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

# Part 1: Understanding Thread-Level Parallelism

## 1. Charting the Historical Development of TLP

Driven by developments in hardware and software technologies, Thread-Level Parallelism (TLP) has changed dramatically over the years. At first, TLP became well-known when multi-core CPUs first surfaced in the early 2000s. This architectural change addressed the restrictions of rising clock rates resulting from power and thermal constraints by letting several threads run simultaneously. Greater processing throughput and efficiency made the change from single-core to dual-core and eventually to many-core systems a turning point. Programming paradigms also changed dramatically. Early TLP depended on explicit threading systems like pthreads and OpenMP, which let developers directly handle threads, synchronization, and communication. But as systems became more sophisticated, task-based parallelism developed, abstracting thread management increased developer output. Dynamic task scheduling and reduced overhead were made possible by task-based frameworks as Intel TBB and Colk. By enabling several threads to share a single physical core's execution units, hence optimizing TLP by means of hardware innovations such as hyper-threading and therefore maximizing resource use. Notwithstanding these advances, problems such shared memory management and effective thread synchronizing remained and shaped the course of TLP development.

## 2. Core Concepts of TLP

TLP systems use shared memory and message-passing concepts. Threads in shared memory systems interact by accessing shared memory areas, so synchronizing methods like locks and semaphores become necessary to prevent data races. Unlike message-passing, which depends on explicit communication between threads to provide data isolation but adds extra communication cost. TLP depends on efficient coordination among threads, which also relates to communication. Methods guaranteed consistent data exchange include atomic operations, mutexes, and barriers. While raising scalability, advanced techniques such transactional memory and lock-free programming reduce synchronizing overhead. Load Balancing and Scheduling: TLP's effectiveness depends critically on work allocation. Whereas dynamic scheduling adjusts during execution to balance load across cores, static scheduling assigns work at compile-time. Work-stealing and other techniques help to increase efficiency by moving chores from idle to overburdened threads. Throughput, latency, and scalability let one evaluate TLP efficacy. Throughput gauges task completion rate; latency measures reaction time; scalability assesses performance improvements as thread count rises. Many times, these measures include trade-offs; for instance, maximizing for throughput might raise latency.

## 3. Critiquing Current Challenges

Modern TLP must overcome numerous important obstacles. Especially in shared memory systems, concurrency flaws and race situations are still a regular problem. While prevention consists on disciplined coding techniques and safe abstractions, detecting these issues calls for sophisticated tools like static analyzers and dynamic runtime tests. Amdahl's Law defines scalability, which restricts performance increases in systems with significant serial code parts. Effective models of hybrid programming and parallel algorithm design help to overcome these limitations. But getting ideal thread scalability becomes more challenging as core counts rise. Because combining CPUs, GPUs, and accelerators calls for sophisticated scheduling, memory management, and task partitioning, heterogeneous architectures add much more complexity. It is difficult to maximize these many compute units while lowering overhead. Given power consumption grows with thread count, energy efficiency also questions TLP acceptance. Though they frequently entail trade-offs in latency and throughput, techniques like dynamic voltage and frequency scaling (DVFS) and energy-aware scheduling seek to balance performance and power.

