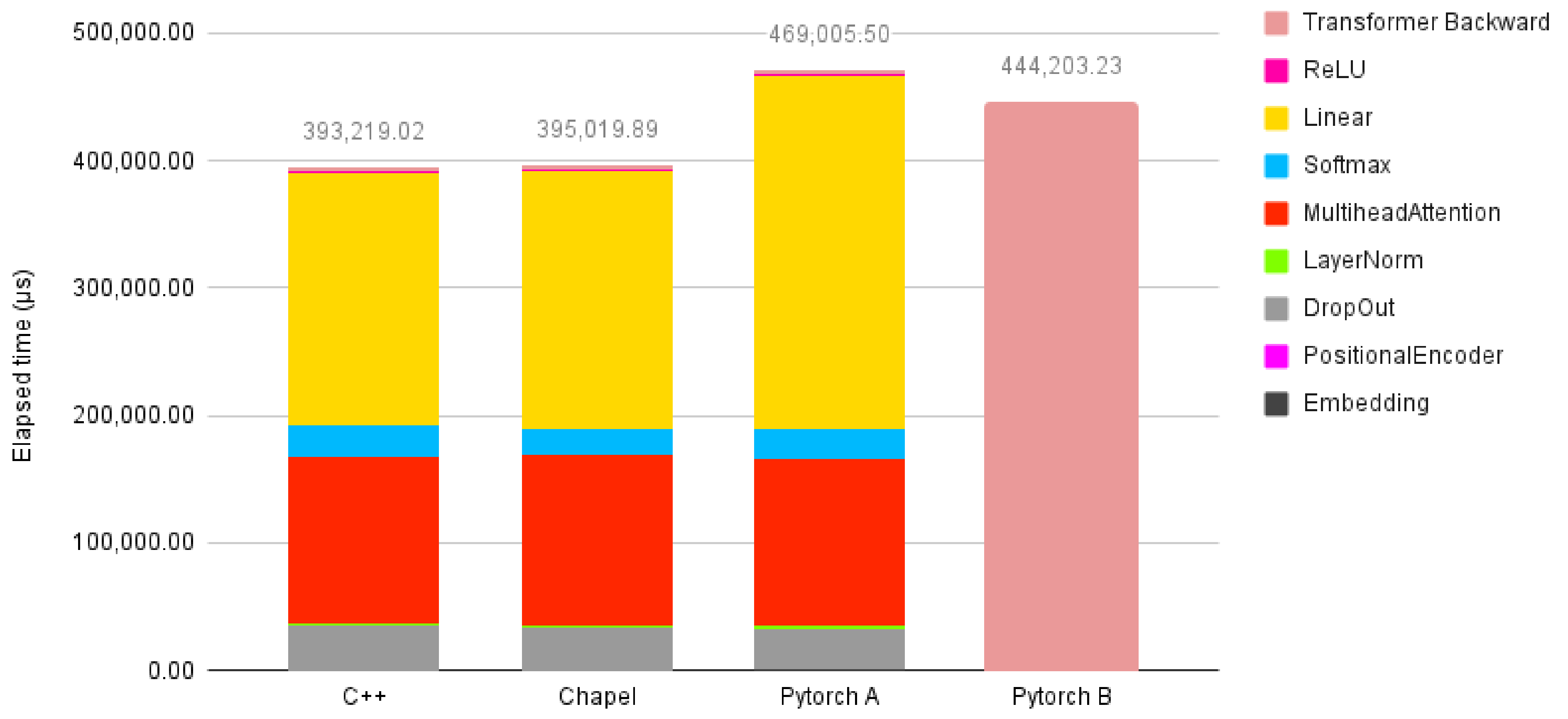
- Tested small model on Machine A
- The models were run for 500 iterations
- GitHub link for all single-thread code: https://github.com/markthitrin/Transformer/tree/SingleThread
- Google Spreadsheet for detailed results: https://docs.google.com/spreadsheets/d/1aHkE9Ckl0-waxVwu-f4dIJ0peM6jIUQv3IU1-bFa0p0/edit?gid=2029252533#gid=2029252533

