

Scalability and Performance Analysis

Orian Louant

Scalability Analysis

Scalability is the ability of a computer system or application to handle increasing workloads as its size or resources grow. The term scaling is commonly used to describe how effectively hardware or software can deliver greater computational power when additional resources, such as processors are added

The speedup S and parallel efficiency E of an application can be defined as

$$S = \frac{t(1)}{t(N)} \qquad E = \frac{t(1)/N}{t(N)}$$

where $t(1)$ is the time to run the application using one processor, and $t(N)$ is the computational time running the same application with N processors

Make sure your application is compiled with optimizations enabled and that the compiler targets the instruction set supported by the NIC5 CPUs (AVX2):

```
gcc/mpicc -march=native -O2 -ftree-vectorize -ffast-math -fopenmp -o fdtd fdtd.c -lm
```

-march=native

Tells the compiler to auto-detect the CPU and generate code optimized for its specific architecture

-O2

Enables moderate/advanced compiler optimizations for speed without aggressive trade-offs

-ftree-vectorize

Enables automatic vectorization of loops using SIMD instructions

-ffast-math

Allows faster but less strictly IEEE-compliant floating-point optimizations

Strong scaling measures how the time to solution decreases when increasing the number of processing elements for a fixed total problem size

- Shows how efficiently an application can use more cores/nodes without increasing workload
- Ideal strong scaling: doubling compute resources halves runtime

In an OpenMP application, strong scaling is evaluated by increasing the number of threads while keeping the total problem size fixed. In an MPI application, strong scaling is measured by increasing the number of MPI ranks under the same fixed workload.

Weak scaling measures how the time to solution changes when the problem size increases proportionally to the number of processing elements, so that the work per processing element remains constant

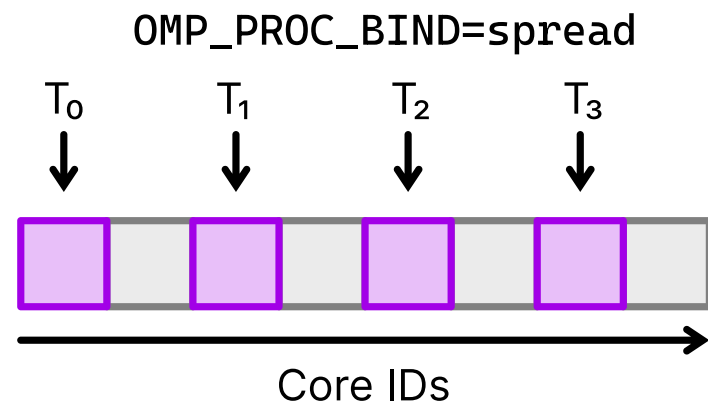
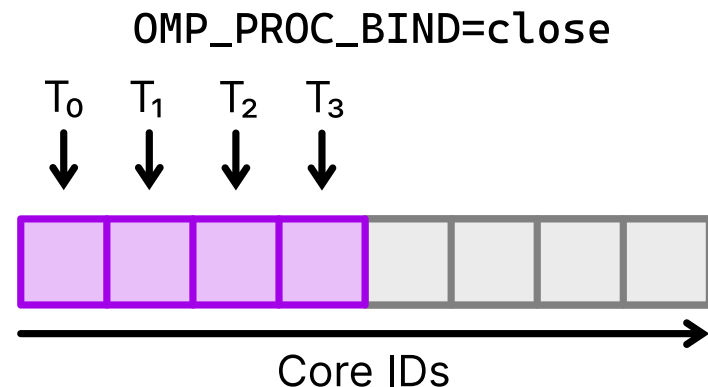
- Shows how efficiently an application can use more cores/nodes when the workload scales with available resources
- Ideal weak scaling: runtime remains constant as compute resources and problem size grow proportionally

In an OpenMP application, weak scaling is evaluated by increasing the number of threads while increasing the problem size so that each thread performs the same amount of work. In an MPI application, weak scaling is measured by increasing the number of MPI ranks while scaling the total problem size so each rank handles an equal-sized portion.

When running your scaling experiment, you have several options for distributing your processing elements, and this placement strategy can significantly influence the outcome of the experiment

- **Close:** place the processing elements to subsequent cores
- **Spread:** distribute threads evenly across the available cores

The environment variable `OMP_PROC_BIND` can be used to control threads placement for OpenMP. The `--cpu-bind="map_cpu:<cpu-list>"` option of `srun` can be used to control placement of MPI ranks



In order to avoid interference from other jobs running on the same node, it's recommended to run the scaling experiment in "exclusive" mode, i.e., on a full compute node. Typical job parameters would be:

```
#SBATCH --job-name="OpenMP strong scaling"
#SBATCH --exclusive
#SBATCH --mem=0
#SBATCH --time=02:00:00
#SBATCH --partition=hmem
```

The scaling experiment can then be run in a single batch job by increasing the number of threads via a bash loop ([download example](#))

```
module load Info0939Tools

numCpuCores=$SLURM_CPUS_ON_NODE
startDir=$(pwd)
executable="/path/to/your/application <args>"

for binding in close spread; do
    export OMP_PROC_BIND=${binding}

    numThreads=1
    while [[ $numThreads -le $numCpuCores ]]; do
        export OMP_NUM_THREADS=${numThreads}

        workDir="openmp_${binding}_${numThreads}"
        output="${numThreads}threads.out"

        mkdir -p ${workDir} && cd ${workDir}
        ${executable} > ${output}
        cd ${startDir}

        (( numThreads *= 2 ))
    done
done
```