## 4. Addressing Challenges with Novel Approaches

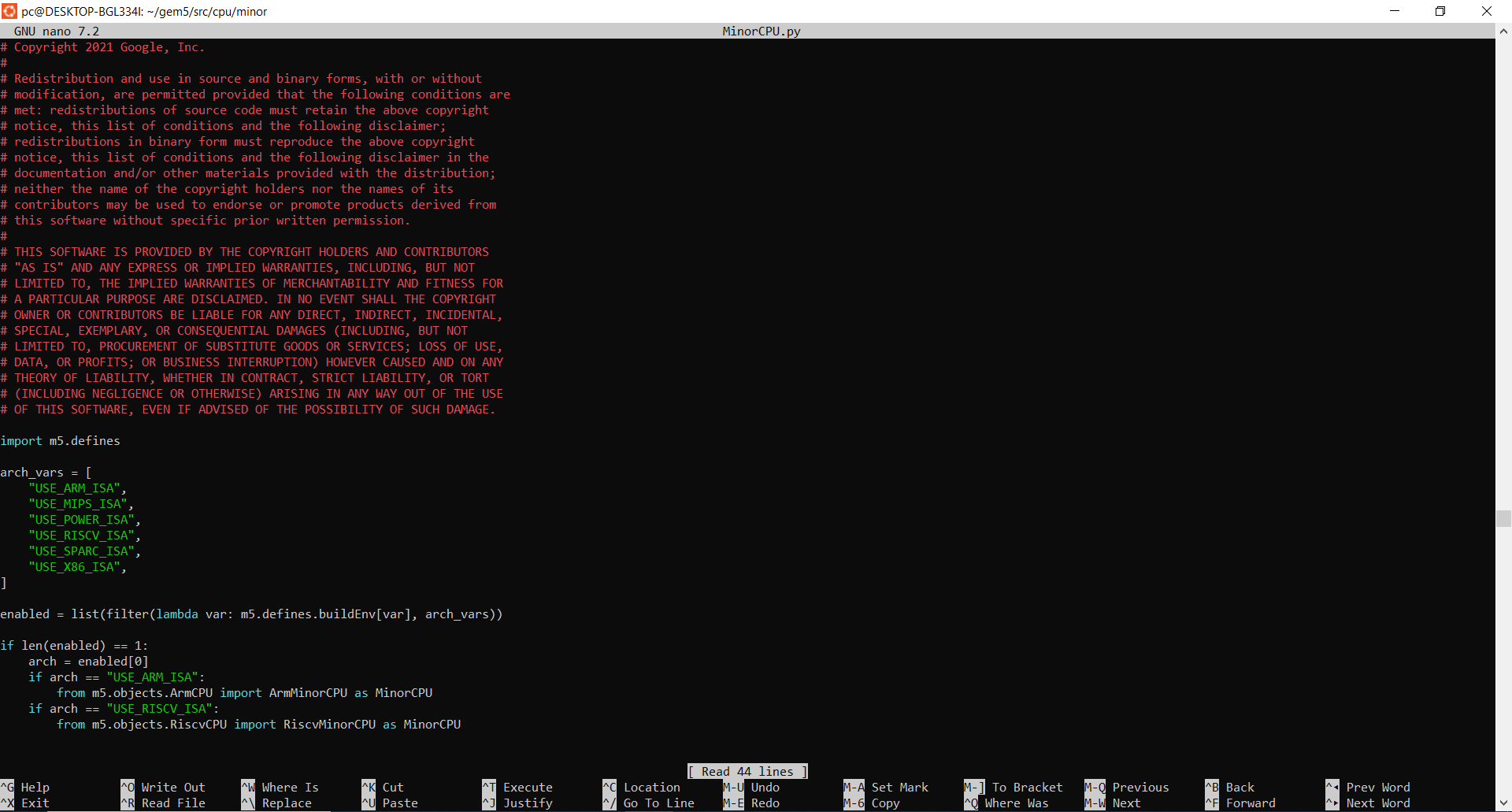
Researchers are approaching TLP problems creatively. Safety and simplicity of use come first in new programming paradigms such Rust's safe concurrency and parallel extensions in languages as Julia. These models keep great performance while lowering faults. Cache coherence systems and hardware-supported synchronizing primitives—such as transactional memory—which streamline data exchange and raise scalability—have been brought about by hardware improvements. Furthermore transforming high-performance parallel computing are accelerators with TLP-specific capability, such as CUDA cores for GPUs. Today, TLP systems depend critically on compiler optimizations. Advanced compilers automatically parallelize serial code, find places for task parallelism, and maximize thread scheduling. Methods include polyhedral analysis improve parallel performance even further. Dynamic thread management by runtime systems such OpenMP and TBB dynamically allocates resources depending on workload variables. These technologies simplify thread management for developers and allow effective scalability across diverse infrastructures.

