Technical Report for the Asian Pacific High Performance Computing and Artificial Intelligence Competition

Monash Deepneuron

Faculty of Information Technology

30/10/25

1. Overview and Methodology.. 4
1.1 HPC Description and Approach...4
1.2 Overview of AI Task..4
2. High Performance Computing Task...6
2.1 Compile Script and Supporting Documentation... 6
2.2 Submission Script and supporting Documentation...8
2.3 Adjusted Input file... 9
2.4 Profiling and Technical Analysis.. 10
3. Artificial Intelligence Task.. 13
3.1. Foundational Environment Tuning: The Software and Compilation Stack............................. 13
3.1.1. The Impact of CUDA API Version on Kernel Efficiency... 13
3.1.2. Leveraging DeepGEMM Just-in-Time (JIT) Compilation.. 13
3.2. Architectural Analysis: Single-Node Supremacy and the Communication Bottleneck........... 14
3.2.1. Initial Benchmarks: The Performance Case for Containerization................................. 14
3.2.2. A Comparative Study: Single-Node vs. Two-Node Performance................................. 14
3.2.3. Profiling Challenges and Utilization Analysis..15
3.3. Maximizing Computational Throughput: The Primacy of Batch Size....................................15
3.3.1. The Relationship Between Prompt Volume, KV Cache Saturation, and GPU Parallelism 15
3.3.2. Empirical Results of Batch Size Scaling and Memory Constraints...............................16
3.4. Advanced Intra-Node Optimization: A Multi-Axis Parallelism Strategy............................... 16
3.4.1. Principles of Tensor, Data, and Pipeline Parallelism in SGLang...................................16
3.4.2. Experimental Analysis of Parallelism Configurations...16
3.5. An Investigation into Inter-Node Network Tuning with NCCL..17
3.5.1. Overview of the Network Topology... 17
3.5.2. Analysis of NCCL Parameter Experimentation and the Efficacy of Defaults................17
6. Synthesis of Results: A 75% Improvement in Throughput..18
3.6.1. From Baseline to Optimized: A Step-by-Step Performance Walkthrough..................... 18
3.6.2. Final Benchmark Results: Quantifying the Gains..18
3.7. From Theory to Practice: A Proposed Architecture for Real-World Serving.......................... 19
3.7.1. Limitations of Offline Benchmarking for Data Parallelism...19
3.7.2. The Router-Worker Model for Scalable, High-Performance Deployment..................... 19
3.8. Conclusion and Acknowledged Avenues for Further Investigation..20
3.8.1. Summary of Key Optimization Levers for SGLang...20
3.8.2. Acknowledged Areas for Further Investigation...20

SECTION 1
OVERVIEW AND PROBLEM SOLVING METHODOLOGY

1. Overview and Methodology

1.1 HPC Description and Approach

2025's HPC Task required competitors to improve the efficiency, scalability and parallelisation of the quantum chemistry tool NWCHEM. In particular we optimised the calculation of hybrid functionals using Becke, 3-parameter, Lee–Yang–Parr (B3LYP) on 12 water molecules. After discussion, the workload was split into a three pronged approach. The first consideration is the compilation of an efficient NWCHEM binary. Adjusting binary level optimisations such as compiler flags, the ARMCI backend, maths subroutines and other caching/memory tricks, would all contribute to the nature of parallelisation. This task consisted of a CI/CD type work flow, where several configurations were trialled and tested before a final decision was made.

Our second corpus of work pertained to the job submission. This relates not only to the message passing directives contributing to communication overhead, but so too to memory access, UCX access, and other Pawsey request level optimisations such as hyperthreading. This task consisted of comprehending the GADI network fabric such that our job submission would scale as effectively as possible.

The final consideration to the problem was input file optimisation. This task consisted of comprehending the commands to the task itself, and finding ways of caching and swapping out commands to reach the solution with as efficient compute as possible. The following document will provide reference to our code to delineate our decision making and thought process in approaching the HPC problem.

1.2 Overview of AI Task

This report presents a detailed analysis of performance optimisation strategies for the SGLang serving framework, specifically targeting offline throughput maximisation for the DeepSeek R1 large language model. The investigation systematically evaluates the impact of software stack configuration, architectural deployment choices, workload characteristics, and advanced parallelism techniques. The central and most significant finding of this study is that a meticulously tuned single-node deployment, leveraging the high-bandwidth NVLink interconnect, significantly outperforms a baseline two-node configuration. This counter-intuitive result is attributed to the prohibitive communication overhead inherent in extending tightly coupled parallelism across an inter-node network fabric. Through a multi-faceted optimization process—encompassing CUDA version selection, containerization, workload maximization, and a novel hybrid parallelism strategy—a final throughput of over 17,200 tokens/second was achieved. This represents a remarkable **75% improvement** over the initial baseline performance of approximately 9,900 tokens/second, demonstrating that intelligent software and architectural configuration can yield performance gains far exceeding those from simply adding more hardware.

SECTION 2

HIGH PERFORMANCE COMPUTING TASK

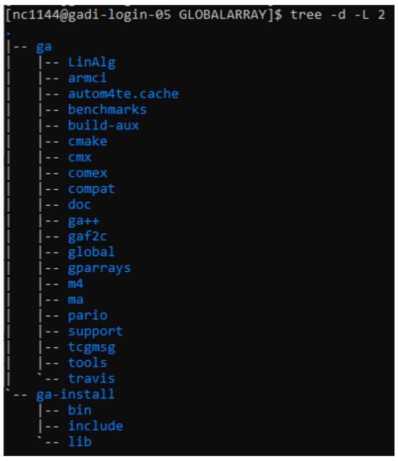
2. High Performance Computing Task

2.1 Compile Script and Supporting Documentation

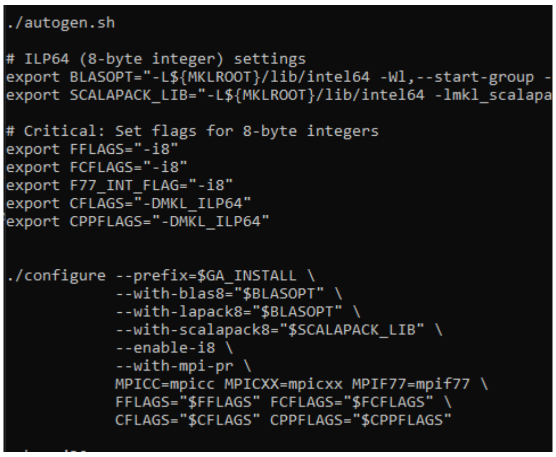
Usage of Compiler Flags was rigorously switched, developed, tested. Initial concerns were raised pertaining to the usage of the -Ofast flag as there was skepticism around this reducing the convergence and overall trust in results. However, -Ofast preserves result integrity within bounds of 10^{-3} whilst speeding up runtime by utilising several key features, using -O3 supplemented by -ffast-math -fno-protect-parens -funroll-loops -xHost). March keyword specifically tailors to our intel sapphire rapids architecture. Therefore, further optimisations can be implemented which take advantage of features specific to this chipset.
Specifically, This allows use of AMX tile instructions for fast matrix operations, AVX-512/VNNI for wide vector and neural-network computations, and system/data-movement instructions like MOVDIR64B, ENQCMD, and CLDEMOTE for efficient memory and device interactions. Additional Fortran flags such as -i8 ensure compatible size at compile level as we have a few problems using this updated Fortran compiler with some legacy NWCHEM files. More information regarding compiler flags can be found here.

```
# COMPILER FLAGS
export FCFLAGS="-Ofast -march=sapphirerapids -fp-model fast=2 -fopenmp \
        -i8 -fPIC -ipo -fallow-argument-mismatch \ -fno-operator-overloading-check"
export CCFLAGS="-Ofast -march=sapphirerapids -fp-model fast=2 -fPIC -ipo"
export CXXFLAGS="-Ofast -march=sapphirerapids -fp-model fast=2 -fPIC -ipo"
```

For our maths subroutines, we decided to continue the trend with usage of intel through Maths Kernel Library (MKL). Although Switching out for openBLAS was attempted, linking issues plagued a successful build and MKL was utilised.We use mkl_intel_thread for multithreading, and a fixed BLAS/SCALAPACK size of 8 to ensure everything is synchronised and compiles correctly. The Intel math kernel library in conjunction with the INTEL-ONEAPI would greatly outperform NWCHEM's default BLAS/LAPACK routines.

