Profiling and Analysis of BERT-Inspired HPC Algorithm

Abishek Chakravarthy
Roll Number: CS22B2054

February 2025

Contents

0.1. Introduction

0.1.1. Problem Statement and Motivation

Modern natural language processing (NLP) models, such as BERT, rely on computationally intensive operations—especially in the transformer encoder block where self-attention and feedforward networks dominate the workload. In this project, a simplified version of these components is implemented in C++ to simulate the key operations of BERT (token embedding, self-attention, feedforward transformation, and softmax computation). Due to the heavy matrix multiplications, normalization steps, and repeated loop iterations, the code represents a significant computational workload with a high degree of data-level and task-level parallelism.

0.1.2. Computational Intensity and Scope for Parallelization

The core operations in this algorithm include:

- **Matrix Multiplication**: Multiple calls for projecting token embeddings into queries, keys, and values, as well as for combining outputs.
- **Self-Attention Computation**: Calculating dot products between tokens and applying softmax across large score matrices.
- **Feedforward Neural Network**: Performing dense matrix multiplications with non-linear activation functions.
- **Layer Normalization**: Computing per-token statistics for numerical stability.

Each of these operations is computationally intensive. In a parallelized implementation (using OpenMP, MPI, or CUDA), these tasks can be divided among multiple cores or GPUs. Initial estimates indicate that by parallelizing the self-attention and matrix multiplication routines, the overall execution time could be reduced by a factor corresponding to the number of available cores, with further improvements expected when scaling to distributed or GPU environments.

0.1.3. Need for Profiling

Before parallelizing, it is crucial to understand the performance characteristics of the serial code. This project uses three profiling techniques:

- **Functional Profiling using gprof**: To identify which functions (e.g., `self_attention`, `matmul`, `feedforward_layer`) consume the most CPU time.

- **Line Profiling using gcov**: To obtain detailed execution counts and branch coverage statistics at the source-code level, helping to pinpoint "hot" lines that are executed most frequently.
- **Hardware Resource Profiling using LIKWID**: To gather hardware-level performance metrics (such as floating-point operations, memory bandwidth, and cache usage) that provide insight into how well the code utilizes the underlying CPU.

These profiling tools help guide the optimization process by identifying bottlenecks and confirming where parallelization will have the greatest impact.

0.2. Functional Profiling Using gprof

0.2.1. Overview of the Serial Code Functions

The main functions in the serial code include:

- **generate_random_matrix**: Generates a matrix with dimensions `rows x cols` using a Xavier/Glorot initialization scheme (basically a scheme for random number generation and initialization).
- **matmul**: Performs matrix multiplication between two matrices A and B.
- **layer_norm**: Normalizes each row of the input matrix so that each token's representation has zero mean and unit variance.
- **self_attention**: Simulates multi-head self-attention by projecting the input into Query (Q), Key (K), and Value (V) matrices, computing scaled dot-product attention, and concatenating the results.
- **feedforward_layer**: Implements a two-layer feedforward neural network with ReLU activation followed by layer normalization.
- **softmax**: Converts raw scores (logits) into probabilities by applying the softmax function with numerical stabilization.

0.2.2. gprof Detailed Output and Key Observations

The code was compiled with the `-pg` flag, and the resulting gprof output has been analyzed in detail. The profiling report (see below) reveals that the `matmul` function

dominates the execution profile, consuming approximately 92.66% of the total runtime (15.40 seconds out of 16.62 seconds), with 35 calls averaging 0.44 ms per call. The `self_attention` function, while critical for BERT's performance, accounted for around 6.32% of the runtime, and the `generate_random_matrix` function contributed only marginally.

Key observations include:

- **Dominance of Matrix Multiplication:** With over 92% of execution time spent in `matmul`, this function is the primary candidate for parallelization.
- **Inefficiencies in Self-Attention:** Although `self_attention` only consumes 6.32% of the time, its interaction with `matmul` (by invoking it multiple times) suggests that even small improvements here could have compound effects.
- **Random Number Generation Overhead:** The Mersenne Twister functions, while individually negligible, are called extensively (over 12 million calls), indicating potential for optimization if needed.
- **Interdependency of Functions:** The `self_attention` function relies heavily on multiple invocations of `matmul`. Even though its direct time consumption is lower, optimizing `matmul` will indirectly benefit self-attention.
- **Scalability Potential:** The clear performance bottlenecks identified by gprof offer a straightforward roadmap for parallelization, particularly targeting the computationally heavy matrix operations.