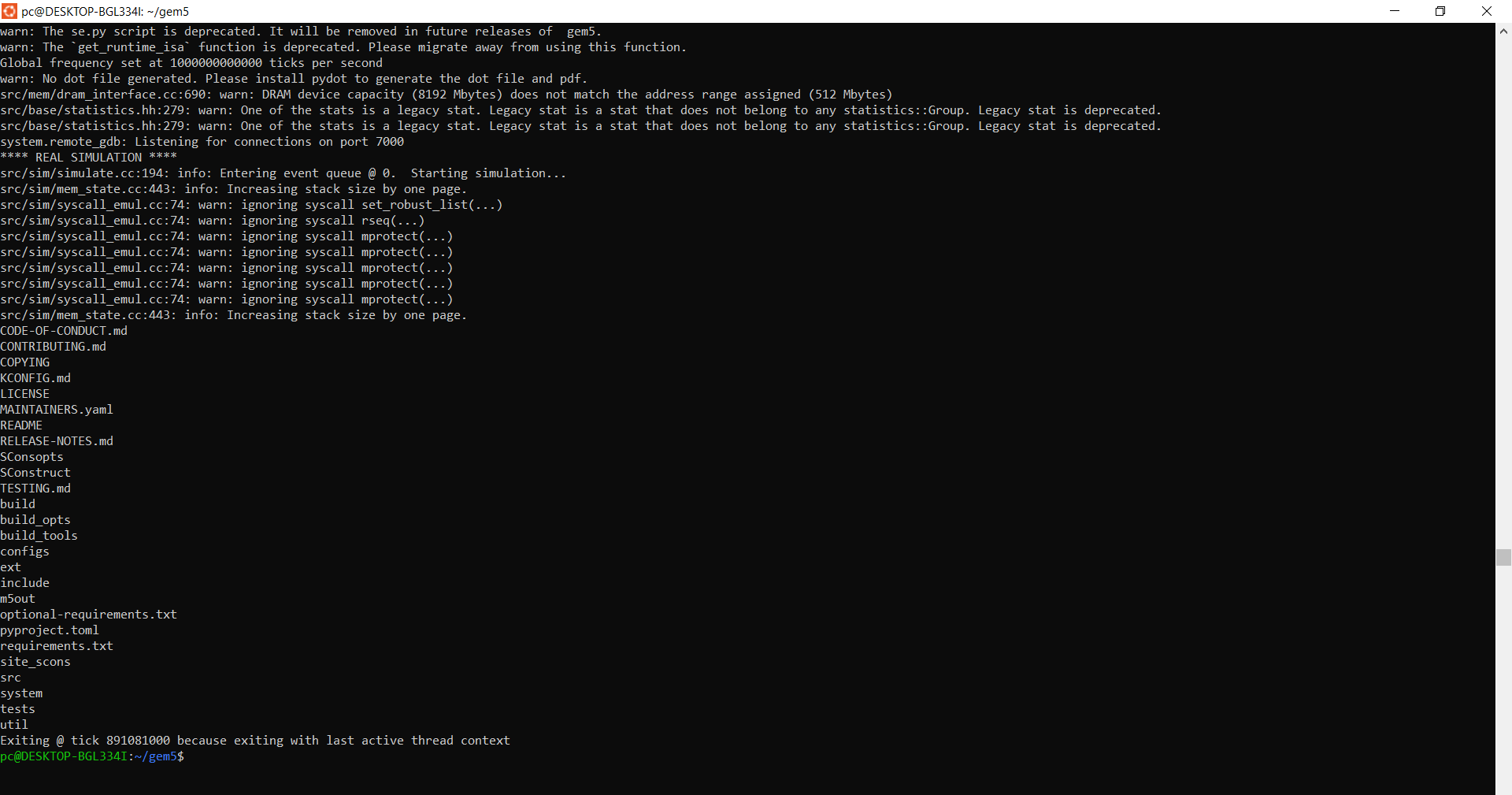
## 5. Synthesizing Future Directions for TLP