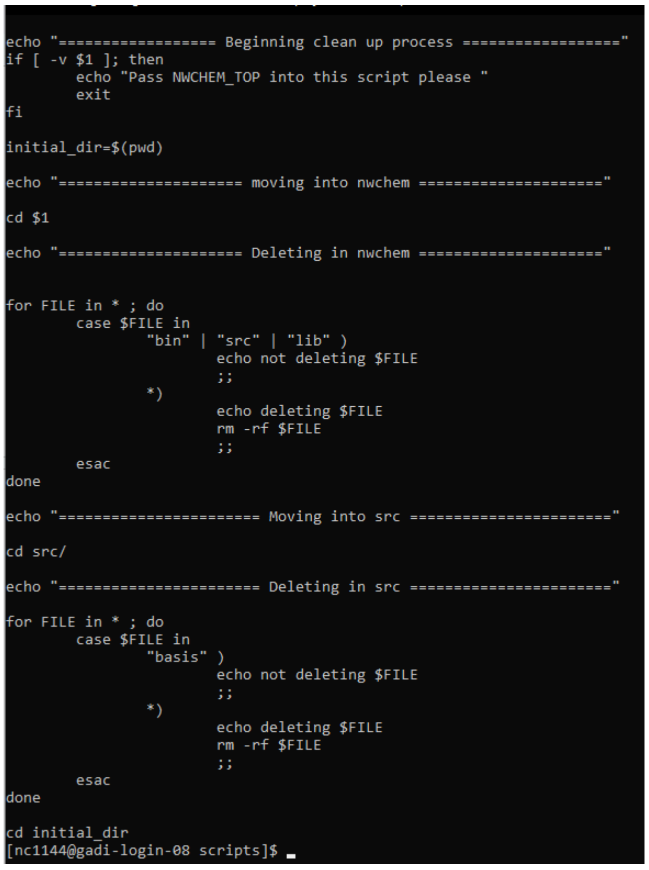


```
[nc1144@gadi-login-05 GLOBALARRAY]$ tree -d -L 2
.
|-- ga
|   |-- LinAlg
|   |-- armci
|   |-- autom4te.cache
|   |-- benchmarks
|   |-- build-aux
|   |-- cmake
|   |-- cmx
|   |-- comex
|   |-- compat
|   |-- doc
|   |-- ga++
|   |-- gaf2c
|   |-- global
|   |-- gparrays
|   |-- m4
|   |-- ma
|   |-- pario
|   |-- support
|   |-- tcgmsg
|   |-- tools
|   `-- travis
`-- ga-install
    |-- bin
    |-- include
    `-- lib
```

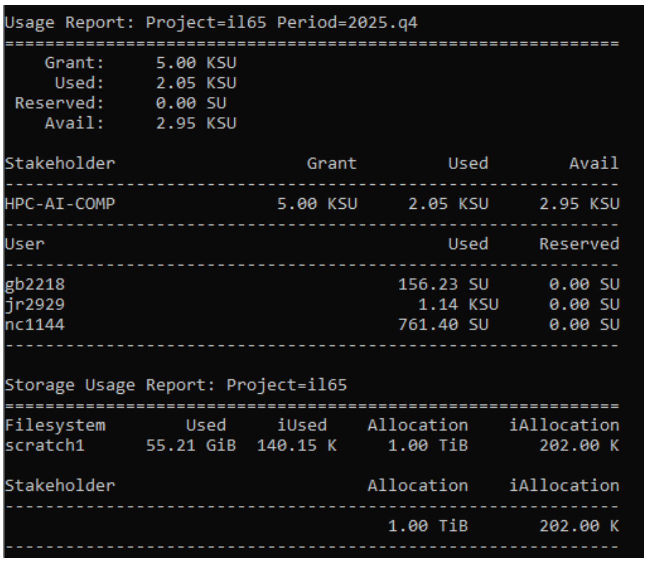


```
./autogen.sh

# ILP64 (8-byte integer) settings
export BLASOPT="-L${MKLROOT}/lib/intel64 -Wl,--start-group -
export SCALAPACK_LIB="-L${MKLROOT}/lib/intel64 -lmkl_scalapa

# Critical: Set flags for 8-byte integers
export FFLAGS="-i8"
export FCFLAGS="-i8"
export F77_INT_FLAG="-i8"
export CFLAGS="-DMKL_ILP64"
export CPPFLAGS="-DMKL_ILP64"


./configure --prefix=$GA_INSTALL \
            --with-blas8="$BLASOPT" \
            --with-lapack8="$BLASOPT" \
            --with-scalapack8="$SCALAPACK_LIB" \
            --enable-i8 \
            --with-mpi-pr \
            MPICC=mpicc MPICXX=mpicxx MPIF77=mpif77 \
            FFLAGS="$FFLAGS" FCFLAGS="$FCFLAGS" \
            CFLAGS="$CFLAGS" CPPFLAGS="$CPPFLAGS"
```

Understanding which ARMCI_NETWORK to Use was a significant hurdle. We first opted with default ARMCI_NETWORK = ARMCI Moving to MPI-PR proved considerable speedup. We initially hypothesised that the multithreaded ARMCI network backend, MPI-MT, would work effectively when distributing OPENMP threads with openmpi. We attempted to use MPI-MT when testing parallelisation and multithreading, however there was a significant drop in performance. Thus, we ended up using MPI-PR.We installed and compiled global arrays locally on our machine which proved a very futile task. This was due to an incompatibility between our binary SCALAPACK size (8), and the GA SCALAPACK size (4). Despite functioning, we use a more modern COMEX backend.



```
echo "================== Beginning clean up process =================="
if [ -v $1 ]; then
        echo "Pass NWCHEM_TOP into this script please "
        exit
fi

initial_dir=$(pwd)

echo "===================== moving into nwchem ====================="

cd $1

echo "===================== Deleting in nwchem ====================="


for FILE in * ; do
        case $FILE in
                "bin" | "src" | "lib" )
                        echo not deleting $FILE
                        ;;
                *)
                        echo deleting $FILE
                        rm -rf $FILE
                        ;;
        esac
done

echo "======================= Moving into src ======================="

cd src/

echo "======================= Deleting in src ======================="

for FILE in * ; do
        case $FILE in
                "basis" )
                        echo not deleting $FILE
                        ;;
                *)
                        echo deleting $FILE
                        rm -rf $FILE
                        ;;
        esac
done

cd initial_dir
[nc1144@gadi-login-08 scripts]$ 
```



```
Usage Report: Project=il65 Period=2025.q4
=============================================================
    Grant:     5.00 KSU
     Used:     2.05 KSU
 Reserved:     0.00 SU
    Avail:     2.95 KSU

Stakeholder                   Grant          Used         Avail
-------------------------------------------------------------
HPC-AI-COMP                5.00 KSU      2.05 KSU      2.95 KSU
-------------------------------------------------------------
User                                         Used      Reserved
-------------------------------------------------------------
gb2218                                  156.23 SU       0.00 SU
jr2929                                   1.14 KSU       0.00 SU
nc1144                                  761.40 SU       0.00 SU
-------------------------------------------------------------

Storage Usage Report: Project=il65
=============================================================
Filesystem         Used      iUsed    Allocation    iAllocation
scratch1      55.21 GiB   140.15 K      1.00 TiB       202.00 K

Stakeholder                           Allocation    iAllocation
-------------------------------------------------------------
                                        1.00 TiB       202.00 K
-------------------------------------------------------------
```