Result of Forward Pass



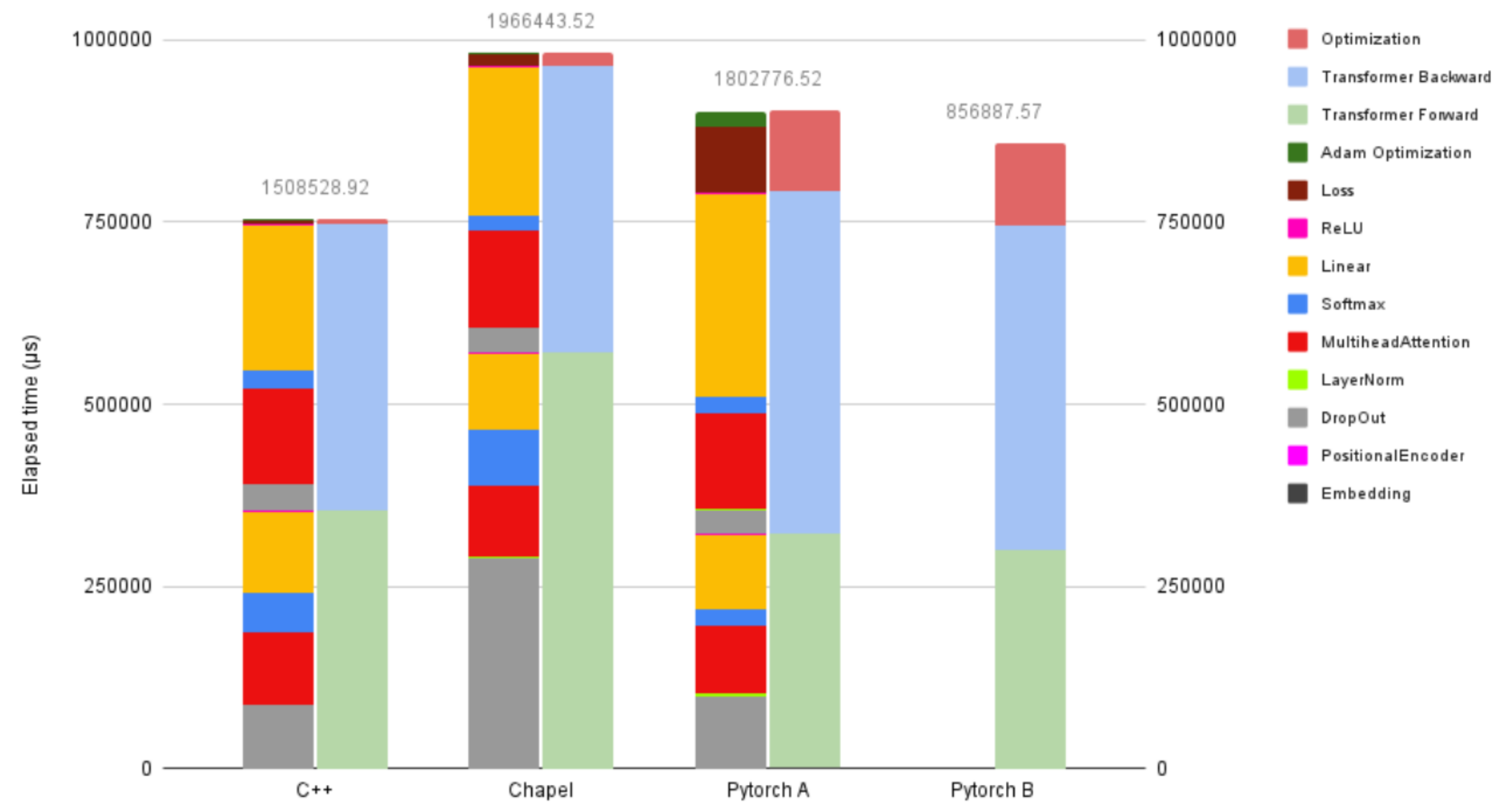
Time spent on each layer (in microseconds) during a single forward-pass training iteration for each model, tested on Machine A (single-threaded) using the small model configuration.

Result of Backward Pass



Time spent on each layer (in microseconds) during a single backward-pass training iteration for each model, tested on Machine A (single-threaded) using the small model configuration.

Overall Result



Time spent on each layer (in microseconds) per training iteration (including forward, backward, and update) for each model, tested on Machine A (single-threaded) using the small model configuration.