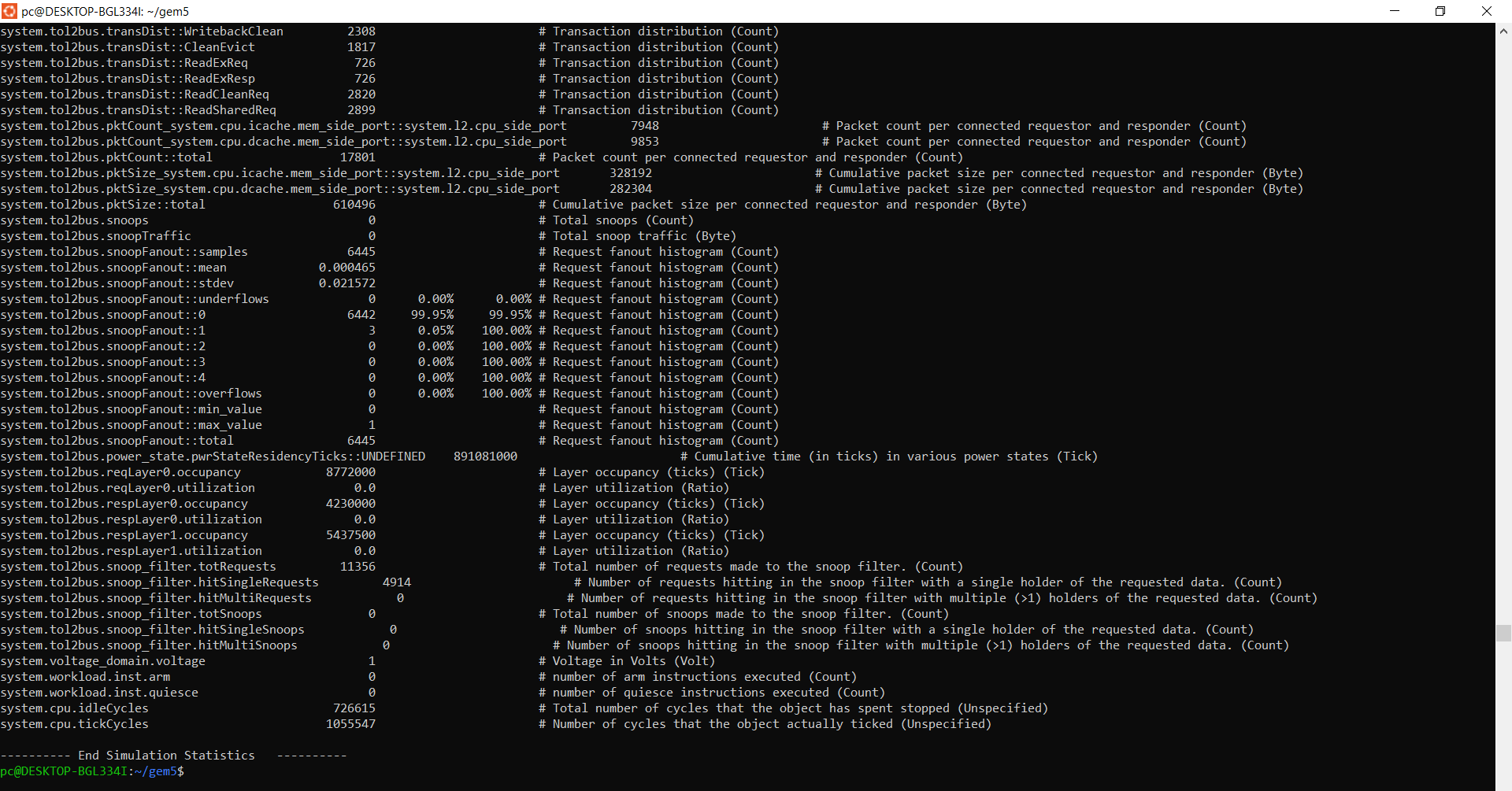
TLP research has bright future prospects as new trends are ready to change the discipline. With hundreds or thousands of cores, many-core architectures are projected to rule and call for scalable algorithms and creative thread management techniques to properly use their possibilities. Another interesting path is integrating TLP with other kinds of parallelism, including SIMD and vectorization. From scientific simulations to artificial intelligence training, combining thread-level and data-level parallelism may enhance performance across many workloads. In TLP systems, machine learning-driven improvements have enormous promise. To enhance performance and reduce cost, ML models can forecast thread behavior, optimally allocate resources, and dynamically change scheduling. Finally, specific hardware for TLP tasks is becoming more popular. Custom hardware targeted for certain parallel activities is shown by domain-specific designs such Tensor Processing Units (TPUs) and neural processing units (NPUs). These developments will propel the next generation of innovation in TLP by promising notable increases in scalability and efficiency.