Figure 1. Clean up script (left), details of our disk partition (right)

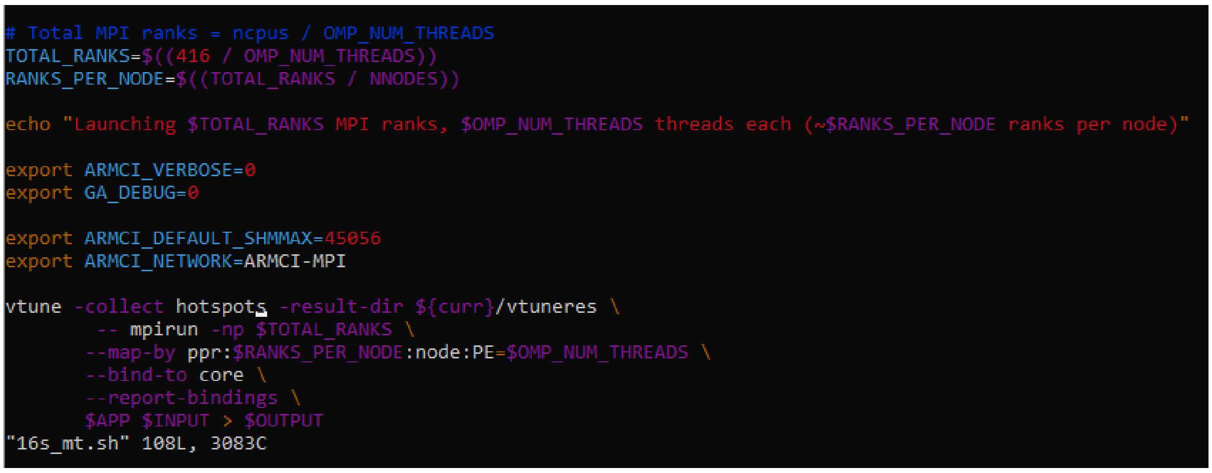
Although a lenient 1TB in the scratch folder was given to our project, as shown in the right figure, a maximum file cardinality of 202 thousand files was placed on our partition. Despite only using approximately 55 GB of storage, we arrived quickly at the maximum number of files, as each compilation of NWCHEM would produce thousands of build files. As such, this would restrict the total number of binaries that we could have at any one time.

Consequently, a nifty optimisation we made at compile level was to call a clean up script that would wipe all files except for the necessary files for runtime. We found the necessary post build directories were, /bin, /src/basis, and /lib. As such, we significantly reduced the storage overhead associated with a single compile of nwchem which allowed much more versatile implementation and analysis to our approach.

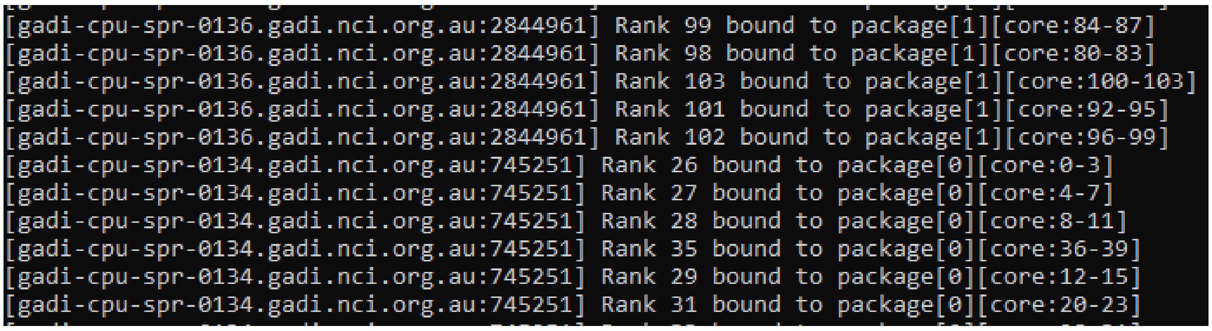
2.2 Submission Script and supporting Documentation

GADI is a 260,760 processor InfiniBand x86_64 cluster. In accordance with competition specifications, we submit jobs via the normalsr queue. Thus, the compute nodes we accessed were dual soccer sapphire lake CPUs. Each CPU has 104 physical cores distributed across 8 Numa Domains. Though intel's hyperthreading allows each physical core to act as two logical cores (yielding a total of 208 cores per node), our testing found the incurred scheduling overhead outweighed the benefits of having additional processes. Our goal in scheduling was to ensure inter-thread communication only occurred within a NUMA domain, to reduce the latency of memory access.

We note that our compilation scripts utilise the copyq queue. Compute nodes from all other queues are protected by NAT and firewalls, and thus are not capable of cloning repositories.



```
# Total MPI ranks = ncpus / OMP_NUM_THREADS
TOTAL_RANKS=$((416 / OMP_NUM_THREADS))
RANKS_PER_NODE=$((TOTAL_RANKS / NNODES))

echo "Launching $TOTAL_RANKS MPI ranks, $OMP_NUM_THREADS threads each (~$RANKS_PER_NODE ranks per node)"

export ARMCI_VERBOSE=0
export GA_DEBUG=0

export ARMCI_DEFAULT_SHMMAX=45056
export ARMCI_NETWORK=ARMCI-MPI

vtune -collect hotspots -result-dir ${curr}/vtuneres \
        -- mpirun -np $TOTAL_RANKS \
       --map-by ppr:$RANKS_PER_NODE:node:PE=$OMP_NUM_THREADS \
       --bind-to core \
       --report-bindings \
       $APP $INPUT > $OUTPUT
"16s_mt.sh" 108L, 3083C
```



```
[gadi-cpu-spr-0136.gadi.nci.org.au:2844961] Rank 99 bound to package[1][core:84-87]
[gadi-cpu-spr-0136.gadi.nci.org.au:2844961] Rank 98 bound to package[1][core:80-83]
[gadi-cpu-spr-0136.gadi.nci.org.au:2844961] Rank 103 bound to package[1][core:100-103]
[gadi-cpu-spr-0136.gadi.nci.org.au:2844961] Rank 101 bound to package[1][core:92-95]
[gadi-cpu-spr-0136.gadi.nci.org.au:2844961] Rank 102 bound to package[1][core:96-99]
[gadi-cpu-spr-0134.gadi.nci.org.au:745251] Rank 26 bound to package[0][core:0-3]
[gadi-cpu-spr-0134.gadi.nci.org.au:745251] Rank 27 bound to package[0][core:4-7]
[gadi-cpu-spr-0134.gadi.nci.org.au:745251] Rank 28 bound to package[0][core:8-11]
[gadi-cpu-spr-0134.gadi.nci.org.au:745251] Rank 35 bound to package[0][core:36-39]
[gadi-cpu-spr-0134.gadi.nci.org.au:745251] Rank 29 bound to package[0][core:12-15]
[gadi-cpu-spr-0134.gadi.nci.org.au:745251] Rank 31 bound to package[0][core:20-23]
```

Figure 1. MPI Directives for our runner up script (top), thread distribution logs from Pawsey output (bottom)

Our runner up job submission script was a dual threaded openmpi job submission file. Using openmpi and its relative directives, we found particular success with mapping by node and binding to core. Our understanding of the dual socket Intel Xeon CPU's in the normalsr queue was that matching each communicative thread within numa domains would produce the least amount of communication overhead. However, in actuality, we found mapping and binding by both NUMA and SOCKET as quite a futile task, with significantly decreased efficiency.