Matrix Representation

In C++, The TensorView is for capturing a portion of the Tensor.

```
class Tensor {
    Tensor(int row, int column) {data = new float[row * column];}
    ~Tensor() {delete[] data;}
    float* data;
};
class TensorView {
    TensorView(Tensor& t) {data = t.data;}
    ~TensorView() {/*do nothing*/}
    float* data;
};
```

But in Chapel, ref is not allowed in a class or record

```
class TensorView {
    // ref data; error
}
```

Matrix Representation

- 1D array
- No multi-dimensional array
  - Slow when doing `for (a,b) in zip(A,B)`
  - No loop optimization (no loop unrolling, no vectorization)
  - Huge overhead from `advance_chpl`
  - Can be avoided with `for i in A.domain`
  - Mentioned in the Chapel website as performance concern
- Nest array, `var A: [0..#N][0..#N] real(32)`, is better but not best
  - Basically 1D array of 1D arrays
  - Non-continuous array

Chapel performance concerns: https://chapel-lang.org/docs/technotes/optimization.html#performance-problems-with-multidimensional-zippered-iteration
GitHub Issue "Multidimensional zippered iteration kills performance": https://github.com/chapel-lang/chapel/issues/13147

Matrix Multiplication

- Chapel can do better than C++ at some specific size of matrix, and worse at other size.
- Even though the compiler-generated code looks the same.

Matrix Operation

- Element-wise addition, multiplication, reduction, etc.
- Design matters a lot than expected

```
proc PlusReduce1(ref A: [?D] real(32), out output: real(32)) : void {
    output = 0.0;
    for i in D {
        output += A[i];
    }
}

proc PlusReduce2(D: domain(1), ref A: [] real(32)) : void {
    output = 0.0;
    for i in D {
         output += A[i];
    }
}
```

Matrix Operation

```
proc PlusReduce3(in start: int, in count: int, ref A: [] real(32), out output: real(32))
: void {
    output = 0.0;
    for i in start..#count {
        output += A[i];
    }
}

proc PlusReduce4(ref A: [?D] real(32), out output real(32)) : void {
    output = + reduce(A);
}
```

Matrix Operation

Design	Optimization
PlusReduce1	No
PlusReduce2	Unrolling
PlusReduce3	Unrolling + vectorize
PlusReduce4	No, create task

To prevent future problem, PlusReduce3's design is used

**This is hard to reproduce; it happens only on some specific code structures*
**I have tried operator overloading too. It gives the same performance as PlusReduce1.*

```
operator +=(ref sum: real(32), ref A: [] real(32)) {
    var output: real(32) = 0.0;
    for i in A.domain {
         output += A[i];
    }
    sum = output;
}
```

Softmax

- Slow compared to C++
- Chapel don't exponential vectorization (`_ZGVdN8v_expf_avx2`) while Clang enables vectorization using `-fveclib=libmvec`
- Chapel refuses to vectorize exponential function, even with:
  - Simple for loop iterating over the array's domain
  - Simple for loop iterating over the array's elements
  - Switching from `real(32)` to `real(64)`
  - Direct assignment `B = exp(A)`
  - Using foreach loops.
  - Passing the same flags used in Clang via `--ccflag`
  - Using `--no-ieee-float`

Process	Performance (μs)
Softmax Forward	C++: 53,759.49 Chapel: 75,521.40
Softmax Backward	C++: 25,531.68 Chapel: 20,907.40
Softmax Total	C++: 79,291.17 Chapel: 96,428.80

Dropout

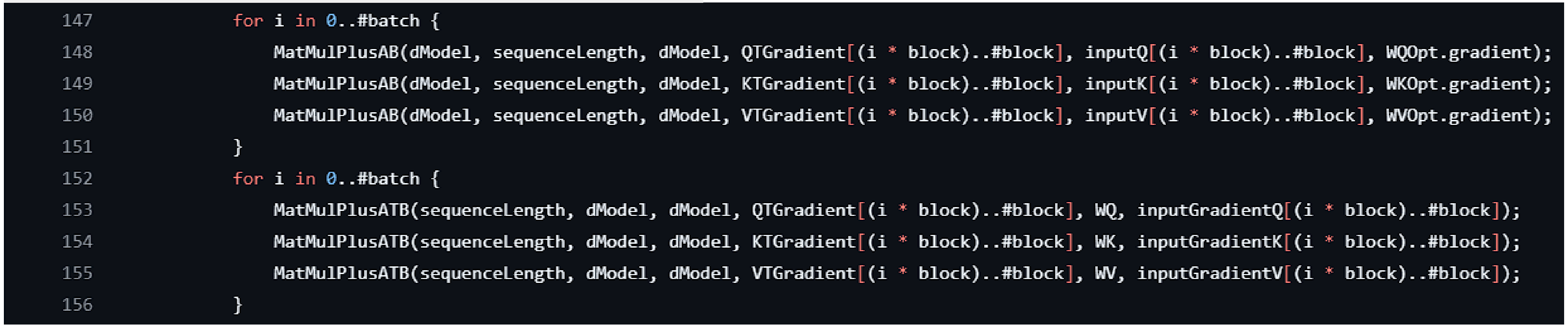
- Use `randomStream.fill()`
  - Need `CHPL_RT_NUM_THREADS_PER_LOCALE=1` when do single thread experiment
- Use integer random.