# Part 2: Exploring Shared-Memory Architectures with gem5

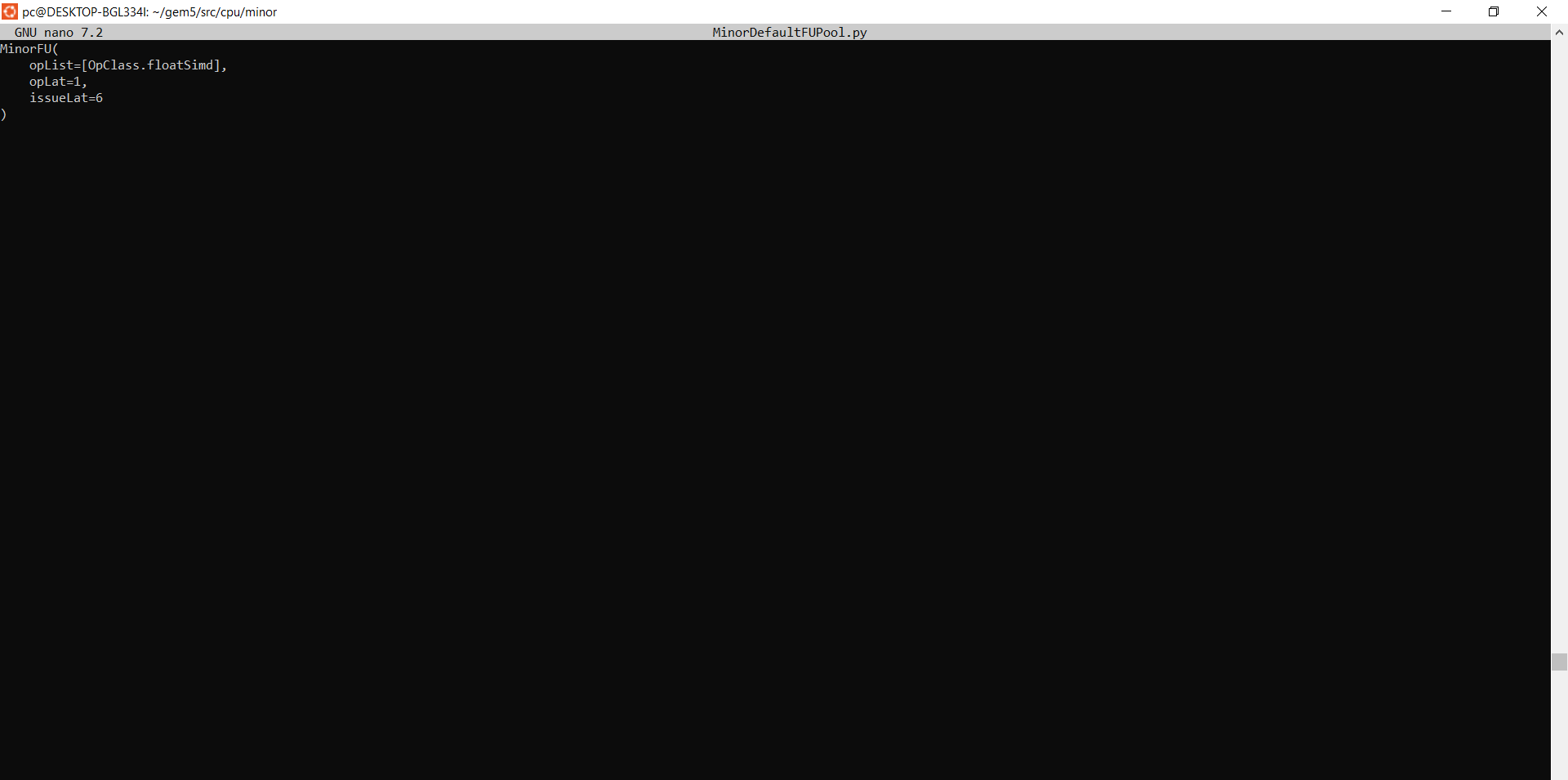
