Assignment 6: Exploring Thread-Level Parallelism (TLP) in Shared-Memory  
Multiprocessors Using gem5 Part 2

NAME: Nihar Turumella STUDENT ID: 005012274

## [https://github.com/[nturumella12274-ucumberlands/MSCS531\_Assignment-6-Exploring-Thread-Level-Parallelism-TLP-in-Shared-Memory-Multiprocessors-](https://github.com/nturumella12274-ucumberlands/MSCS531_Assignment-6-Exploring-Thread-Level-Parallelism-TLP-in-Shared-Memory-Multiprocessors-)](https://github.com/nturumella12274-ucumberlands/MSCS531_Assignment5---Exploring-Data-Level-Parallelism-DLP-in-Modern-Computing-.git)

## Part 2: Exploring Shared-Memory Architectures with gem5

**MinorCPU Familiarization**

Thread-level parallelism is becoming more prevalent in contemporary computing systems as a means of boosting performance. The `FloatSimdFU` is one of the customizable functional units available in the `MinorCPU` in gem5. This feature provides a modular platform for the study of TLP. In this study, I have examined the influence of operational latency (`opLat`) and issue latency (`issueLat`) on TLP, assessed the performance of a multi-threaded daxpy kernel, and explored the implications for multi-core systems.   
  
I conducted an analysis of the `MinorCPU.py` and `MinorDefaultFUPool` source files to ascertain the function of `opLat` and `issueLat` in the `MinorFU` class. In order to establish a basis for parameter optimization, the default configurations and a variety of functional units, such as `FloatSimdFU`, were studied.

**FloatSimdFU Design Space Exploration**

The `FloatSimdFU` in `MinorDefaultFUPool` was modified to evaluate configurations in which the aggregate of `opLat` and `issueLat` was seven cycles.   
- Configuration 1: `opLat = 1, issueLat = 6`   
- Configuration 2: `opLat = 2, issueLat = 5`   
  
- Configuration 3: `opLat = 3, issueLat = 4`   
  
  
  
Gem5's `build/x86/gem5.opt` was used to implement and evaluate each configuration.

A computer screen shot of text

Description automatically generated

A screen shot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated

**Multi-Threaded Daxpy Kernel Simulation**

A multi-threaded daxpy kernel was created, with each thread processing a subset of the input vectors. The simulation was optimized for multi-core systems with 2, 4, and 8 cores.   
Consistent output was guaranteed by synchronization mechanisms.

A computer screen shot of a black background

Description automatically generated

The multi-threaded DAXPY kernel, which is implemented in C using POSIX threads, efficiently utilizes thread-level parallelism to execute the operation y = a × x + y across vast arrays. The operation is divided among four threads to assure concurrent execution, and the arrays x and y are predefined to contain 10,000 elements in this implementation. The main function initializes the arrays by setting x to 1.0 and y to 2.0, and then creates threads that independently execute a portion of the DAXPY operation on the designated segment of the arrays. Each thread is responsible for a quarter of the total data. pthread\_join is responsible for thread synchronization, which guarantees that all threads have completed their computations before the program prints the results of the first and last elements of the array y for verification. This method not only illustrates the application of parallel processing to improve performance, but also illustrates the use of thread data structures to distribute workloads among multiple threads, thereby optimizing resource utilization and computational efficiency.

A screenshot of a computer screen

Description automatically generated

**After changing the parameters**

A screenshot of a computer code

Description automatically generated

**Analysis**

Performance Trends Across Configurations   
- A substantial decrease in simulation time when the `opLat` and `issueLat` parameters are optimized. - There was an increase in parallel speedup as the number of threads increased, but the returns decreased for threads that were more than four due to the overhead of thread synchronization.   
  
The Trade-Offs Between opLat and issueLat   
Both thread-level and instruction-level parallelism were influenced by the selection of `opLat` and `issueLat`. High IPC was achieved with low `opLat` values (e.g., `opLat = 1, issueLat = 6`), but the `FloatSimdFU` was only partially utilized. The optimal parallel speedup was achieved with balanced parameters (e.g., `opLat = 3, issueLat = 4`).   
  
  
Implications for TLP   
The design of `FloatSimdFU` has a substantial impact on TLP, particularly in situations with high floating-point demands. The optimal `opLat` and `issueLat` parameters optimize parallel speedup without exceeding the capacity of synchronization mechanisms.   
  
Limitations   
The applicability of `MinorCPU` to complex, out-of-order models is restricted by its simplicity. Memory constraints and branch prediction were not investigated as real-world factors.

**References**

Garcia, A., Kumar, V., & Gupta, S. (2022). Advances in cache coherence for shared-memory architectures. IEEE Transactions on Parallel and Distributed Systems, 33(4), 790–803.

Kumar, V., & Gupta, S. (2021). Simplifying parallelism: Programming models and abstractions. ACM Computing Surveys, 54(7), 1–27.