Process	Performance (μs)
Dropout Forward	C++: 87,045.49 Chapel: 288,983.16
Dropout Backward	C++: 35,605.02 Chapel: 34,410.46
Dropout Total	C++: 122,650.51 Chapel: 323,393.62

Multihead Attention

- The forward pass works fine
- The issue is in the weight gradient and the next layer’s gradient computation during the backward pass

Transformer/Chapel/MultiheadAttention.chpl



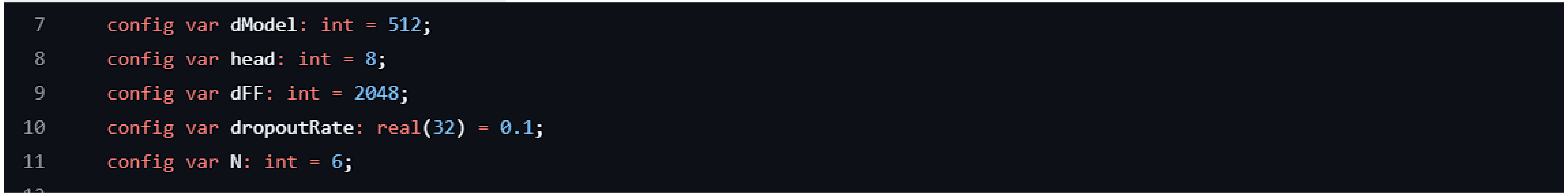
```
        for i in 0..#batch {
            MatMulPlusAB(dModel, sequenceLength, dModel, QTGradient[(i * block)..#block], inputQ[(i * block)..#block], WQOpt.gradient);
            MatMulPlusAB(dModel, sequenceLength, dModel, KTGradient[(i * block)..#block], inputK[(i * block)..#block], WKOpt.gradient);
            MatMulPlusAB(dModel, sequenceLength, dModel, VTGradient[(i * block)..#block], inputV[(i * block)..#block], WVOpt.gradient);
        }
        for i in 0..#batch {
            MatMulPlusATB(sequenceLength, dModel, dModel, QTGradient[(i * block)..#block], WQ, inputGradientQ[(i * block)..#block]);
            MatMulPlusATB(sequenceLength, dModel, dModel, KTGradient[(i * block)..#block], WK, inputGradientK[(i * block)..#block]);
            MatMulPlusATB(sequenceLength, dModel, dModel, VTGradient[(i * block)..#block], WV, inputGradientV[(i * block)..#block]);
        }
```

The loop was heavily unrolled but no vectorization

Multihead Attention

- The Problem was fixed by changing `param` to `var` in `config.chpl`

Transformer/Chapel/Config.chpl



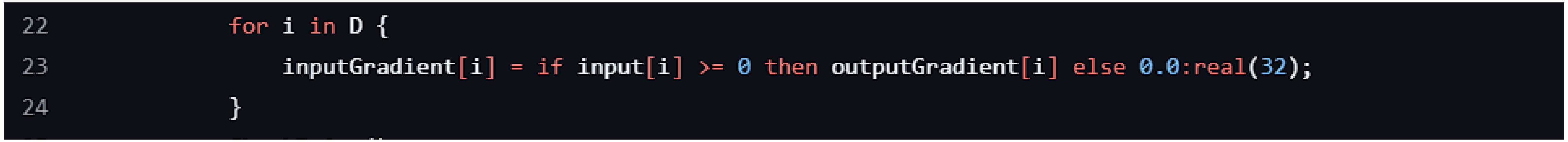
```
config var dModel: int = 512;
config var head: int = 8;
config var dFF: int = 2048;
config var dropoutRate: real(32) = 0.1;
config var N: int = 6;
```

This is quite a tricky solution

ReLU

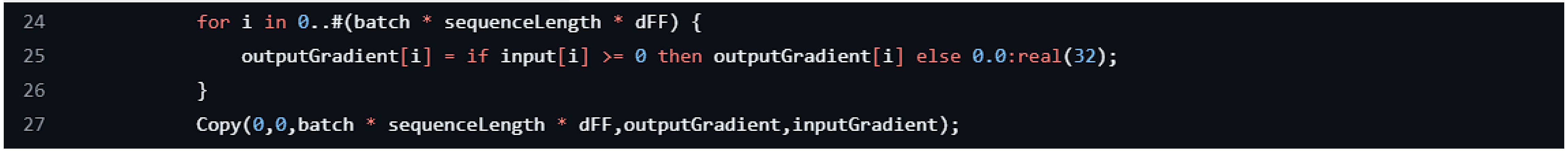
- One problem is that the backward pass needs to be divided into two sections