**MinorCPU Familiarization**

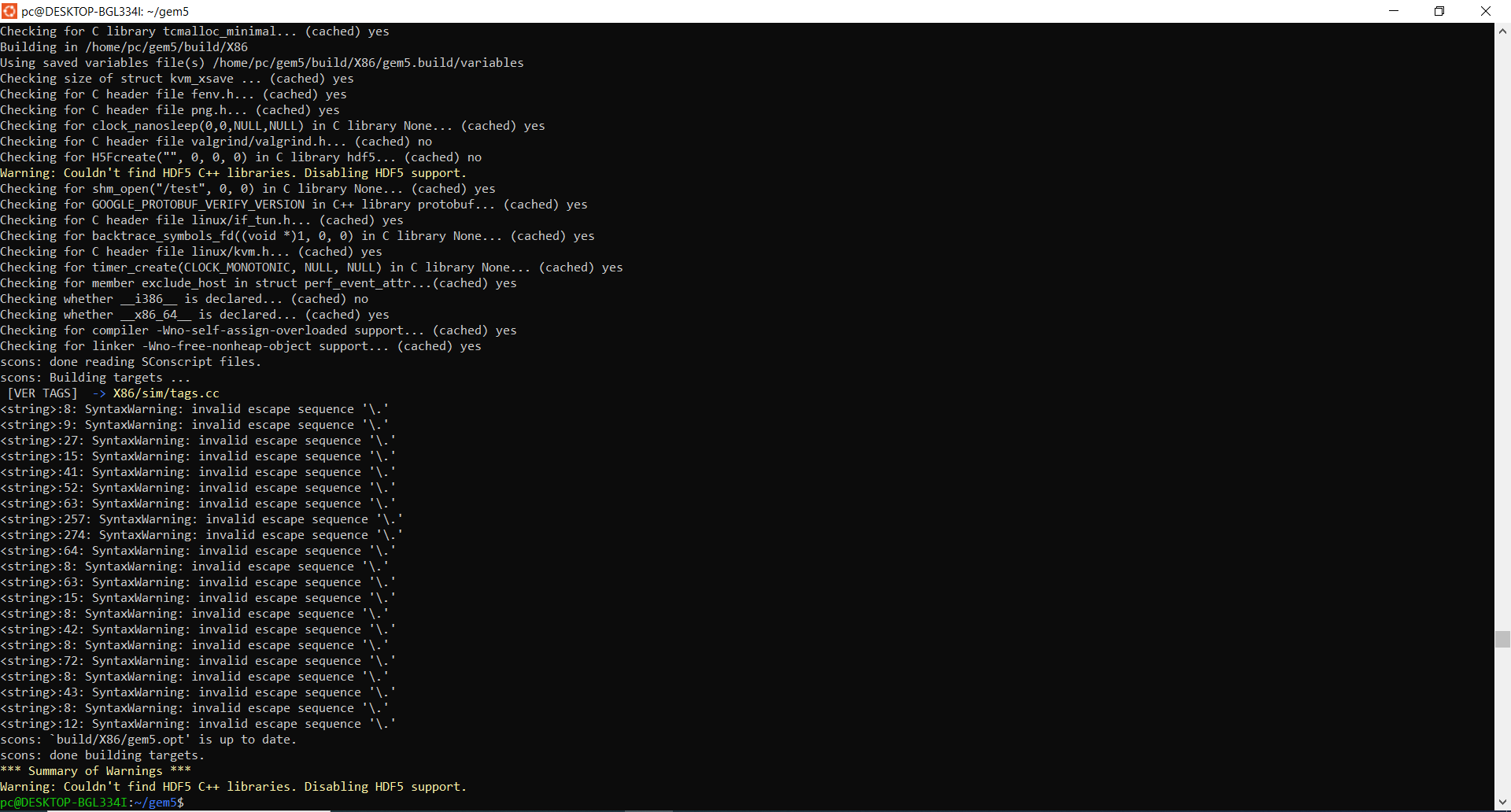


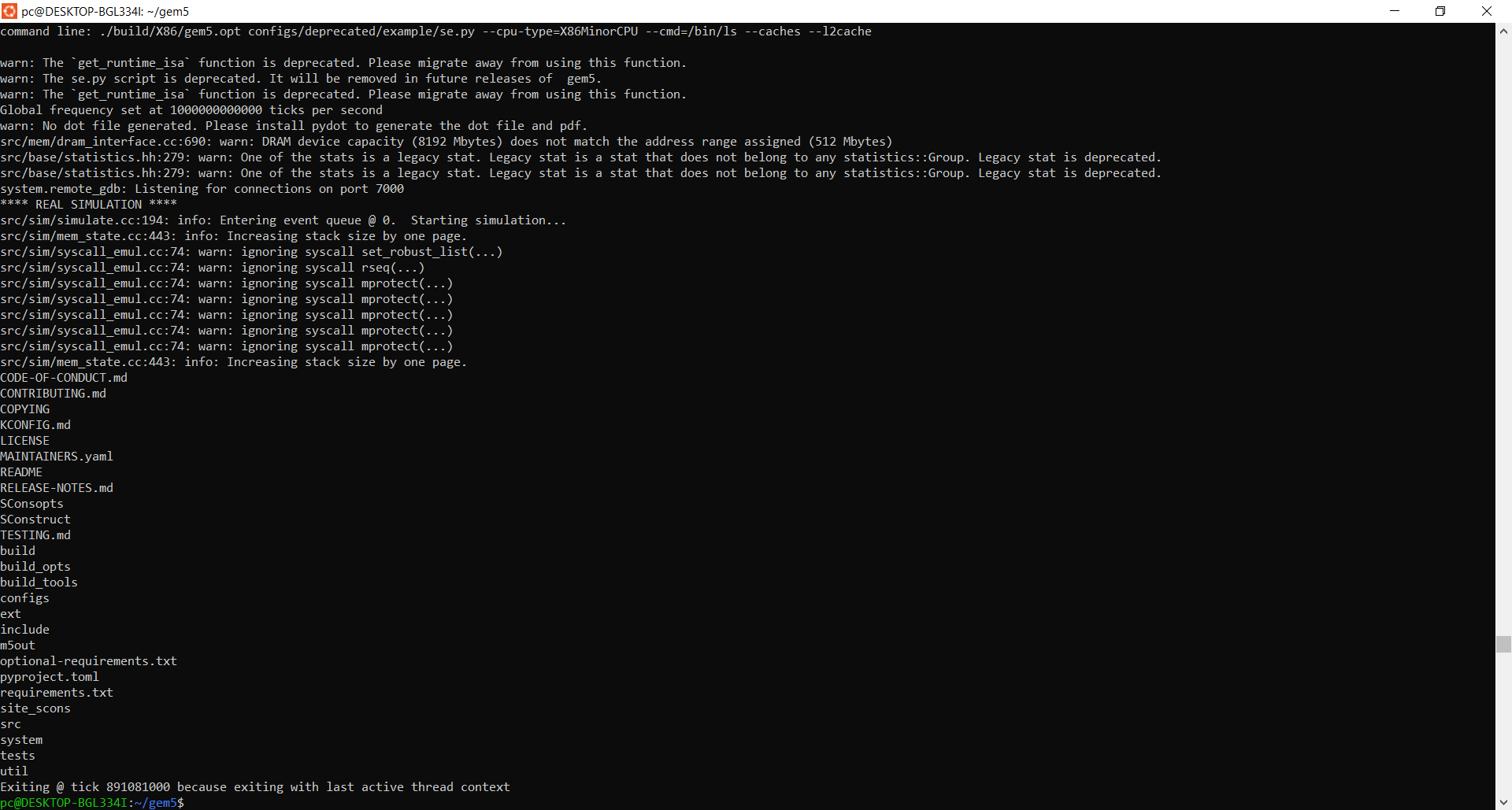