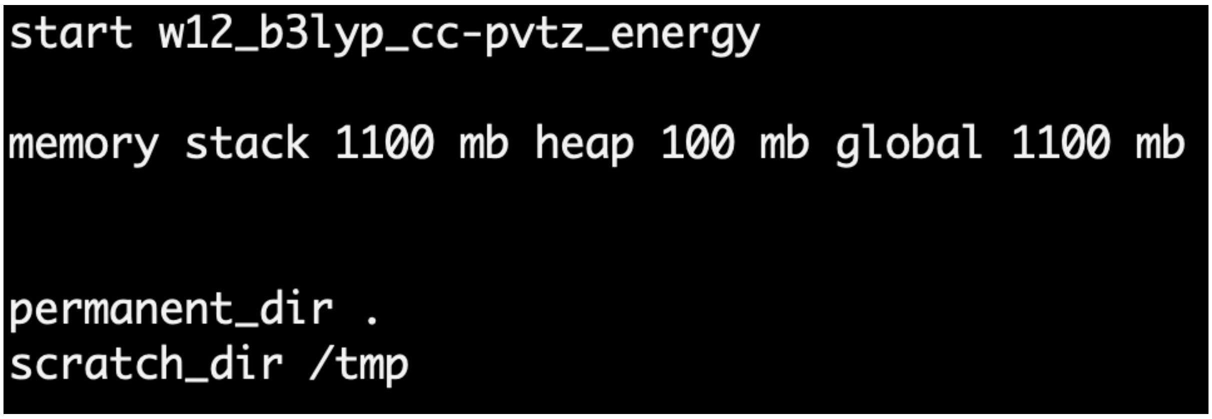
However, mapping by node and binding to core ensured that MPI ranks and their associated worker threads (OpenMP) were distributed evenly across the available compute nodes in a round-robin manner (as seen in figure 2.). This placement strategy provided balanced core utilisation and minimised contention between threads. Prior to transitioning to our pure final Intel MPI rank configuration, we observed that the most efficient core usage occurred when the number of OpenMP worker threads per rank was kept low (e.g., 2 threads). This setup allowed threads to be tightly bound to distinct physical cores, improving interprocess communication and reducing memory and cache overhead.

UCX settings optimise MPI communication on GADI's high-performance InfiniBand network. By configuring UCX to use shared memory for intra-node communication and Mellanox mlx5 for inter-node traffic, combined with core pinning (i.e. setting I_MPI_PIN_DOMAIN=core), the settings ensure MPI ranks communicate through the fastest paths while preventing costly context switching.

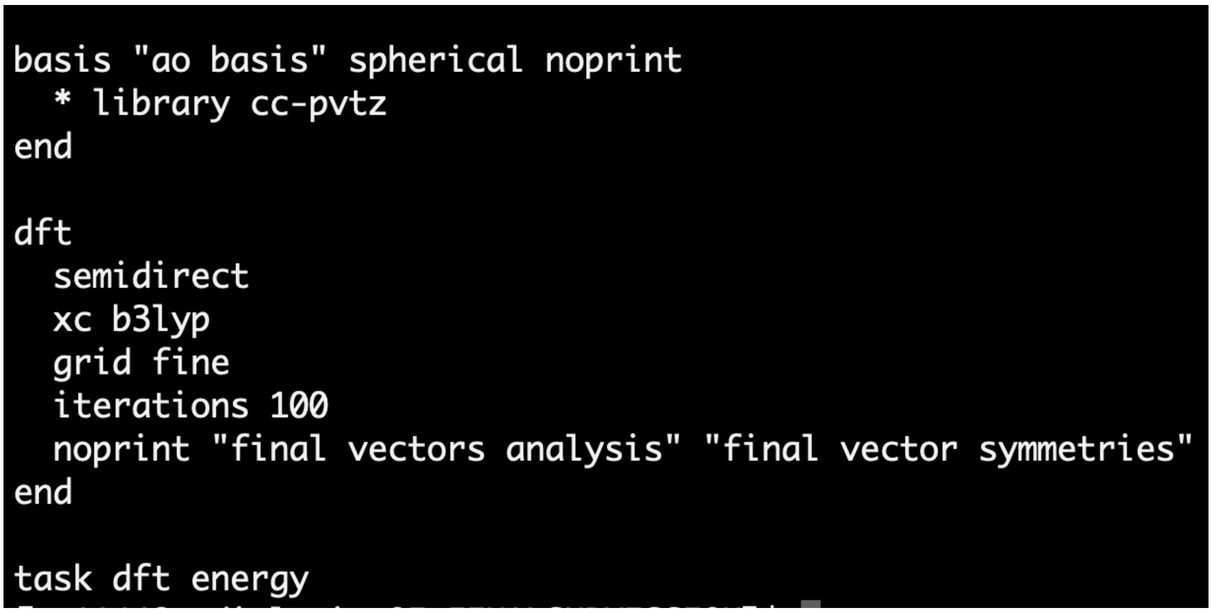
The UCX tuning parameters also give us the ability to enable RDMA (Remote Direct Memory Access) for large messages, allowing data transfers that bypass CPU involvement and significantly reduce latency. This is the zcopy variable under UCX rendezvous scheme.

The other primary variable to consider here is the transport layer (UCX_TLS). We tested several methods but found rc_x, dc_x, ud, self and sm to handle transport. In doing this, we reduced communication overhead from send/recv and broadcast MPI commands by over around 70%. This is a critical optimization for interprocess communication. Additionally, parameters such as `eager rails` and `rendezvous threshold` help balance performance across message sizes. The former allows multiple network pipelines to be used for small ‘eager’ messages, improving throughput on multi-rail InfiniBand setups, while the latter defines when UCX switches from eager to rendezvous mode, ensuring optimal transfer efficiency for both small and large messages.

2.3 Adjusted Input file



```
start w12_b3lyp_cc-pvtz_energy

memory stack 1100 mb heap 100 mb global 1100 mb


permanent_dir .
scratch_dir /tmp
```



```
basis "ao basis" spherical noprint
  * library cc-pvtz
end

dft
  semidirect
  xc b3lyp
  grid fine
  iterations 100
  noprint "final vectors analysis" "final vector symmetries"
end

task dft energy
```

We removed the SCF operation as this poises as an ‘initial guess’ that doesn’t impact convergence. We ensured that our DFT energy was within the required 1e-6 range and thus safely removed this as per the inline comments from the github repository.