0.2.3. gprof Output Listing

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 92.66     15.40    15.40       35     0.44     0.44  matmul
  6.32     16.45     1.05        1     1.05    15.71  self_attention
  0.54     16.54     0.09    19268     0.00     0.00  mersenne_twister_engine
  0.36     16.60     0.06       28     0.00     0.01  generate_random_matrix(int, in
  0.06     16.61     0.01 12015092     0.00     0.00  mersenne_twister_engine::opera
  0.06     16.62     0.01                             _init
  0.00     16.62     0.00       65     0.00     0.00  std::vector<...>::vector(...)
  0.00     16.62     0.00       30     0.00     0.00  frame_dummy
```

0.00	16.62	0.00	4	0.00	0.00	std::__cxx11::basic_string<...
0.00	16.62	0.00	3	0.00	0.00	print_matrix(...)
0.00	16.62	0.00	3	0.00	0.00	std::vector<...>::~vector()
0.00	16.62	0.00	2	0.00	0.00	layer_norm
0.00	16.62	0.00	1	0.00	0.00	print_vector
0.00	16.62	0.00	1	0.00	0.89	feedforward_layer
0.00	16.62	0.00	1	0.00	0.00	softmax
0.00	16.62	0.00	1	0.00	0.00	std::_Vector_base<...>::~_Vect

The call graph further details the nested calls and confirms that most of the computation time cascades from `self_attention` into multiple invocations of `matmul`. This reinforces the need to prioritize optimizing matrix operations in any parallelization strategy.

0.3. Line Profiling Using gcov

0.3.1. Coverage Statistics and Execution Analysis

Line-level profiling was performed using gcov with the flags `-fprofile-arcs -ftest-coverage -O0 -g`. The detailed execution report highlighted that:

- **Hotspots in Core Loops:** Core computational loops, especially within `matmul` and `self_attention`, are executed an enormous number of times (in the order of billions for inner loops). This clearly marks these loops as critical performance bottlenecks.
- **Predictable Control Flow:** Branch coverage in functions such as `softmax` and normalization routines is nearly complete, indicating predictable control flows. This allows optimization efforts to focus on enhancing arithmetic throughput without worrying about branching inefficiencies.
- **Uniform Workload Distribution:** The uniformity of execution counts across loop iterations suggests that the workload is evenly distributed. This is promising for effective parallelization, as it implies that loop iterations can be distributed across multiple processing units without significant load imbalance.
- **Opportunities for Loop Unrolling and Vectorization:** The frequency of loop execution suggests that manual optimizations like loop unrolling or leveraging compiler vectorization may provide additional performance benefits, complementing the parallelization strategy.

These results confirm that the inner loops of the `matmul` function are the most frequently executed, which directly correlates with the gprof findings and reinforces the case for parallelizing these critical sections.

0.4. Hardware Resource Profiling Using LIKWID

0.4.1. Overview and Challenges

Hardware resource profiling was conducted using LIKWID to collect low-level performance metrics. However, during the process, a segmentation fault occurred when running `likwid-accessD`, likely due to missing kernel modules or permission issues. After resolving these issues, LIKWID provided a comprehensive breakdown of hardware performance across several groups including FLOPS, memory, L2 and L3 caches, cycle activity, micro-operations, TLB data, energy, and false sharing.

0.4.2. Detailed LIKWID Output and Key Observations

The following is an excerpt from the LIKWID output for Hardware Thread 3, illustrating the extensive metrics collected:

```
Group 1: FLOPS_DP
+-----------------------------------------+---------+--------------+
|                  Event                  | Counter |  HWThread 3  |
+-----------------------------------------+---------+--------------+
|             INSTR_RETIRED_ANY           |  FIXC0  | 964007861388 |
|          CPU_CLK_UNHALTED_CORE          |  FIXC1  | 429267025311 |
|          CPU_CLK_UNHALTED_REF           |  FIXC2  | 246953313756 |
| FP_ARITH_INST_RETIRED_128B_PACKED_DOUBLE |   PMC0  |            0 |
|   FP_ARITH_INST_RETIRED_SCALAR_DOUBLE   |   PMC1  |    177716008 |
| FP_ARITH_INST_RETIRED_256B_PACKED_DOUBLE |   PMC2  |            0 |
+-----------------------------------------+---------+--------------+

+-------------------------+------------+
|          Metric         | HWThread 3 |
+-------------------------+------------+
|   Runtime (RDTSC) [s]   |    99.8709 |
|   Runtime unhalted [s]  |   165.6100 |
|       Clock [MHz]       |  4505.6117 |
|           CPI           |     0.4453 |
```