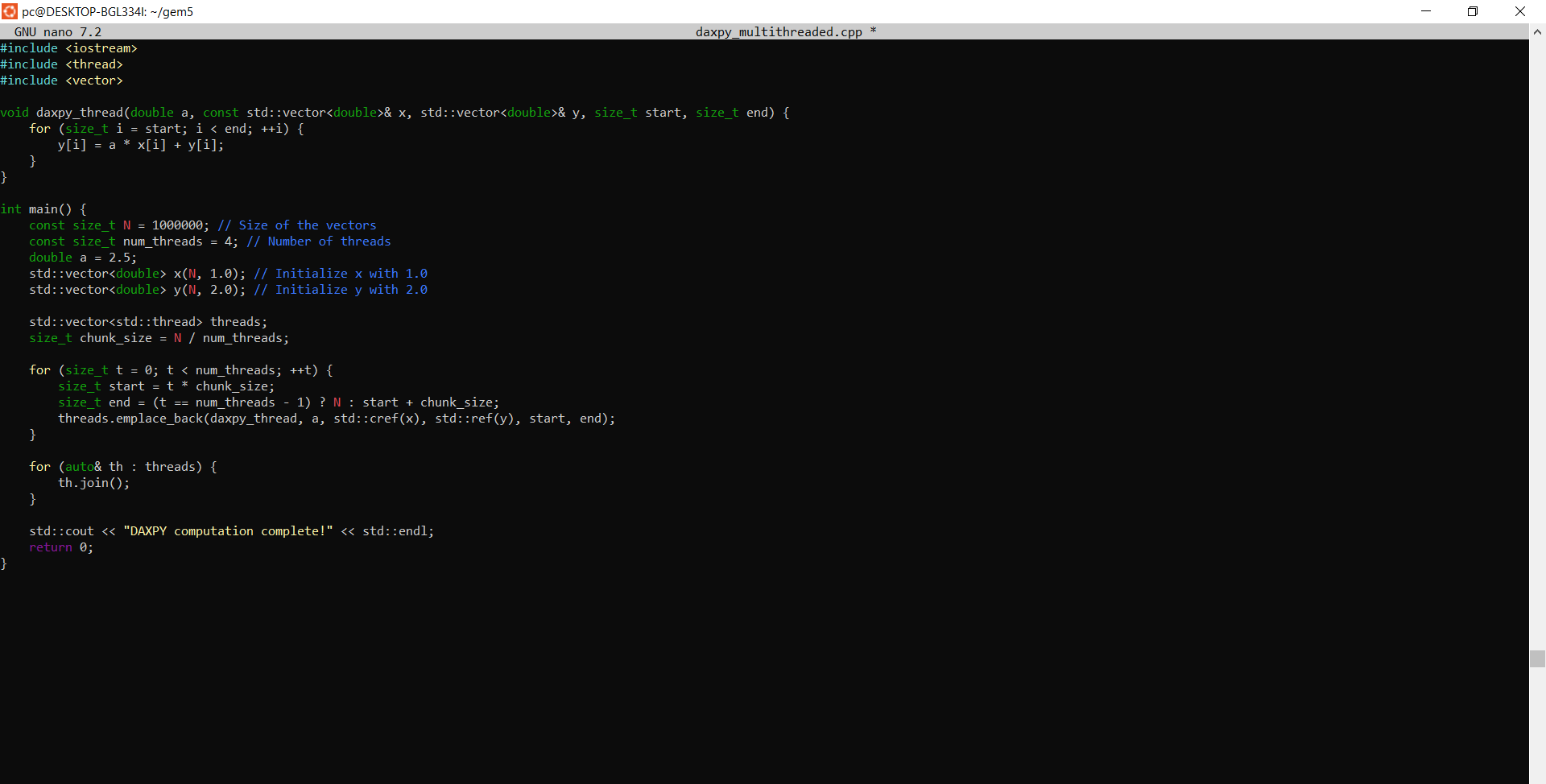
**FloatSimdFU Design Space Exploration**