Transformer/Chapel/ReLU.chpl (old)



```
for i in D {
    inputGradient[i] = if input[i] >= 0 then outputGradient[i] else 0.0:real(32);
}
```

Transformer/Chapel/ReLU.chpl (new)



```
for i in 0..#(batch * sequenceLength * dFF) {
    outputGradient[i] = if input[i] >= 0 then outputGradient[i] else 0.0:real(32);
}
Copy(0,0,batch * sequenceLength * dFF,outputGradient,inputGradient);
```

This allows optimization to take place

ReLU

- Another mystery is that when tested on the small-size model, Chapel is slightly faster in the forward pass. But when tested on the full-size model, it becomes much slower

Process	Performance (μs)
ReLU Forward (Small Size)	C++: 2,003.26 Chapel: 1,170.46
ReLU Forward (Full Size)	C++: 42,211.31 Chapel: 239,258.80

- The only difference I found in the compiler-generated code is that Chapel and C++ took different approaches

```
// Chapel
load mem -> res
max 0,res -> res
store res -> mem
```

```
// C++
max 0,mem -> res
store res -> mem
```

**Both version got same degree of vectorized and loop unrolling*

- This effect can also be seen in the backward pass of LayerNorm.

Other Layers

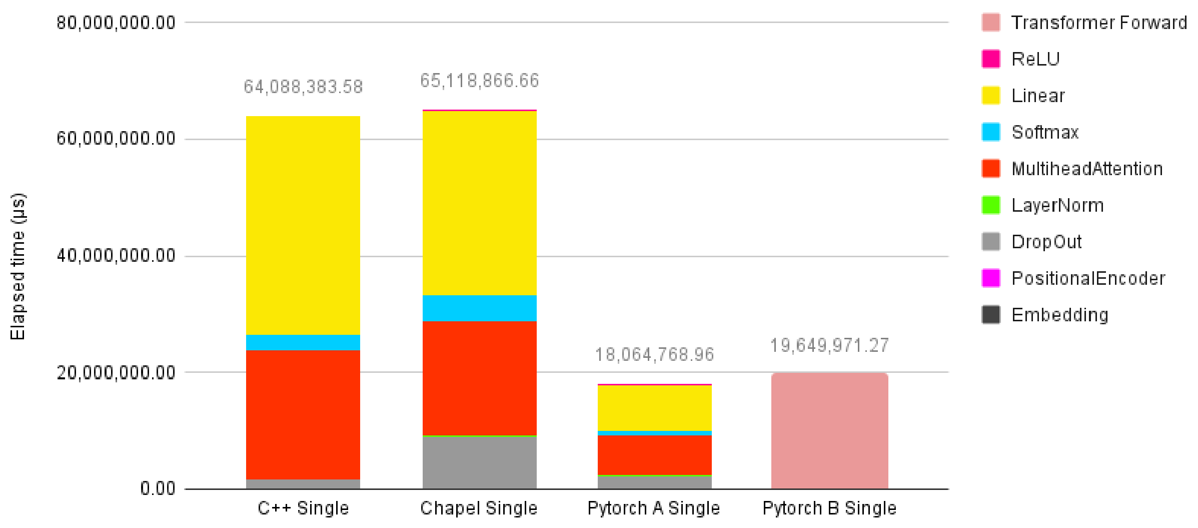
- Other layers are working fine
- The parameter updating (Adam optimization) in C++ and Chapel is much faster than than in PyTorch