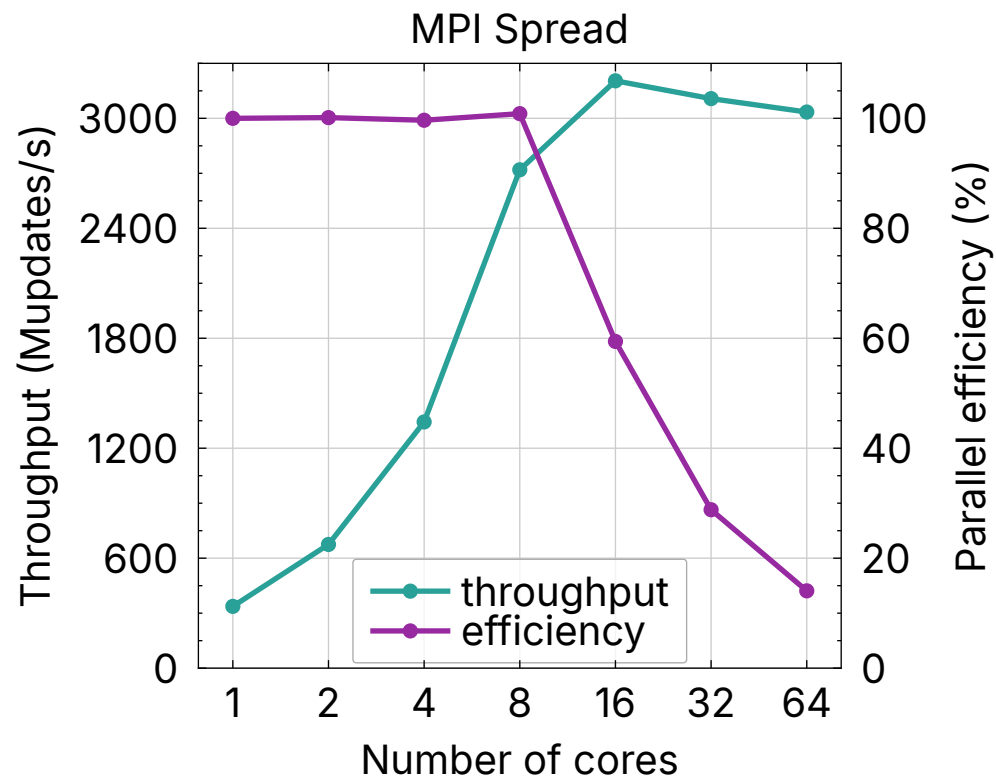
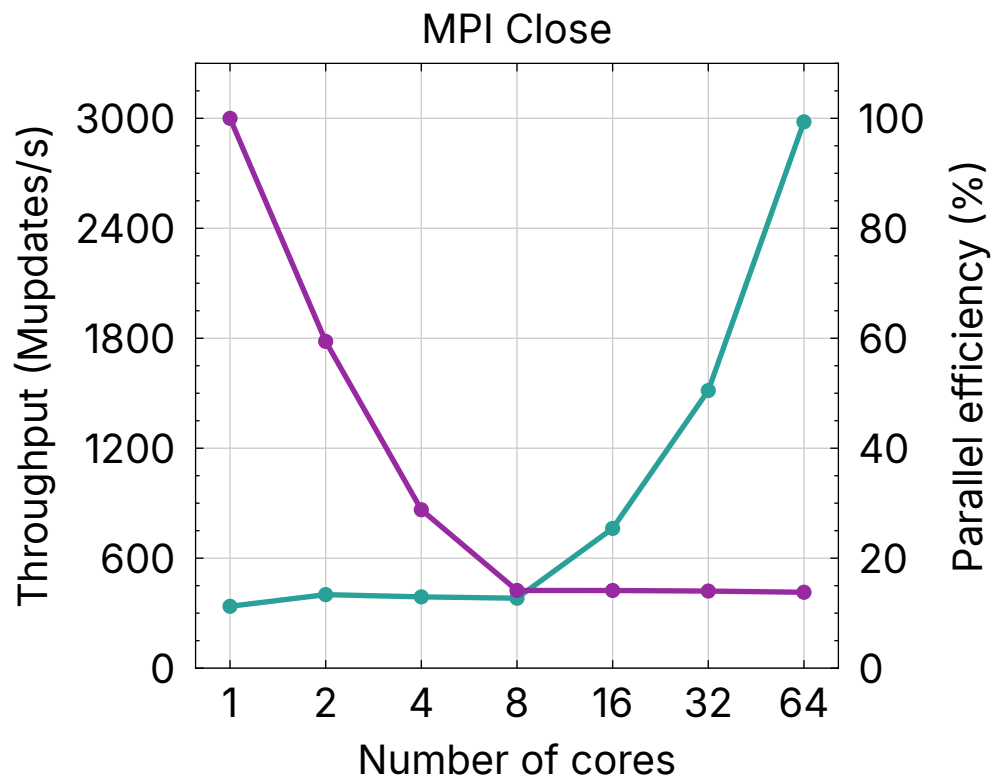