```
|       DP [MFLOP/s]       |     1.7795 |
|     AVX DP [MFLOP/s]     |          0 |
|     Packed [MUOPS/s]     |          0 |
|     Scalar [MUOPS/s]     |     1.7795 |
| Vectorization ratio [%]  |          0 |
+--------------------------+------------+

Group 1: MEM
+-----------------------+---------+--------------+
|         Event         | Counter |  HWThread 3  |
+-----------------------+---------+--------------+
|   INSTR_RETIRED_ANY   |  FIXC0  | 964008441894 |
| CPU_CLK_UNHALTED_CORE |  FIXC1  | 442999592328 |
|  CPU_CLK_UNHALTED_REF |  FIXC2  | 253265457024 |
|       DRAM_READS      | MBOX0C1 |   1492452923 |
|      DRAM_WRITES      | MBOX0C2 |    249429659 |
+-----------------------+---------+--------------+
...

Group 1: ENERGY
+-----------------------+---------+--------------+
|         Event         | Counter |  HWThread 3  |
+-----------------------+---------+--------------+
|   INSTR_RETIRED_ANY   |  FIXC0  | 964009049543 |
| CPU_CLK_UNHALTED_CORE |  FIXC1  | 421457948552 |
|  CPU_CLK_UNHALTED_REF |  FIXC2  | 240736509540 |
|       TEMP_CORE       |   TMP0  |           77 |
|     PWR_PKG_ENERGY    |   PWR0  |    4170.3107 |
|     PWR_PP0_ENERGY    |   PWR1  |    3978.5703 |
|     PWR_PP1_ENERGY    |   PWR2  |            0 |
|    PWR_DRAM_ENERGY    |   PWR3  |     118.1470 |
+-----------------------+---------+--------------+
```

Key observations from the LIKWID data include:

- **Low Floating-Point Throughput:** The DP (double precision) throughput is measured at approximately 1.78 MFLOP/s, which is significantly lower than expected, indicating that the current implementation does not effectively leverage vectorized instructions.

- **Memory Bandwidth Utilization:** With a memory bandwidth of over 1082 MB/s and substantial data volumes being processed, the memory subsystem is heavily utilized. Optimizations in memory access patterns could thus have a notable impact.
- **Cache Behavior:** L2 and L3 cache metrics show low miss rates, but the cycle activity data indicates that a non-negligible fraction of cycles are spent waiting on memory (as seen in the high percentage of cycles without execution due to L1D and L2 pending events).
- **Energy Efficiency:** The energy consumption data (with the package energy around 4170 J and an average power of approximately 43 W) provides a baseline for measuring the impact of future optimizations on power efficiency.

0.5. Key Findings and Observations

0.5.1. Dominant Functions

- The `matmul` function is the most time-consuming, consuming nearly 93% of the execution time.
- Self-attention accounts for 4% of the execution time.

0.5.2. Line Profiling Insights

- The gcov report reveals that inner loops within `matmul` are executed billions of times, suggesting that optimizing these lines can lead to substantial performance gains.
- Branch coverage is high in critical regions, indicating that the control flow is predictable and optimization should focus on computational efficiency rather than logic changes.

0.5.3. Potential for Parallelization

- Given the high computational intensity in `matmul`, parallelization using OpenMP (for multi-core CPUs) or CUDA (for GPUs) is likely to yield significant speed-ups.
- The scalability potential is high since many of the operations can be independently computed (data parallelism) or split across different processing units (model parallelism).

0.5.4. Profiling Tools Utility

- gprof provided a clear function-level breakdown, identifying hotspots.
- gcov offered a detailed view of which lines are executed most frequently, confirming that the performance-critical sections are in the inner loops.
- LIKWID, despite initial setup challenges, delivered a comprehensive view of hardware resource usage, highlighting potential bottlenecks in floating-point throughput, memory bandwidth, and energy efficiency.

0.6. Conclusion

This report provided a comprehensive profiling analysis of a BERT-inspired HPC algorithm. The functional profiling with gprof identified that `matmul` is the primary performance bottleneck, consuming 92.66% of the CPU time, while the line-by-line analysis with gcov confirmed that critical inner loops are the most frequently executed parts of the code. The LIKWID hardware profiling, despite initial setup issues, further revealed significant insights into the hardware utilization, particularly in floating-point operations and memory accesses.

The key findings suggest that significant performance improvements can be achieved by parallelizing these computationally intensive operations. Optimizing the inner loops, particularly in `matmul` and `self_attention`, using OpenMP, MPI, or CUDA, is likely to reduce execution time dramatically. Overall, the profiling results justify the need for parallelization and provide a clear roadmap for where to focus optimization efforts.