Full-Size Model on Single and Multiple Threads

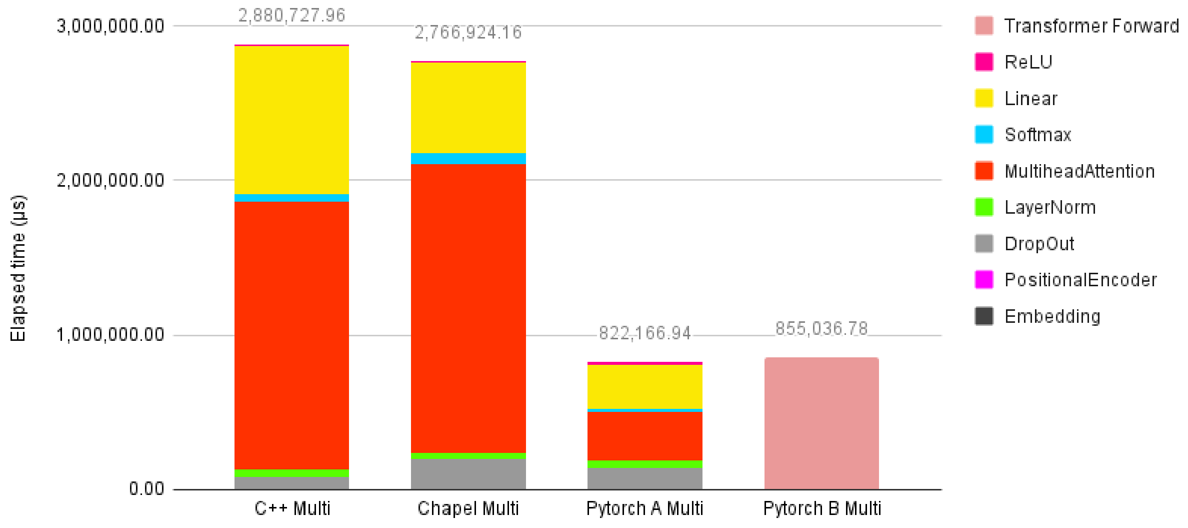
Description

- Tested full-size model on the Machine B
- The models were run for 40 iterations
- C++ uses OpenMP
- Chapel uses `forall`, `coforall` , and custom iterators
- The degree of parallelism is estimated for each layer/operation
- The degree of parallelism is the same in both C++ and Chapel
- GitHub link for all single-thread code: https://github.com/markthitrin/Transformer/tree/MultiThread
- Google Spreadsheet for detailed results: https://docs.google.com/spreadsheets/d/1aHkE9Ckl0-waxVwu-f4dIJ0peM6jIUQv3IU1-bFa0p0/edit?gid=2029252533#gid=2029252533

Result of Forward Pass



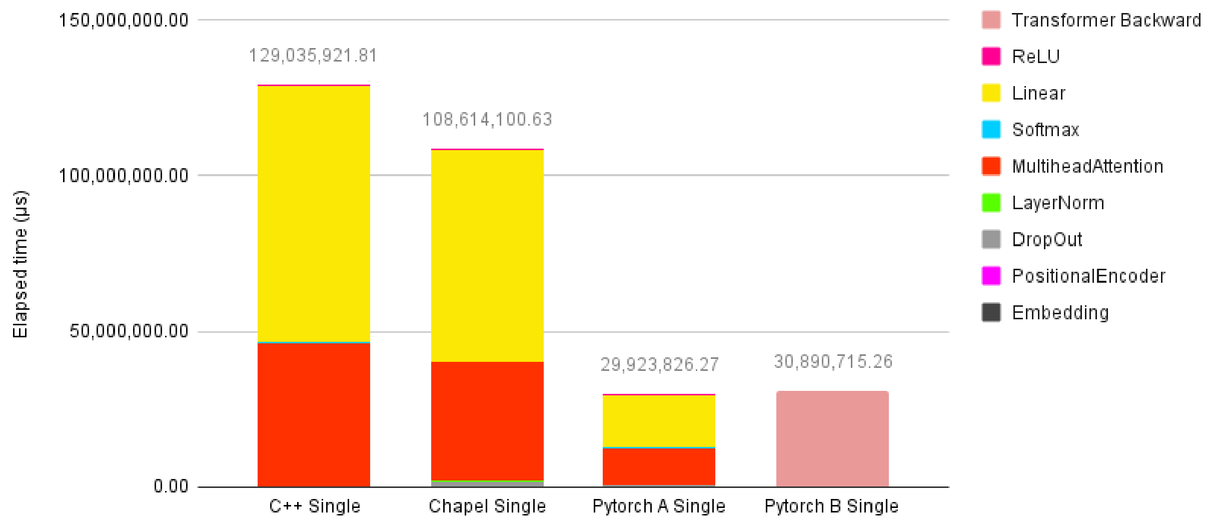
(a)