The script presented in the previous slide can be modified for MPI the following way ([download example](#)):

```
for binding in close spread; do
  numRanks=$SLURM_JOB_NUM_NODES
  while [[ $numRanks -le $((numCpuCores * SLURM_JOB_NUM_NODES)) ]]; do
    if [[ ${binding} == "spread" ]]; then
      step=$((numCpuCores / (numRanks / SLURM_JOB_NUM_NODES)))
      srunBind=$(seq -s ',' 0 ${step} $((numCpuCores - 1)))
    else
      srunBind=$(seq -s ',' 0 $((numRanks / SLURM_JOB_NUM_NODES - 1)))
    fi

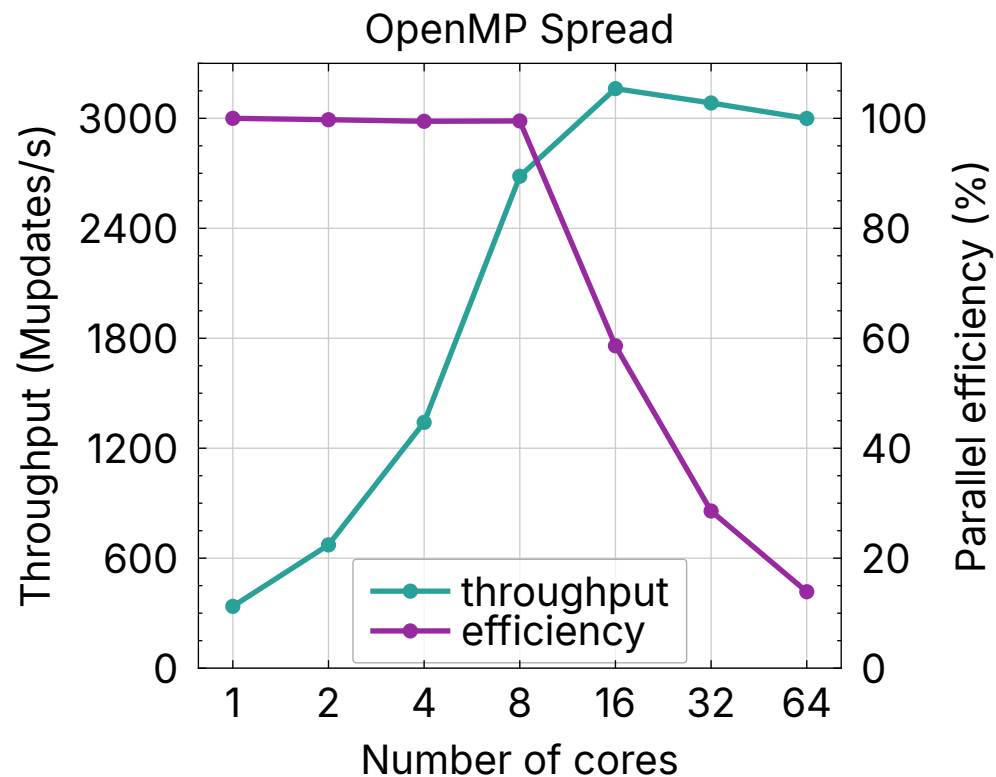
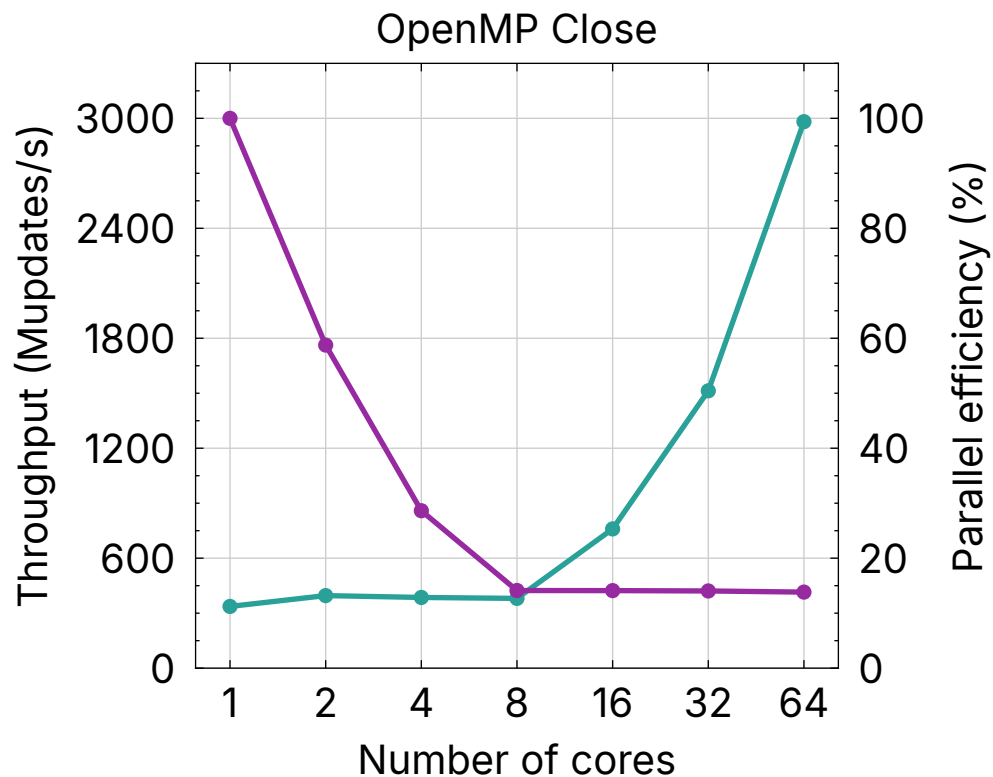
    workDir="mpi_${binding}_${SLURM_JOB_NUM_NODES}nodes_${numRanks}"
    output="${numRanks}ranks.out"
    mkdir -p ${workDir} && cd ${workDir}

    srun -N $SLURM_JOB_NUM_NODES -n ${numRanks} --cpu-bind="map_cpu:${srunBind}" ${executable}
    cd ${startDir}

    (( numRanks *= 2 ))
  done
done
```



The example result are for a single compute node runs. You are free to experiment with multiple nodes to see how the results are affected



If your OpenMP implementation is well written, the results should be similar to the results of the MPI implementation

For the exam, we expect from you that you explain your results, not just run the scaling experiments

- In your analysis, consider the limitations of the system and CPUs on which your experiments were executed
- Do not forget to account for the hardware topology: the organization of the cores, the cache hierarchy, and the structure of the memory system

For the strong scaling, choose a problem size big enough for the maximum level of parallelism of your experiment!

Performance evaluation using hardware performance counters

LIKWID is a command-line performance tool suite for Linux, used by performance-oriented programmers to measure **hardware performance counters**

- CPU performance counters are specialized hardware registers built into modern processors that track various low-level events and activities happening during program execution
- Measured activities include but are not restricted to instructions executed, number of clock cycles, cache hits and misses and memory operations
- CPU performance counters are vendor/CPU model dependent and LIKWID helps the analysis via performance group that hides the low-level details

Throughout our introduction to LIKWID, we will use a matrix-matrix multiplication as a basis for our investigation. For square matrices A and B of size $N \times N$, each element of the result matrix C , also of size $N \times N$, are computed as

$$C_{ij} = \sum_k^N A_{ik} \cdot B_{kj}$$

and the naive C implementation is as follows:

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        for (int k = 0; k < n; k++) {  
            C[i * n + j] += A[i * n + k] * B[k * n + j];  
        }  
    }  
}
```

The first step is to compile and run our code in order to have a rough idea of its performance:

```
$ gcc -march=native -O2 -ftree-vectorize -ffast-math -o dgemm dgemm.c  
$ ./dgemm 2048  
Multiplication of matrices of size 2048 done is 225.457 secs
```

From this result, we can evaluate the performance of our code in terms of FLOPS/secs:

$$\frac{2 \text{ FLOPS} \times 2048 \times 2048 \times 2048}{225.457 \text{ secs}} = 76.2 \text{ MFLOPS/secs}$$

This value is significantly lower than the single-core peak performance of the NIC5 CPUs (46.4 GFLOPS/secs) and therefore warrants further investigation

The `likwid-perfctr` is the LIKWID tool used for measuring hardware performance metrics

```
module load Info0939Tools
```

```
likwid-perfctr -C <core_list> -g <perf_group> <application> <args>
```

`-C <core_list>` specifies the CPU cores to pin the application to (e.g., `-C 0-3` binds to cores 0-3)

`-g <perf_group>` selects a predefined performance group

Pinning ensures that the code always runs on the same core(s), making performance measurements consistent and repeatable. Without pinning, the operating system scheduler may migrate the process between cores, causing cache flushes, loss of branch predictor state, and other side effects that can distort performance results

Available performance groups can be listed using the `-a` option. This command displays all predefined performance groups along with their names (used as arguments for the `-g` option) and a brief description of each

More detailed information about a specific performance group can be obtained using the `-H` option