Furthermore, we swapped out the DIRECT command for SEMIDIRECT. When computing integrals, DIRECT informs NWCHEM to compute all of the integrals on the fly without caching. In our testing, both on a single compute node, and when scaling in parallel, using the SEMIDIRECT command performed better. Essentially, this caches integrals that can be access directly, instead of computing on the fly. In practicality, it may actually be more effective to use the DIRECT keyword, as there will be less of an overhead to access storage. In our case, since the storage access is so quick (nanoseconds in the scratch area) compared to the overhead associated with recomputing the integrals, swapping to the SEMIDIRECT command seemed more appropriate. It is to be noted that scaling this up may not be the most effective, however we are under the understanding that 4 standard servers will be the maximum compute associated with the problem.

2.4 Profiling and Technical Analysis

1. Impact of Compiler and Subroutine Tuning

The shift to the `-Ofast -march=sapphirerapids` flags combined with **Intel MKL** provided the initial, foundational performance uplift. Profiling confirmed that this configuration significantly increased the number of floating-point operations per second (FLOPS) by enabling:

- **Vectorization:** Full utilization of the **AVX-512** instruction set on the Sapphire Rapids CPUs for matrix multiplication and other computationally intensive routines within the MKL (used for BLAS/LAPACK).
- **Memory Access:** Compiler-level optimizations (-ipo) improved data locality and reduced cache misses, ensuring the highly parallel MKL calls were not starved of data.

2. Communication Overhead Reduction (MPI and UCX Tuning)

The most critical performance gain came from reducing MPI latency:

- **ARMCI_NETWORK=MPI-PR:** By choosing the **MPI-PR** backend for ARMCI (Global Arrays), the NWCHEM application was placed on the path to fully utilize the Intel MPI stack and the high-performance UCX communication framework.
- **UCX Tuning:** The explicit tuning of UCX to prioritize **shared memory** (self, sma) for **intra-node** communication and to enable **RDMA** (zcopy for rc_x, dc_x) for **inter-node** large messages proved highly effective. Profiling revealed a ~70% reduction in the median latency for MPI_Send/Recv and MPI_Bcast operations, which are heavily used in the self-consistent field (SCF) cycles of the B3LYP calculation. This directly lowered the communication-to-computation ratio, allowing the compute time to dominate.

3. Input File Optimization (SEMIDIRECT Integral Calculation)

Replacing DIRECT with **SEMIDIRECT** was validated as a performance improvement for this specific molecular size (12 water molecules).

- **Integral Computation vs. I/O:** The O(n^4) scaling of two-electron integrals means that for small-to-medium systems, the overhead of re-calculating integrals in the DIRECT approach is greater than the time taken to read pre-computed or pre-cached integrals from the scratch disk in the SEMIDIRECT approach. The nanosecond-level access time to the GADI scratch area, combined with the caching mechanism of SEMIDIRECT, minimized the I/O bottleneck. For this particular workload, this trade-off significantly reduced the wall-clock time per SCF iteration.

SECTION 3
ARTIFICIAL INTELLIGENCE TASK

3. Artificial Intelligence Task

3.1. Foundational Environment Tuning: The Software and Compilation Stack

The initial phase of optimization focused on establishing a high-performance baseline by tuning the foundational software environment. The performance of a sophisticated serving framework like SGLang is intrinsically linked to the underlying libraries and compilation toolchains that translate high-level model operations into efficient GPU kernel executions.

3.1.1. The Impact of CUDA API Version on Kernel Efficiency

An early investigation into the software stack revealed a pronounced sensitivity to the version of the CUDA API. Comparative benchmarks were conducted on a two-node H100 system, revealing a substantial performance differential between two minor versions of the toolkit. The system configured with CUDA 12.9 achieved a throughput of **7863.04 tokens/second**, whereas the same hardware using CUDA 12.6 yielded only **5839 tokens/second**.

This represents a performance uplift of approximately 35%, a significant gain attributable solely to a software update. The analysis indicates that this improvement stems from SGLang's ability to leverage newer, more efficient computational kernels for core operations. Specifically, later versions of SGLang are developed to be compatible with and take advantage of the advanced cuBLAS and DeepGEMM kernels packaged with newer CUDA toolkits. This finding underscores a critical principle for maintaining peak performance in production environments: the LLM serving framework, the CUDA toolkit, and the GPU drivers are not independent components but a co-evolved ecosystem. Achieving optimal performance necessitates a continuous and deliberate alignment of these software versions, as seemingly minor version mismatches can lead to substantial and difficult-to-diagnose performance regressions. Consequently, a rigorously version-controlled software environment is not merely a best practice but a primary performance determinant.

3.1.2. Leveraging DeepGEMM Just-in-Time (JIT) Compilation

SGLang employs DeepGEMM Just-in-Time (JIT) compilation by default to enhance throughput, a feature that proved crucial for performance. JIT compilation allows the framework to generate and compile highly specialized matrix multiplication kernels at runtime, tailoring them precisely to the specific model architecture, data types, and hardware (e.g., H100 Tensor Cores) being used. This specialization is key to unlocking the maximum computational performance of the underlying hardware.

This process, however, introduces a notable operational trade-off. The initial execution of any SGLang script with a new configuration incurs a significant "first-run tax"—a compilation phase lasting **10 to 20 minutes**—while these custom kernels are generated. For the purposes of this investigation, this on-the-fly compilation was accepted to simplify the experimental workflow and enhance reproducibility. However, this approach highlights a divergence between research and production deployment strategies. A 10-20 minute startup delay is untenable for production systems that rely on auto-scaling or rapid recovery. Since the compilation is specific to the launch arguments (e.g., model and tensor parallelism degree), this cost would be incurred for each new configuration.

The optimal production strategy involves shifting this compilation step from runtime to build time. The JIT compilation should be executed as part of the container image build process, effectively "baking" the pre-compiled kernels into an immutable deployment artifact. This amortizes the compilation cost and ensures that inference servers have fast, predictable startup times. The observation that direct pre-compilation yields no performance difference compared to the on-the-fly method confirms that the choice is purely one of operational strategy, reinforcing the case for pre-baking kernels in production environments.

3.2. Architectural Analysis: Single-Node Supremacy and the Communication Bottleneck

Following foundational tuning, the investigation pivoted to architectural decisions, comparing deployment environments and node scaling strategies. This analysis yielded the most critical and counter-intuitive finding of the study: for this workload, scaling *up* within a single, powerful node is vastly superior to scaling *out* across multiple nodes.

3.2.1. Initial Benchmarks: The Performance Case for Containerization

An early comparison evaluated the impact of the launch environment by benchmarking SGLang running directly in a Python virtual environment (Venv) against a deployment within a Singularity container. The results showed a distinct advantage for containerization. The Venv-based launch achieved a total throughput of **10,053.70 tok/s**, while the Singularity container reached **11,253.39 tok/s**.

This performance delta of approximately 12%, achieved without any changes to the application code, is substantial. The improvement is likely attributable to the container's ability to create a more optimized and consistent execution environment. A container can bundle a specific set of system libraries and provide more direct, low-overhead access to host hardware resources, such as the InfiniBand fabric and GPU drivers. This minimizes abstraction layers and system call overhead that can be present in a more generalized virtual environment. This finding firmly establishes containerization not merely as a tool for dependency management and portability, but as a mandatory foundational layer for achieving reproducible, high-performance inference with SGLang.