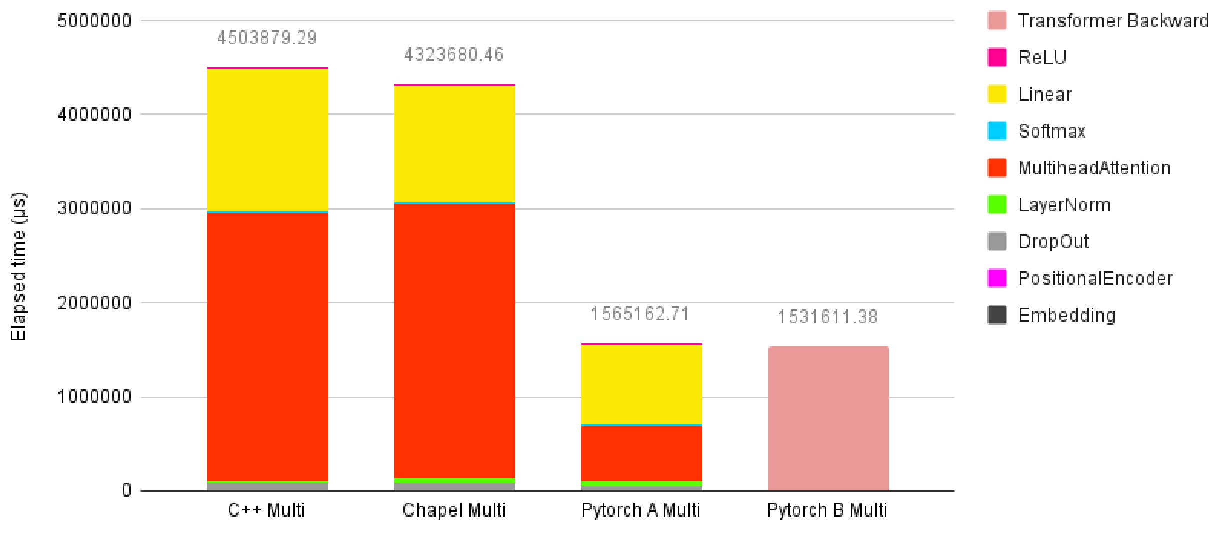
(b)

Time spent on each layer (in microseconds) during a single forward-pass training iteration for each model, measured on Machine B (single-threaded (a), multi-threaded (b)) using the full-size model configuration.

Result of Backward Pass



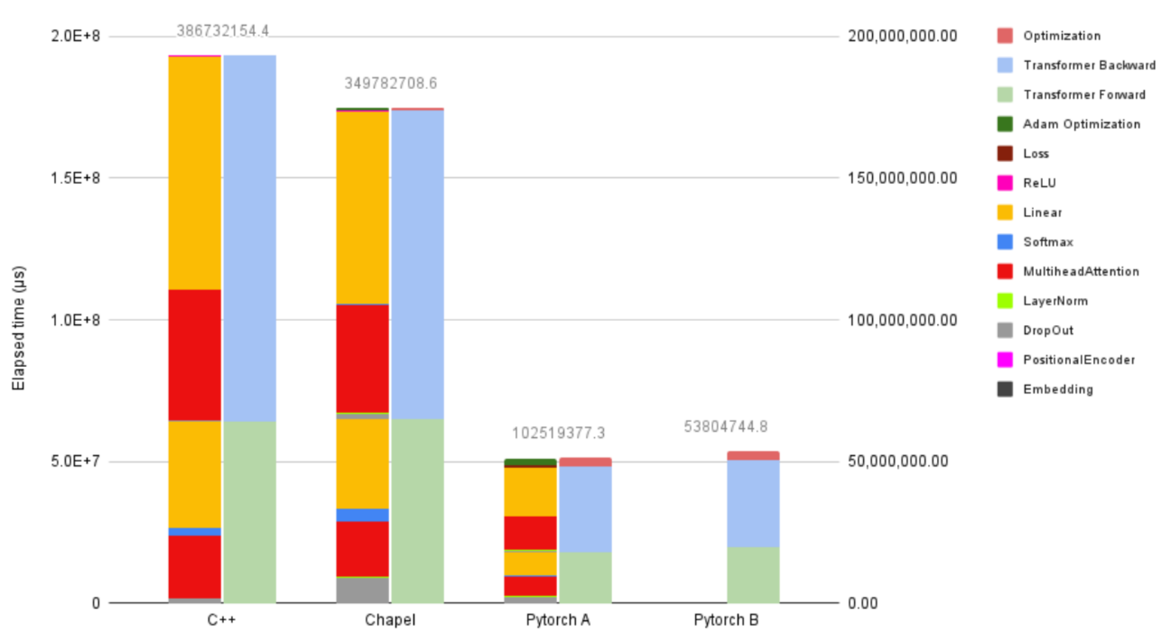
(a)