```
likwid-perfctr -H -g <perf_group>
```

```
$ likwid-perfctr -a
```

Group name	Description
MEM	Main memory bandwidth in MBytes/s
MEM_DP	Main memory bandwidth in MBytes/s
MEM_NUMA_0	Main memory bandwidth for NUMA node 0
MEM_NUMA_1	Main memory bandwidth for NUMA node 1
MEM_NUMA_2	Main memory bandwidth for NUMA node 2
MEM_NUMA_3	Main memory bandwidth for NUMA node 3
MEM_SP	Main memory bandwidth in MBytes/s
BRANCH	Branch prediction miss rate/ratio
CACHE	Data cache miss rate/ratio
CLOCK	Cycles per instruction
CPI	Cycles per instruction
DATA	Load to store ratio
DIVIDE	Divide unit information
ENERGY	Power and Energy consumption
FLOPS_DP	Double Precision MFLOP/s
FLOPS_SP	Single Precision MFLOP/s
ICACHE	Instruction cache miss rate/ratio
L2	L2 cache bandwidth in MBytes/s
L3	L3 cache bandwidth in MBytes/s
NUMA	Local and remote memory accesses
TLB	TLB miss rate/ratio

In most cases, we don't want to measure the performance of the entire application, but rather focus on specific sections of code. The LIKWID Marker API allows you to instrument your code to measure individual code regions instead of full program runs

To use the Marker API:

- Initialize it with `LIKWID_MARKER_INIT`
- Register the code region(s) you want to measure using `LIKWID_MARKER_REGISTER(tag)`
- At the end of the application finalize the measurements with `LIKWID_MARKER_CLOSE`

```
#ifdef LIKWID_PERFMON
#include <likwid-marker.h>
#else
#define LIKWID_MARKER_INIT
#define LIKWID_MARKER_REGISTER(region_tag)
#define LIKWID_MARKER_START(region_tag)
#define LIKWID_MARKER_STOP(region_tag)
#define LIKWID_MARKER_CLOSE
#endif

int main(int argc, char* argv[]) {
    LIKWID_MARKER_INIT;
    LIKWID_MARKER_REGISTER("dgemm");

    // A, B, C allocation and initialization

    dgemm(n, A, B, C);

    LIKWID_MARKER_CLOSE;

    return 0;
}
```

The next step is to use `LIKWID_MARKER_START(tag)` and `LIKWID_MARKER_STOP(tag)` to mark the beginning and end of the specific code section you want to measure

The approach presented here activates the Marker API only when the code is compiled with the `-DLIKWID_PERFMON` flag, allowing the program to be built both with and without LIKWID instrumentation

To simplify compilation, a compiler wrapper, `likwid-gcc` is provided. This wrapper automatically enables the Marker API and links your application against the LIKWID library

```
void dgemm(int n, const double* A,
           const double* B, double* C) {
    LIKWID_MARKER_START("dgemm");

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                C[i * n + j] += A[i * n + k] * B[k * n + j];
            }
        }
    }

    LIKWID_MARKER_STOP("dgemm");
}
```

Now that we have instrumented our code, we can perform our first measurement using the CACHE performance group. The results indicate the following:

- The instruction throughput is very poor, with a clocks per instruction (CPI) value of 11.5, meaning that, on average, each instruction requires 11.5 CPU cycles to execute
- The cache miss ratio is also high (0.23), implying that approximately one in every four data accesses results in a cache miss

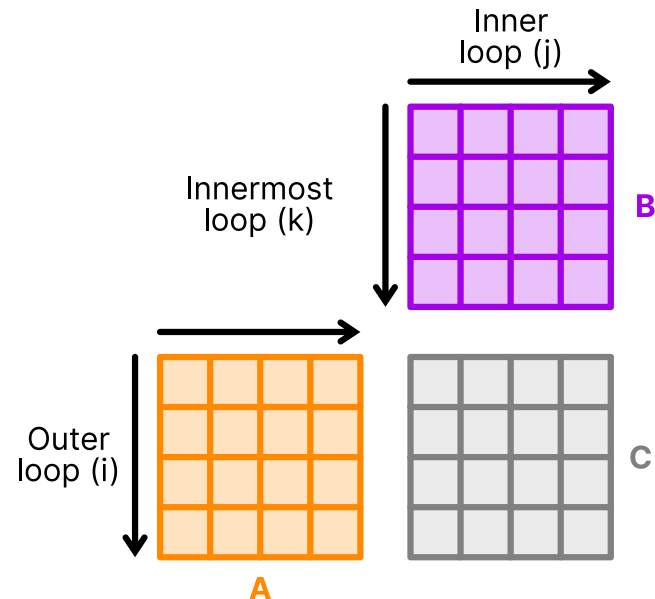
```
$ likwid-gcc <other_flags> -o dgemm dgemm
$ likwid-perfctr -m -C 0 -g CACHE ./dgemm 2048
```

...

Metric	HWThread 0
Runtime (RDTSC) [s]	237.8803
Runtime unhalted [s]	273.6172
Clock [MHz]	3335.6490
CPI	11.4704
data cache requests	199907600000
data cache request rate	2.9074
data cache misses	47846170000
data cache miss rate	0.6959
data cache miss ratio	0.2393

Reasons for the high cache miss ratio:

- In the innermost loop (k), we access columns of the B matrix. Since the matrices are stored in row-major order, these non-contiguous accesses cause many cache lines to be only partially utilized
- For the C matrix, the same memory location is updated repeatedly across iterations of the k loop. but we might have temporal locality issues



If we invert the order of the k and j loop, we have contiguous accesses for both B and C

```
for (int i = 0; i < n; i++) {  
    for (int k = 0; k < n; k++) {  
        for (int j = 0; j < n; j++) {  
            C[i * n + j] += A[i * n + k] * B[k * n + j];  
        }  
    }  
}
```

Now that we have changed the order of the loops, let's do a new measurement

- The instruction throughput massively improved from 11.5 CPI to 0.9, a 12.7x improvement
- The cache miss ratio did go down from 0.23 to 0.05, a 4.6x improvement

In addition, the runtime also massively improved, from 237.8 secs down to 3.5 secs, a significant improvement of ~68x

```
$ likwid-gcc <other_flags> -o dgemm dgemm
$ likwid-perfctr -m -C 0 -g CACHE ./dgemm 2048
```

...

Metric	HWThread 0
Runtime (RDTSC) [s]	3.5301
Runtime unhaltd [s]	4.0796
Clock [MHz]	3351.2556
CPI	0.9051
data cache requests	10441220000
data cache request rate	0.8028
data cache misses	530963400
data cache miss rate	0.0408
data cache miss ratio	0.0509

To investigate the significant speedup, we use the FLOPS_DP performance group, which measures double-precision FLOPS. The results show that, while the total number of retired FLOPS remains the same, the number of retired instructions decreases after loop reordering. **After reordering, we have more FLOPS than retired instructions!**

Before loop reordering