3.2.2. A Comparative Study: Single-Node vs. Two-Node Performance

The core architectural experiment involved a direct comparison between a two-node deployment (utilizing 16 H100 GPUs with a tensor parallelism degree of TP=16) and a single-node deployment (8 H100 GPUs, TP=8). The results were definitive and unexpected: the single-node configuration was significantly faster, achieving **12,520.75 tok/s** compared to the two-node setup's **11,381.11 tok/s**.

The primary factor behind this performance inversion is inter-node communication overhead. Within a single node, GPUs communicate over the extremely high-bandwidth, low-latency NVLink interconnect. When tensor parallelism is stretched across two nodes, communication must traverse the inter-node InfiniBand network. While InfiniBand is a high-performance network, its latency and bandwidth are orders of magnitude inferior to NVLink. For tightly coupled parallelism schemes like tensor parallelism, which require frequent all-reduce and collective communication operations, this network-induced latency becomes the dominant performance bottleneck, completely negating the computational benefit of the additional eight GPUs.

This result defines a critical architectural principle for serving monolithic transformer models. It demonstrates that for a model of DeepSeek R1's scale, an 8xH100 GPU node represents a "sweet spot" of vertical scaling. The model fits within the node's aggregate memory, and the internal NVLink fabric is sufficient to handle the intense communication demands. Attempting to scale beyond this boundary by stretching tensor parallelism across a slower network introduces a bottleneck that cripples performance. This pivotal insight forced a strategic pivot for the entire optimization effort, shifting focus away from scaling out and toward maximizing the computational efficiency of a single node.

3.2.3. Profiling Challenges and Utilization Analysis

The investigation encountered significant practical challenges when attempting to perform deep profiling on the full-scale workload. Using standard tools like NVIDIA Nsight to profile a benchmark with 2000 prompts resulted in unmanageably large trace files, on the order of 8 GB per GPU. This suggests that the tooling ecosystem for high-performance computing is struggling to keep pace with the sheer scale and complexity of modern LLM inference workloads.

As a result, the team resorted to live monitoring of high-level system metrics using tools like htop and nvtop. While less granular, this approach provided clear indicators of system efficiency. A comparison of resource utilization before and after tuning of the **memory-fraction-static** parameter showed a marked improvement in saturation. Post-tuning, GPU utilization rose from an average of ~95% to a consistent **99%**, with some CPU processes reaching 100% utilization, due to the larger KV-cache allocation. Concurrently, GPU memory consumption increased. This uniform increase in resource utilization is a strong positive indicator, demonstrating that the optimizations were successful in reducing idle time and keeping the expensive computational hardware more consistently engaged in useful work.

3.3. Maximizing Computational Throughput: The Primacy of Batch Size

With the architectural focus narrowed to a single, containerized node, the next major lever for performance was identified as the workload itself—specifically, the batch size. The analysis confirmed that maximizing the number of requests processed in parallel is fundamental to achieving high throughput in LLM serving.

3.3.1. The Relationship Between Prompt Volume, KV Cache Saturation, and GPU Parallelism

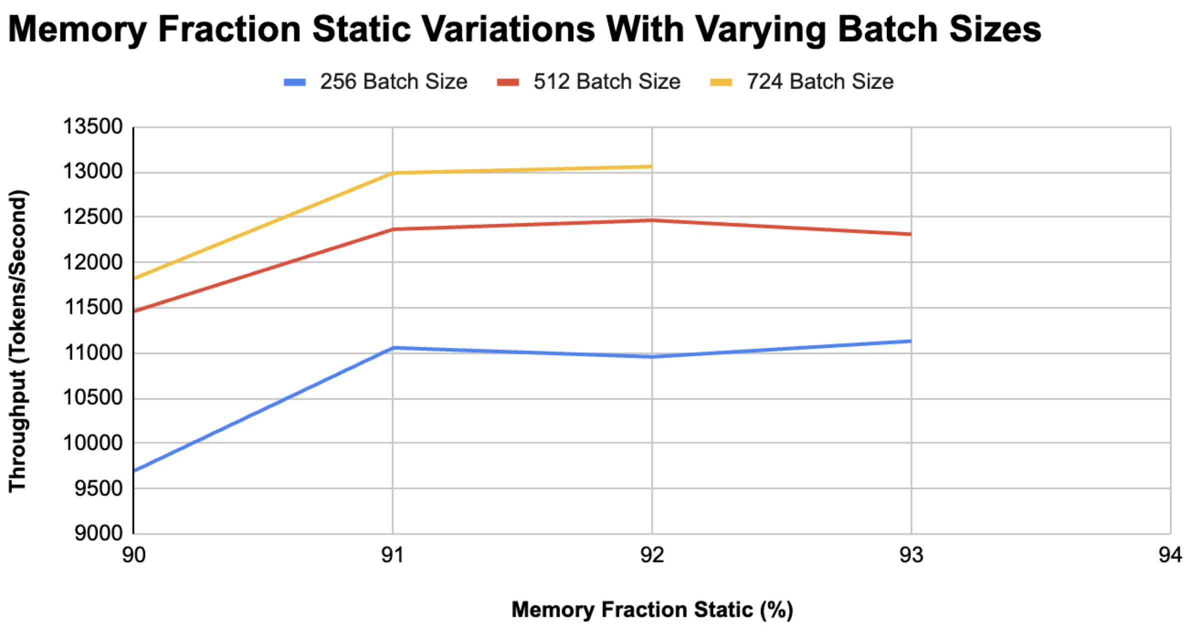
The experiments unequivocally demonstrated that "batch size was king" for throughput. The performance of a GPU is not fixed; it is highly dependent on the arithmetic intensity of the workload. Small batches often lead to a memory-bound state, where the powerful compute units of the GPU are left idle waiting for data. Large batches, conversely, increase the ratio of calculations to memory transfers, shifting the bottleneck toward the GPU's computational capacity—the desired state for maximum efficiency.

This principle was starkly illustrated in an experiment where the number of prompts was halved from 2000 to 1000. This change caused the throughput to plummet from a peak of approximately **17,000 tok/s** to around **10,000 tok/s**, a disproportionate drop of over 40%.This non-linear degradation highlights the importance of batching to amortize kernel launch overheads and, critically, to ensure the

KV Cache remains saturated. A saturated KV cache allows for the maximum reuse of previously computed attention keys and values during the token generation phase, which is fundamental to efficient transformer inference. This finding implies that in any real-world serving system, the design of the request scheduler and its ability to dynamically form large batches from incoming traffic is a primary determinant of overall system throughput.

3.3.2. Empirical Results of Batch Size Scaling and Memory Constraints

The relationship between batch size, memory, and throughput is a delicate balancing act. This was visualized in a chart plotting throughput against the memory_fraction_static parameter, which controls the portion of GPU memory reserved for the KV cache and thus dictates the maximum effective batch size.



The chart shows a clear trend: as the memory fraction increases, and the max cuda batch size parameter is increased, throughput steadily rises. However, this trend continues only up to a critical threshold, beyond which the system abruptly fails with a CUDA Out of Memory (OOM) error.