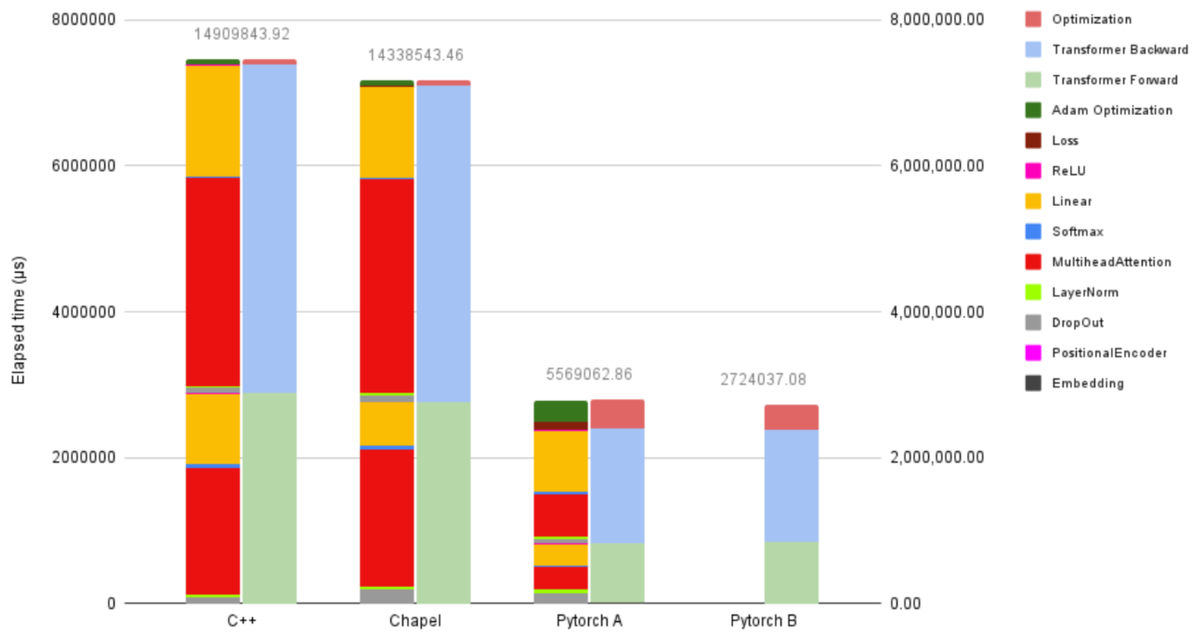
(b)

Time spent on each layer (in microseconds) during a single backward-pass training iteration for each model, measured on Machine B (single-threaded (a), multi-threaded (b)) using the full-size model configuration.

Overall Result



(a)



(b)

Time spent on each layer (in microseconds) per training iteration (including forward, backward, and update) for each model tested on Machine B (single-threaded (a), multi-threaded (b)) using the full-size model configuration.

Softmax

- A buffer is needed to store exponential values
- Unlike C++, Chapel can not perform stack allocation

```
// C++
#pragma omp parallel
for(int i = 0;i < row;i++) {
    float buffer[size];// stack memory
    // do something
}
```

The buffer is stored in stack memory

```
// Chapel
forall i in Par(start, end, numThread) {
    var buffer: [0..&N] real(32);
    // do something
}
```

The buffer is allocated and deallocated in every iteration

- This can be solved by moving the buffer declaration outside the loop

Other Layer

- Many layer perform as well as C++, even if they are slower in single-threaded. This is likely due to being bound by memory bandwidth
- Parameter Updates (Adam optimization) in C++ and Chapel are significantly faster than in PyTorch

Productivity

Productivity

- Things I like:
  - Easy to learn, similar to Python
  - Simple parallel programming through `for` loops
  - Requires type declaration of variables
  - Object memory management
  - Memory management across threads
  - Easier to run programs on multiple locales
- A Few Drawbacks I Noticed:
  - Long compilation time
  - All the performance issues that needed tricky solution I mentioned
  - Type casting between number types (e.g. `real(32)` from/to `real(64)`)
  - Generative AI support is limited

Conclusion

Conclusion

- This project compares four transformer models, implemented in C++, Chapel, Python
  - The achieve performance is reasonable
  - Chapel outperforms C++ in some parts
  - Performance issues were found, required tricky solutions
- Limitation in this project
  - No GPU and multi-locales
  - Not the most optimal code

For suggestions, advice, comments, or questions?, please contact me here:
thitrin.sastarasadhit@gmail.com

Thank you