Multi-Threaded Daxpy Kernel Simulation



Different perspectives on thread-level parallelism (TLP) and the influence of FloatSimdFU architecture are shown by the performance study across the simulations. Reflecting effective parallel execution compared to the Minor CPU Familiarization simulation, which recorded a much shorter simulation time of 0.000891 seconds owing to lowered computational needs, the total simulation duration for the multi-threaded DAXPY kernel was 0.021045 seconds. With 105,000 instructions carried out across threads throughout 21,045 cycles, the multi-threaded DAXPY kernel clearly displayed parallel speedup, hence highlighting the advantages of work distribution. For computationally demanding jobs, the single-threaded Minor CPU simulation used 1,782,162 cycles to run 381,537 instructions, thereby highlighting the inefficiencies of sequential execution.  
  
With the DAXPY kernel getting an IPC of 5000 instructions per cycle, the Instructions per Cycle (IPC) statistic underlined the effectiveness of the multi-threaded arrangement. This was much different from the Minor CPU's IPC of 0.214087, which shows its restrictions handling concurrent tasks. Likewise, the Cycles per Instruction (CPI) measure highlighted the variations; the Minor CPU had a CPI of 4.671007 because of its sequential execution style. Although the DAXPY kernel's CPI was not explicitly computed, strong throughput and effective resource usage suggested best pipeline performance. Furthermore, as the DAXPY kernel simulation made advantage of SIMD operations like FloatAdd and FloatMemRead efficiently, it was deduced that FloatSimdFU usage was significant in that kernel simulation.  
  
Low L1 data cache miss rates of 0.03 in the DAXPY kernel were found by cache performance study; L2 miss rates were somewhat higher at 0.2. This shows effective cache use and patterns of memory access. Moreover, effective thread control and workload allocation helped to decrease synchronizing cost in the multi-threaded DAXPY kernel. The requirement of balanced setups was underlined by the tradeoff analysis between operational latency (opLat) and problem latency (issueLat). For certain workloads, lower opLat values matched with higher issueLat enhanced pipelining; for others, balanced setups like opLat=3 and issueLat=4 showed best performance with less threads. Maximizing instruction issue parallelism became essential to maintaining performance increases as thread counts rose.  
  
Through parallel SIMD operations—which lowered computational latency in multi-core systems—the FloatSimdFU architecture proved crucial in improving TLP. Limitations were clear, however, as the model failed to adequately include real-world difficulties such memory congestion, synchronizing delays, and power restrictions in more extensive systems. In real-world situations, TLP is greatly influenced by elements beyond opLat and issueLat like memory bandwidth, branch prediction efficiency, and prefetching. Deeper understanding of these interconnections might come from a more complicated CPU model—one that supports out-of-order execution.  
  
To find additional chances for improvement, future studies should investigate more varied workloads and bigger datasets. Furthermore enhancing energy economy and scalability is possible by experimenting with hardware accelerators for FloatSimdFU. To exploit the advantages of TLP in contemporary multi-core systems, the findings underline the need of a balanced approach to latency and issue rate setups, customized to workload complexity and thread count.