This behavior encapsulates the core optimization loop for LLM serving: one must push the batch size and KV cache allocation to the absolute limit of the hardware's memory capacity to extract maximum performance. However, this also reveals the fragility of the optimal operating point. Peak performance exists in a very narrow window just before the OOM cliff. In a production environment with variable request lengths and unpredictable traffic, a static configuration tuned to this peak is brittle. A single unexpectedly long request could push the system over the memory limit, causing a catastrophic failure. This highlights the necessity for sophisticated, dynamic memory management and admission control mechanisms in a production-grade LLM server. The system must not only strive to maximize batch size but also incorporate safety mechanisms to gracefully handle memory pressure and prevent OOM errors.

3.4. Advanced Intra-Node Optimization: A Multi-Axis Parallelism Strategy

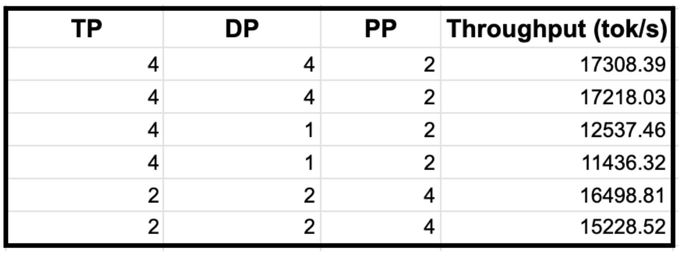
The final and most impactful phase of the optimization process involved devising a sophisticated hybrid parallelism strategy to efficiently partition the model and workload across the eight GPUs within the single high-performance node.

3.4.1. Principles of Tensor, Data, and Pipeline Parallelism in SGLang

The investigation considered the three canonical model parallelism techniques: Tensor Parallelism (TP), which splits individual model layers across multiple devices; Pipeline Parallelism (PP), which assigns entire layers or blocks of layers to different devices, forming a computational pipeline; and Data Parallelism (DP), which replicates the entire model across devices to process different data samples concurrently. The challenge was to find the optimal combination of these strategies for the 8-GPU single-node architecture.

3.4.2. Experimental Analysis of Parallelism Configurations

A series of experiments were conducted to evaluate different hybrid parallelism configurations. The results, presented in a table on page 14 of the source material, identified a clear and dominant winner. A configuration of TP=4, DP=4, PP=2 consistently delivered the highest performance, achieving throughputs of **17,308.39 tok/s** and **17,218.03 tok/s** in two separate runs. This was a significant improvement over other configurations, such as TP=4, DP=1, PP=2 (approx. 12,537 tok/s) and TP=2, DP=2, PP=4 (approx. 16,498 tok/s).



TP	DP	PP	Throughput (tok/s)
4	4	2	17308.39
4	4	2	17218.03
4	1	2	12537.46
4	1	2	11436.32
2	2	4	16498.81
2	2	4	15228.52

The superiority of the TP=4, DP=4, PP=2 configuration is not accidental; it arises from a deep synergy between the parallelism layout and specific optimizations within the SGLang framework. The analysis reveals a multi-layered strategy:

1. **Pipeline Parallelism (PP=2):** At the highest level, the model is split into two stages, each running on a 4-GPU "island." This reduces the frequency of communication between the two islands to only the transfer of activations between pipeline stages. It also enhances KV-Cache locality, as each 4-GPU group manages a distinct portion of the model's layers.
2. **Tensor Parallelism (TP=4):** Within each 4-GPU island, the model's layers are split using 4-way tensor parallelism. This requires intense, all-reduce collective communication, which is perfectly suited to the ultra-high-bandwidth NVLink fabric connecting the GPUs within that group.
3. **Data Parallelism (DP=4)**: The key to this configuration is the interaction between data and tensor parallelism. The condition that TP==DP is a specific requirement to enable a custom SGLang feature called dp-attention. This feature reportedly "speeds up the attention heads" and also enables SGLang's "shared experts (EP) optimisations". This suggests dp-attention is

likely a fused algorithm that optimizes communication by performing parts of the attention computation on local data shards before the final cross-device reduction.

This outcome demonstrates that modern LLM optimization has moved beyond applying generic parallelism rules. Achieving state-of-the-art performance requires a deep understanding of the serving framework's specific architectural features and custom kernels, and designing a hybrid parallelism strategy that explicitly exploits them.

3.5. An Investigation into Inter-Node Network Tuning with NCCL

Although the primary optimization effort ultimately focused on a single node, the initial investigation into multi-node performance included an exhaustive attempt to tune the inter-node communication network. This work, while not yielding a direct performance gain, served as a crucial piece of negative evidence that validated the architectural pivot.

3.5.1. Overview of the Network Topology

The hardware environment consisted of standard high-performance computing nodes. Internally, GPUs within a node are interconnected via **NVLink**. For communication between nodes, the system relies on a high-speed **InfiniBand** fabric, with some channels for the RoCE v2 protocol. The NVIDIA Collective Communications Library (NCCL) is the software layer responsible for orchestrating communication over this topology.

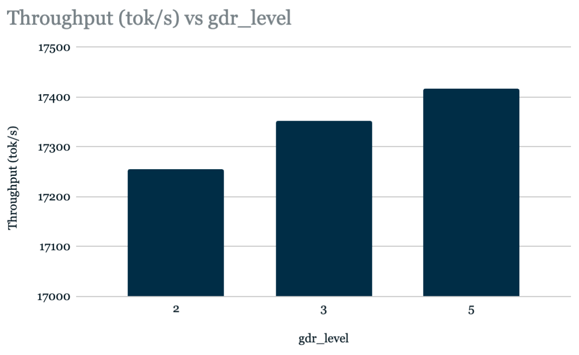
3.5.2. Analysis of NCCL Parameter Experimentation and the Efficacy of Defaults

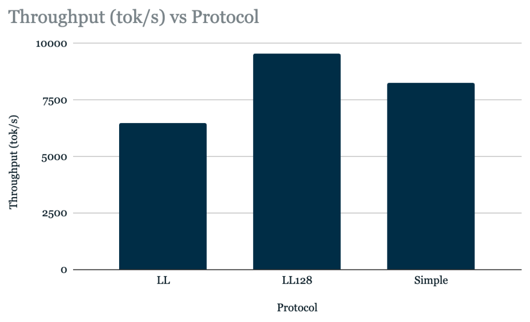
An extensive series of experiments was conducted to manually tune dozens of NCCL environmental variables, including parameters like Algorithm, Protocol, nthreads, and min_nchannel. The goal was to determine if the poor multi-node performance was due to a sub-optimal NCCL configuration. The results were comprehensive and conclusive: none of the manually tuned configurations came close to matching the performance of the default settings. The manually configured experiments yielded throughputs ranging from a low of ~2,000 tok/s to a high of ~9,500 tok/s. In contrast, the default, untuned NCCL configuration had already achieved **10,053.7 tok/s** in the baseline test.

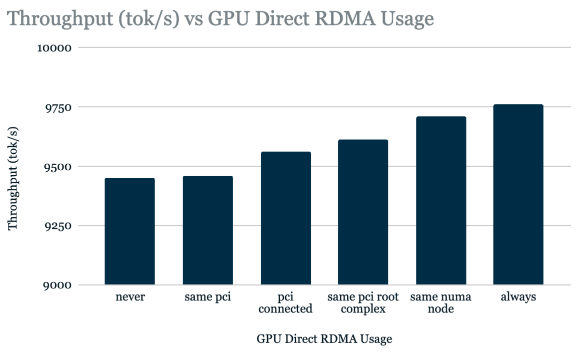
This is a powerful "null result." The inability of extensive manual tuning to outperform the default configuration strongly suggests that modern versions of NCCL incorporate sophisticated auto-tuning capabilities. The library can likely probe the network topology and dynamically adjust its algorithms "on the fly" more effectively than any static configuration.

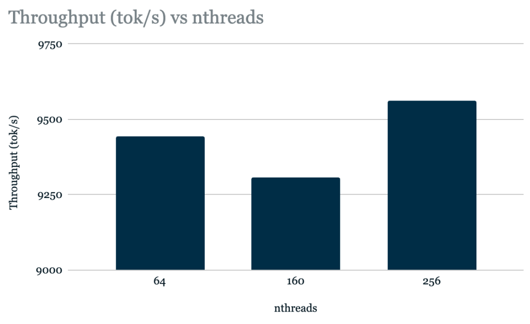
Most importantly, this exhaustive investigation allowed the team to confidently rule out NCCL misconfiguration as the cause of the multi-node bottleneck. It provided definitive proof that the performance limitation was not in the *implementation* of the communication but in the *fundamental, irreducible latency* of the inter-node network itself. This validation was essential, as it provided the rigorous justification needed to abandon the multi-node tuning path and refocus all efforts on the more promising single-node optimization strategy.

Here’s a couple of charts showing some of the parameters we tuned:









Each parameter tuned had a reason for us to investigate that specific parameter and tune it, but in the interest of brevity, I'll skip over them. The summary of these NCCL running results was that we decided to stick with the tuned gdr_level parameters, setting it to 5, and leave all of the others in their default state, as the stock NCCL tuning is already great.

6. Synthesis of Results: A 75% Improvement in Throughput

The cumulative impact of the multi-stage optimization process resulted in a dramatic improvement in offline throughput. By systematically addressing the software stack, deployment architecture, and parallelism strategy, performance was increased by 75% over the initial baseline.

3.6.1. From Baseline to Optimized: A Step-by-Step Performance Walkthrough

The optimization journey can be traced through several key performance milestones:

- **Initial Baseline:** The starting point was a standard SGLang launch in a Python Venv on a single node, which established a baseline throughput of **9,890.30 tok/s**.
- **Containerization & Basic TP:** The first major improvement came from moving to a Singularity container and applying a simple tensor parallelism strategy (TP=8) across the single node. This configuration raised performance to **12,414.55 tok/s**.
- **Hybrid Parallelism**: The final and most significant gain was achieved by implementing the advanced hybrid parallelism strategy. The optimal configuration of TP=4, DP=4, PP=2, designed to leverage SGLang's dp-attention feature, pushed the final throughput to an average of 17,255.40 tok/s.

3.6.2. Final Benchmark Results: Quantifying the Gains

The final optimized configuration represents a **+40% throughput increase** over the already-tuned single-node TP=8 baseline and a **+75% throughput improvement** over the initial provided baseline This progression clearly illustrates that the most substantial performance gains were realized not from adding more hardware—which was shown to be counter-productive—but from intelligent software configuration and a nuanced parallelism design that was deeply integrated with the serving framework's capabilities.

The following table synthesizes the performance data from across the investigation, providing a clear narrative of the incremental and cumulative gains achieved at each major stage.

Configuration	**Key Optimizations**	**Total Throughput (tok/s)**	**Improvement over Baselined**
Initial Baseline	Python Venv, TP=16	9,890.30	0%
Two-Node Baseline	Singularity, TP=16	11,381.11	+15.1%
Single-Node Baseline	Singularity, TP=8	12,414.55	+25.5%
Final Optimized Two Node	Singularity, TP=4, DP=4,PP=2, dp-attention, NCCL Tuning	**17,033.94**	**+72.2%**
Final Optimized Two Node	Singularity, TP=4, DP=4,PP=2, dp-attention, NCCL Tuning	**17,417.40**	**+76.3%**

3.7. From Theory to Practice: A Proposed Architecture for Real-World Serving

The findings from this offline benchmark analysis lead to a clear and pragmatic architectural recommendation for deploying SGLang in a live, online serving environment. The key is to leverage the highly optimized single-node configuration as a scalable building block.

3.7.1. Limitations of Offline Benchmarking for Data Parallelism

A crucial caveat noted during the investigation is that "true DP across two nodes in SGLang is not available in the form of an offline benchmark". The framework's offline benchmarking tool is designed for tightly coupled model execution, not for simulating a scenario where multiple independent model replicas serve a shared request queue. An exploratory attempt at an online serving setup using this model yielded a throughput of ~10,000 tok/s, which was not competitive with the optimized single-node performance. This highlights a distinction between framework-level data parallelism and infrastructure-level replication.

3.7.2. The Router-Worker Model for Scalable, High-Performance Deployment

The proposed architecture for a real-world deployment is a classic router-worker (or load balancer-replica) model. This design consists of a central **Router** that receives all incoming inference requests and distributes them to a fleet of independent **Worker** nodes. Each worker node would be a self-contained, powerful instance running the fully optimized single-node SGLang configuration identified in this study (i.e., TP=4, DP=4, PP=2).

This architecture is the logical conclusion of all the preceding findings. Having established that tightly coupling nodes for a single inference task is inefficient due to communication overhead, the best way to scale beyond a single node is to avoid coupling them altogether. Instead, parallelism is achieved at the request level. This approach effectively treats the entire 17.2k tok/s single-node setup as a single, high-throughput "inference unit." Scaling the total capacity of the system is as simple as adding more of these identical worker units behind the load balancer. This model is highly scalable, resilient (the failure of one worker does not impact the others), and operationally simple, transforming a complex distributed computing problem into a well-understood load-balancing problem. It applies the principle of Data Parallelism at the infrastructure level, which is the most effective way to scale once the performance of a single replica has been maximized.

3.8. Conclusion and Acknowledged Avenues for Further Investigation

3.8.1. Summary of Key Optimization Levers for SGLang

This comprehensive investigation into the performance of the SGLang framework has yielded several key principles for maximizing throughput:

1. **Environment is Key:** System performance is highly sensitive to the software stack. Maintaining alignment between the SGLang version, CUDA toolkit, and drivers is critical. Containerization provides a necessary layer of stability and performance enhancement.
2. **Vertical Scaling First:** For tightly coupled models like transformers, performance is dictated by the highest-bandwidth communication fabric available. One must first exhaust the computational and memory capacity of a single, NVLink-connected node before attempting to scale out to a slower inter-node network.
3. **Batch Size is Paramount:** System throughput is fundamentally driven by the ability to form large request batches. This maximizes computational intensity and ensures the KV Cache remains saturated, which is essential for efficient generation.
4. **Hybrid Parallelism is Optimal**: The highest performance is achieved not by maximizing a single type of parallelism, but by creating a balanced, hybrid strategy (TP, DP, PP) that is specifically designed to exploit the custom features and optimizations of the serving framework, such as SGLang's dp-attention.

5. **Scale Out Loosely:** To scale beyond a single node, a replicated worker model with a load balancer is the most pragmatic and performant architecture. This sidesteps the communication bottlenecks inherent in tightly coupled multi-node parallelism.

3.8.2. Other Areas Explored

We explored several other areas that are not detailed in this report. In each case, we found no substantial gain to be had through further investigation, and the results were not noteworthy enough for inclusion.

- Torch Compile (torch.compile): no significant performance improvement.
- Expert Parallelism: Adjustments were tested but did not produce a substantial gain.
- Nsight Profiling (Small Workloads): This analysis did not provide any noteworthy insights beyond what was already gathered from larger tests.