```
$ likwid-perfctr -m -C 0 -g FLOPS_DP ./dgemm 2048
```

Event	Counter	HWThread 0
RETIRED_INSTRUCTIONS	PMC0	68757520000
RETIRED_SSE_AVX_FLOPS_ALL	PMC2	17179870000

Metric	HWThread 32
Runtime (RDTSC) [s]	233.7374
DP [MFLOP/s]	73.5007

After loop reordering

```
$ likwid-perfctr -m -C 0 -g FLOPS_DP ./dgemm 2048
```

Event	Counter	HWThread 0
RETIRED_INSTRUCTIONS	PMC0	13006610000
RETIRED_SSE_AVX_FLOPS_ALL	PMC2	17179870000

Metric	HWThread 32
Runtime (RDTSC) [s]	3.4977
DP [MFLOP/s]	4911.7466

The reason the number of retired instructions is lower than the number of FLOPS is that loop reordering enables the compiler to exploit vectorization. The NIC5 CPU supports AVX2 (256-bit vectors), which allows operations on four double-precision values simultaneously

```
.K_LOOP:
```

```
vmovsd    (%r8), %xmm2
```

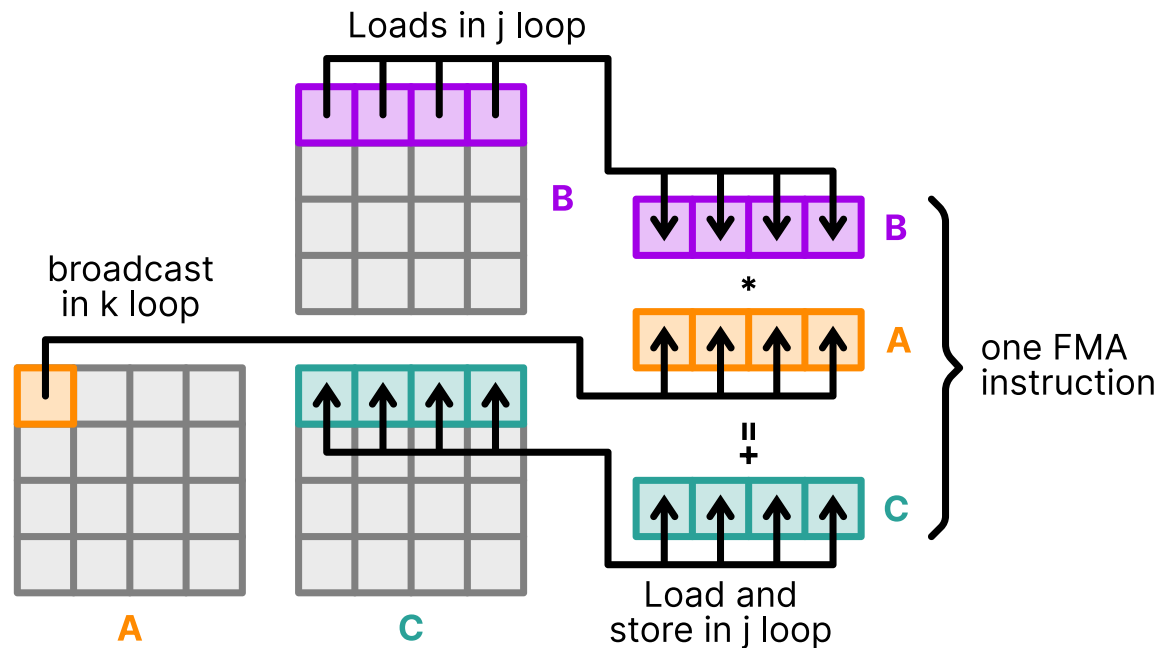
```
vbroadcastsd %xmm2, %ymm1
```

```
.J_LOOP:
```

```
vmovupd    (%rsi,%rcx), %ymm0
```

```
vfmadd213pd (%rax,%rcx), %ymm1, %ymm0
```

```
vmovupd    %ymm0, (%rax,%rcx)
```



Another useful performance group is MEM_DP, which measures memory volume, memory bandwidth, and FLOPS, allowing you to compute the arithmetic intensity of your application

```
$ likwid-perfctr -C 0 -m -g MEM_DP ./dgemm 8192
```

Metric	HWThread 0
Runtime (RDTSC) [s]	235.1049
Runtime unhaltd [s]	230.1263
Clock [MHz]	2838.6303
CPI	0.8026
DP [MFLOP/s]	4676.6869
Memory bandwidth [MBytes/s]	18753.4314
Memory data volume [GBytes]	4409.0236
Operational intensity	0.2494

For this measurement, ensure that the problem size is large enough, since only traffic to main memory is counted. If your data fits in the L3 cache, most requests will be served from the cache rather than main memory, leading to misleading results

LIKWID can be used in a multithreaded environment. In such cases, most events are collected individually for each thread. However, some hardware events, known as uncore events, are not associated with a specific core. These events are only measured on designated cores within the system:

- L3 events are collected only on the first core of each L3 slice (one slice of 16 MB per 4 cores)
- Main memory events are collected only on core 0 of each socket.

However, even though these events are reported only on specific cores, the measured values represent the aggregated activity of all cores within the corresponding uncore domain.

The NUMA performance group can be used to analyze NUMA-related effects. For example, consider running the matrix multiplication example with four cores and four threads, binding each thread to the first core of each NUMA domain on the first socket:

```
$ likwid-perfctr -C 0,8,16,24 -m -g NUMA ./dgemm 8192
```

Metric	Sum	Min	Max	Avg
Local bandwidth [MBytes/s] STAT	57090.7299	12283.9705	19551.0468	14272.6825
Local data volume [GBytes] STAT	6442.5716	1386.2203	2206.2946	1610.6429
Remote bandwidth [MBytes/s] STAT	21115.5064	0.0001	7268.0977	5278.8766
Remote data volume [GBytes] STAT	2382.8419	9.024000e-06	820.1896	595.7105
Total bandwidth [MBytes/s] STAT	78206.2363	19550.6448	19552.4764	19551.5591
Total data volume [GBytes] STAT	8825.4135	2206.2512	2206.4579	2206.3534

We can see that most data cache requests hit the local NUMA domain (6442 GB), but there is still a significant amount of data accessed from a remote domain (2382 GB)

Another way to examine NUMA effects is to use the MEM performance group and inspect the amount of data transferred on each memory channel:

```
$ likwid-perfctr -C 0,8,16,24 -m -g MEM ./dgemm 8192
```

Event	Counter	Sum	Min	Max	Avg
ACTUAL_CPU_CLOCK STAT	FIXC1	1255426400000	300336800000	327145200000	313856600000
MAX_CPU_CLOCK STAT	FIXC2	1258564900000	300722000000	328084800000	314641225000
RETIRED_INSTRUCTIONS STAT	PMC0	2200787800000	550175400000	550258800000	550196950000
CPU_CLOCKS_UNHALTED STAT	PMC1	1250435600000	299099600000	326108000000	312608900000
DRAM_CHANNEL_0 STAT	DFC0	34362430000	0	34362430000	8590607500
DRAM_CHANNEL_2 STAT	DFC1	24516780	0	24516780	6129195
DRAM_CHANNEL_4 STAT	DFC2	1525707	0	1525707	381426.7500
DRAM_CHANNEL_6 STAT	DFC3	2343867	0	2343867	585966.7500

Here, we can see that the memory channel of the first NUMA node (DRAM_CHANNEL_0) is handling the majority of memory requests.

To improve NUMA locality, we leverage the first-touch policy by initializing our matrices in parallel, ensuring that blocks of rows are distributed across the different NUMA nodes:

```
#pragma omp parallel for
for (int rowIdx = 0; rowIdx < n; rowIdx++) {
    for (int colIdx = 0; colIdx < n; colIdx++) {
        A[rowIdx * n + colIdx] = (double)rand();
        B[rowIdx * n + colIdx] = (double)rand();
        C[rowIdx * n + colIdx] = 0.0;
    }
}
```

Note that this approach is not perfect. Remote NUMA accesses will still occur, but it helps balance the memory load across the memory subsystems of each NUMA node

After updating our code, a new measurement using the NUMA performance group shows that the total number of local and remote accesses did not change significantly. However, the accesses are now balanced across threads, with minimum and maximum values much closer. Additionally, the runtime improved from 112 seconds to 64 seconds

```
$ likwid-perfctr -C 0,8,16,24 -m -g NUMA ./dgemm 8192
```

Metric	Sum	Min	Max	Avg
Local bandwidth [MBytes/s] STAT	91650.1722	22891.8403	22957.8496	22912.5431
Local data volume [GBytes] STAT	5905.6551	1475.0797	1479.3331	1476.4138
Remote bandwidth [MBytes/s] STAT	45393.4846	11299.2077	11375.2625	11348.3711
Remote data volume [GBytes] STAT	2925.0164	728.0861	732.9869	731.2541
Total bandwidth [MBytes/s] STAT	137043.6568	34256.2538	34272.6847	34260.9142
Total data volume [GBytes] STAT	8830.6715	2207.3675	2208.4262	2207.6679

The reduction in runtime can be explained by the fact that all available memory channels are now being utilized. Previously, DRAM_CHANNEL_0 handled 34 billion transactions, but these transactions are now distributed across the other memory channels. As a result, the memory bandwidth increased from 38.9 GB/s to 70.7 GB/s

```
$ likwid-perfctr -C 0,8,16,24 -m -g MEM ./dgemm 8192
```

Event	Counter	Sum	Min	Max	Avg
DRAM_CHANNEL_0 STAT	DFC0	8464177000	0	8464177000	2116044250
DRAM_CHANNEL_2 STAT	DFC1	8609297000	0	8609297000	2152324250
DRAM_CHANNEL_4 STAT	DFC2	8596527000	0	8596527000	2149131750
DRAM_CHANNEL_6 STAT	DFC3	8730601000	0	8730601000	2182650250

Performance measurement with Score-P and Scalasca

When profiling an application, you have two main options:

- **Sampling:** Periodically records the program counter (PC) or call stack to provide a statistical approximation of program behavior. This method has low overhead but is less precise, as it may miss short-lived functions or fine-grained events
- **Instrumentation:** Inserts measurement code at function entries/exits or specific code regions, capturing every event: exact call counts, timings, and call paths. This method has higher overhead, especially for short-lived or frequently called functions

Score-P is a performance measurement infrastructure for HPC applications that relies on both automatic and manual instrumentation. It collects detailed metrics such as function runtimes, call counts, and communication patterns

The general recommendations for running a profiling tracing analysis are similar to those for a scaling study. Pay particular attention to the following points:

- The workload must be representative of the real application so that communication patterns, load imbalance, and computational hotspots reflect actual behavior. Too small a problem may skew the analysis
- However, tracing can produce a large amount of data and long runs can generate trace files that are too large to process efficiently. In practice, running up to about 100 time steps is usually enough to capture representative behavior while keeping trace size manageable

Profiling introduces some overhead, but the runtime of your application with profiling enabled should remain close to the original, ideally within about 5%

To instrument your code, you need to compile the code using the Score-P instrumentation command (`scorep`), which is added as a prefix to your compile statement

```
module load Info0939Tools  
scorep gcc <compiler_flags>
```

or, for an MPI application

```
module load Info0939Tools  
scorep mpicc <compiler_flags>
```

In most cases the Score-P instrumentor is able to automatically detect the programming paradigm (MPI, OpenMP) from the set of compile and link options given to the compiler

Once your application is instrumented, you can start a profiling run by setting the following environment variables:

```
export SCOREP_ENABLE_PROFILING=true
export SCOREP_ENABLE_TRACING=false
export SCOREP_EXPERIMENT_DIRECTORY=profiling
```

The variable SCOREP_EXPERIMENT_DIRECTORY specifies the directory where Score-P will store the profiling results. After setting these variables, run your application as usual. For an OpenMP application:

```
export OMP_NUM_THREADS=<numthreads>
<application> <args>
```

For an MPI application

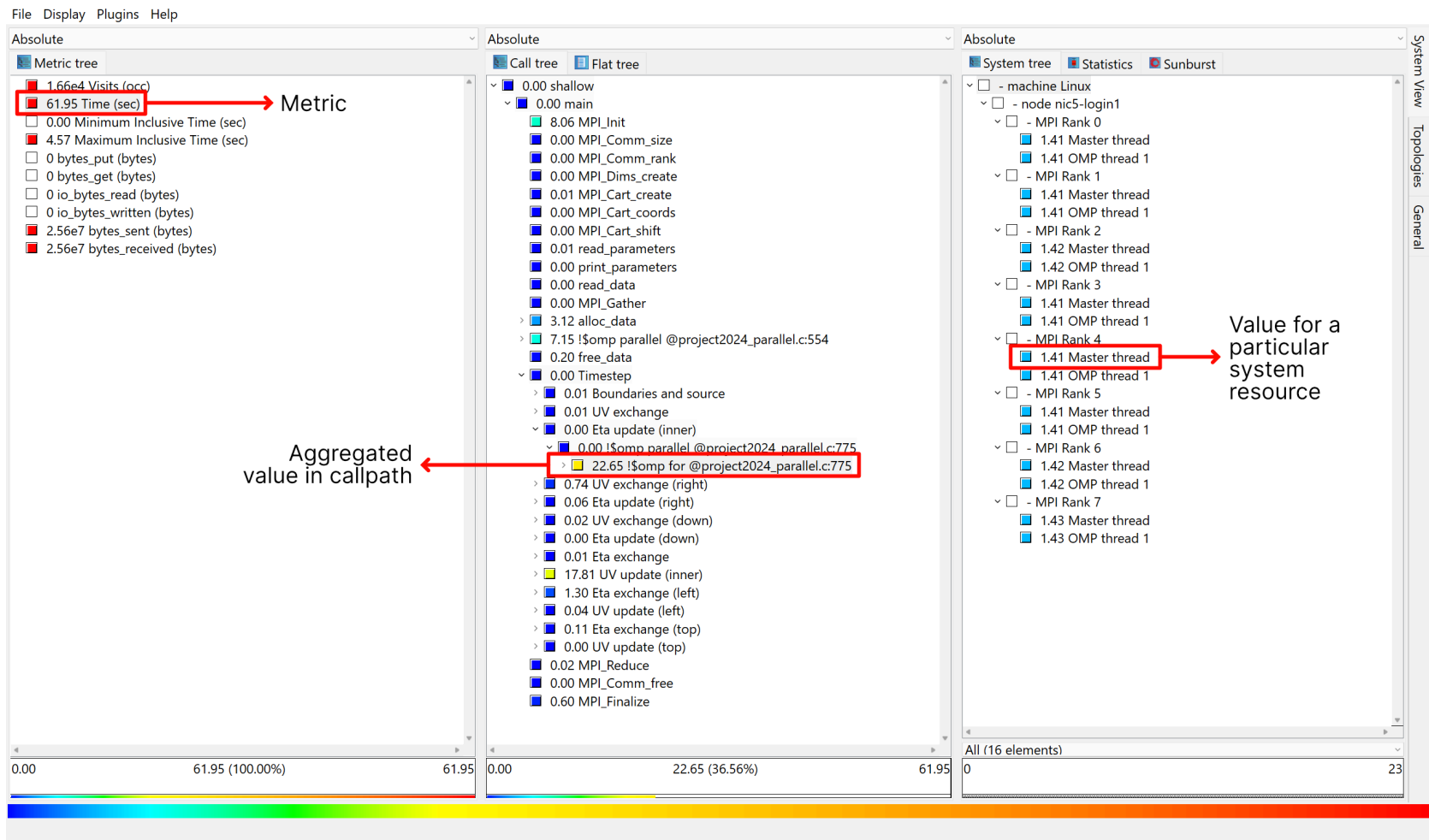
```
mpirun <application> <args>
```

At the end of the execution of the application, you should have a file in the experiment directory named `profile.cubex`

The generated `profile.cubex` located in the experiment directory can be downloaded and opened using CubeGUI. Binary packages are available for all major operating systems (download links for [Windows](#), [MacOS](#) and [Linux](#))

The name of the tools comes from the fact that it displays performance information in a three-dimensional performance space consisting of the dimensions

- performance metric
- call path
- system resource



Automatic instrumentation can generate extremely large traces—especially for fine-grained functions or frequently executed code paths. Filters help focus on the parts of the application that matter for analysis

Score-P provide a filtering mechanism that allows you to define in a filter file with include or exclude rules to prevent uninteresting or high-frequency functions from being recorded. The goal is twofold

- reduce the volume of performance data collected during instrumentation by selectively including or excluding specific functions from the measurement
- make the observed performance behavior more representative of the application true execution: the tracing experiment execution time should be close to the non-instrumented time

Score-P provide the `scorep-score` tool to evaluate the size of the trace that will be generated by a tracing experiment from a cube file generated by a profiling run:

```
$ scorep-score profiling/profile.cubex
```

```
Estimated aggregate size of event trace:                20GB
Estimated requirements for largest trace buffer (max_buf): 5GB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):    5GB
(warning: The memory requirements cannot be satisfied by Score-P to avoid
intermediate flushes when tracing. Set SCOREP_TOTAL_MEMORY=4G to get the
maximum supported memory or reduce requirements using USR regions filters.)
```

flt	type	max_buf[B]	visits	time[s]	time[%]	time/visit[us]	region
	ALL	5,201,103,627	800,165,477	193.22	100.0	0.24	ALL
	USR	5,201,040,286	800,160,041	82.18	42.5	0.10	USR
	OMP	36,974	2,952	105.04	54.4	35582.24	OMP
	MPI	16,810	1,016	6.00	3.1	5907.33	MPI
	COM	9,516	1,464	0.01	0.0	4.00	COM
	SCOREP	41	4	0.00	0.0	34.81	SCOREP

Using the `-r` option of `scorep-score` to display recorded regions and their contribution to the trace can guide the creation of an effective filter file:

```
$ scorep-score -r profiling/profile.cubex
```

flt	type	max_buf[B]	visits	time[s]	time[%]	time/visit[us]	region
	USR	5,201,040,052	800,160,008	82.06	42.5	0.10	interpolate_data
	MPI	7,120	320	0.00	0.0	5.01	MPI_Irecv
	MPI	7,120	320	0.00	0.0	5.91	MPI_Isend
	MPI	2,080	320	3.64	1.9	11374.65	MPI_Waitall

In this example, the `interpolate_data` function has a major impact on the trace because it is invoked more than 800 million times, even though each call is very short ($< 1 \mu\text{s}$)

A Score-P filter file is a text file in which you specify the regions to exclude from instrumentation. In our example, we create a file named `myapp.scorep-filt` with the following content in order to exclude the `interpolate_data` data from the measurements:

```
$ cat myapp.scorep-filt  
  
SCOREP_REGION_NAMES_BEGIN  
EXCLUDE  
    interpolate_data  
SCOREP_REGION_NAMES_END
```

We can evaluate the effect of the filtering with the `scorep-score` tool using the `-f` option

```
$ scorep-score -f myapp.scorep-filt profiling/profile.cubex
```

```
Estimated aggregate size of event trace:          238kB  
Estimated requirements for largest trace buffer (max_buf): 63kB  
Estimated memory requirements (SCOREP_TOTAL_MEMORY): 4097kB  
(hint: When tracing set SCOREP_TOTAL_MEMORY=4097kB to avoid intermediate flushes  
or reduce requirements using USR regions filters.)
```

The size of the trace file was reduced from 20 GB to 238 kB!

We are now ready to run our tracing experiment. To do so, configure the environment using the following variables:

```
export SCOREP_ENABLE_PROFILING=false
export SCOREP_ENABLE_TRACING=true
export SCOREP_EXPERIMENT_DIRECTORY=tracing
export SCOREP_TOTAL_MEMORY=4097kB
export SCOREP_FILTERING_FILE=myapp.scorep-filt
```

Here, SCOREP_TOTAL_MEMORY corresponds to the value recommended by the scorep-score tool (see previous slide)

With the environment configured, we can now run our application as usual. The results will be stored in the `tracing` directory, as specified by the SCOREP_EXPERIMENT_DIRECTORY environment variable

Score-P generates trace files in the Open Trace Format (OTF2), which can be visualized with tools such as Vampir. However, Vampir requires a license that is not available on NIC5. To support trace visualization, we provide a small utility that performs a rough conversion from OTF2 to the Perfetto trace format. This tool, `otf2perfetto`, can be used as follows:

```
otf2perfetto tracing/traces.otf2 traces.pftrace
```

The resulting `traces.pftrace` file can be downloaded and opened in your browser using the [!\[\]\(e2376d476d06eb31946dc01a69a4403a_img.jpg\) Perfetto UI](#) for visualization

Note: The `otf2perfetto` tool is still in early development and may be slow when converting traces, as well as produce very large output files. We therefore recommend limiting your tracing experiments to a maximum of about 100 time steps



User instrumentation in Score-P allows developers to manually mark specific regions of their code for performance measurement. This is particularly useful when most of the computation is concentrated in large functions (like `main`), where automatic instrumentation alone may provide limited insight into finer-grained behaviors

```
#ifndef SCOREP_USER_ENABLE
#include <scorep/SCOREP_User.h>
#else
#define SCOREP_USER_REGION_DEFINE(handle)
#define SCOREP_USER_REGION_ENTER(handle)
#define SCOREP_USER_REGION_INIT(handle, name, type)
#define SCOREP_USER_REGION_END(handle)

#define SCOREP_USER_REGION_TYPE_COMMON
#define SCOREP_USER_REGION_TYPE_LOOP
#endif
```

To begin the instrumentation process, insert the code snippet above at the top of your application's source file.

The next step is to define and initialize your regions of interest using the SCOREP_USER_REGION_DEFINE and SCOREP_USER_REGION_INIT macro. Region corresponding to a loop can be marked as SCOREP_USER_REGION_TYPE_LOOP at the initialization stage

Then, within your time-stepping code, mark the beginning and end of each region using SCOREP_USER_REGION_ENTER and SCOREP_USER_REGION_END

```
SCOREP_USER_REGION_DEFINE(apply_source)
SCOREP_USER_REGION_DEFINE(update_step)

SCOREP_USER_REGION_INIT(apply_source,
    "Apply source", SCOREP_USER_REGION_TYPE_COMMON)
SCOREP_USER_REGION_INIT(update_step,
    "Update step", SCOREP_USER_REGION_TYPE_LOOP)

for (int tstep; tstep < num_timestep; tstep++) {
    SCOREP_USER_REGION_ENTER(apply_source)
    // code to apply the source
    SCOREP_USER_REGION_END(apply_source)

    SCOREP_USER_REGION_ENTER(update_step)
    #pragma omp parallel for
    for (int ix = 0; ix < nx; ix++) {
        // code for the timestep update
    }
    SCOREP_USER_REGION_END(update_step)
}
```


By default, user instrumentation is disabled. You can enable it at compile time using the `scorep` command `--user` option:

```
scorep --user gcc/mpicc <compile_flags>
```


Once user instrumentation is enabled, the remainder of the Score-P workflow is identical to that used with only automatic compiler instrumentation

Note: as with automatic compiler instrumentation, avoid creating short-lived user-defined regions that are called frequently. Doing so can significantly increase measurement overhead and trace size, as illustrated in the previously discussed example with `interpolate_data`

Scalasca is a performance analysis tool for HPC applications which focus on scaling and communication performance. It's built on top of Score-P, relying on instrumentation to collect performance data. Key features include:

- Identification of load imbalance and communication bottlenecks
- Supports call-path aware analysis for both computation and MPI communication

Scalasca analyze a trace file and produce a report as a CUBE file. This cube file contains key metrics to understand the potential parallel bottleneck of a parallel application

The report contains a list of  **performance properties** that may limit the scalability of the application, along with the code regions and hardware resources responsible for them.

Scalasca's workflow is built on top of Score-P, so most of the steps presented earlier still apply:

- Run an initial profiling session to evaluate the expected trace size and assess measurement overhead.
- Create a filtering file if needed to keep the trace at a manageable size and avoid excessive overhead.
- Determine the required memory buffer size for trace collection from this initial run, and export the recommended value to the environment before launching Scalasca:

```
export SCOREP_TOTAL_MEMORY=<value_suggested_by_scorep_score>
```

This ensures that Scalasca has sufficient buffer space to collect trace data efficiently.

To run a Scalasca analysis, use the following commands:

```
module load Info0939Tools
scalasca -analyze -q -t -f myapp.scorep-filt srun <instrumented_application> <args>
```

Here, the `-f <filter_file>` option specifies the Score-P filtering file, while `-q -t` enables tracing mode

The analysis results will be stored in a directory named following this pattern: `scorep_<appname>_<numranks>x<numthreads>_trace`. To generate the final Scalasca report, run:

```
scalasca -examine -s scorep_<appname>_<numranks>x<numthreads>_trace
```

The experiment directory will now contain a `trace.cubex` file, which can be downloaded and visualized with CubeGUI

When presenting results from profiling tools (e.g., LIKWID, Score-P, Scalasca), ensure they serve a clear purpose. Do not include profiling data merely to “check a box”

Examples of the level of explanation expected:

“We performed profiling and found from **insert metric**, that **insert observation** was limiting the performance of our code. Based on this, we applied **insert improvement** and then re-measured. The results showed a **insert quantification** improvement.”

or

“We performed profiling and found from **insert metric**, that **insert observation** was limiting the performance of our code. Although we did not have time to address it, we believe that applying **insert potential improvement** would likely enhance performance”

Performance measurement on GPU with NVIDIA Tools

NVIDIA Nsight System is system-wide performance analysis tool designed to help developers optimize CPU/GPU interactions. Key features include

- Visualization of CPU–GPU workloads across an entire system
- Identifies performance bottlenecks, synchronization issues, and inefficient kernel launches
- Supports multi-threaded and multi-process profiling
- Integrates with CUDA, but also with graphics API like Vulkan and DirectX

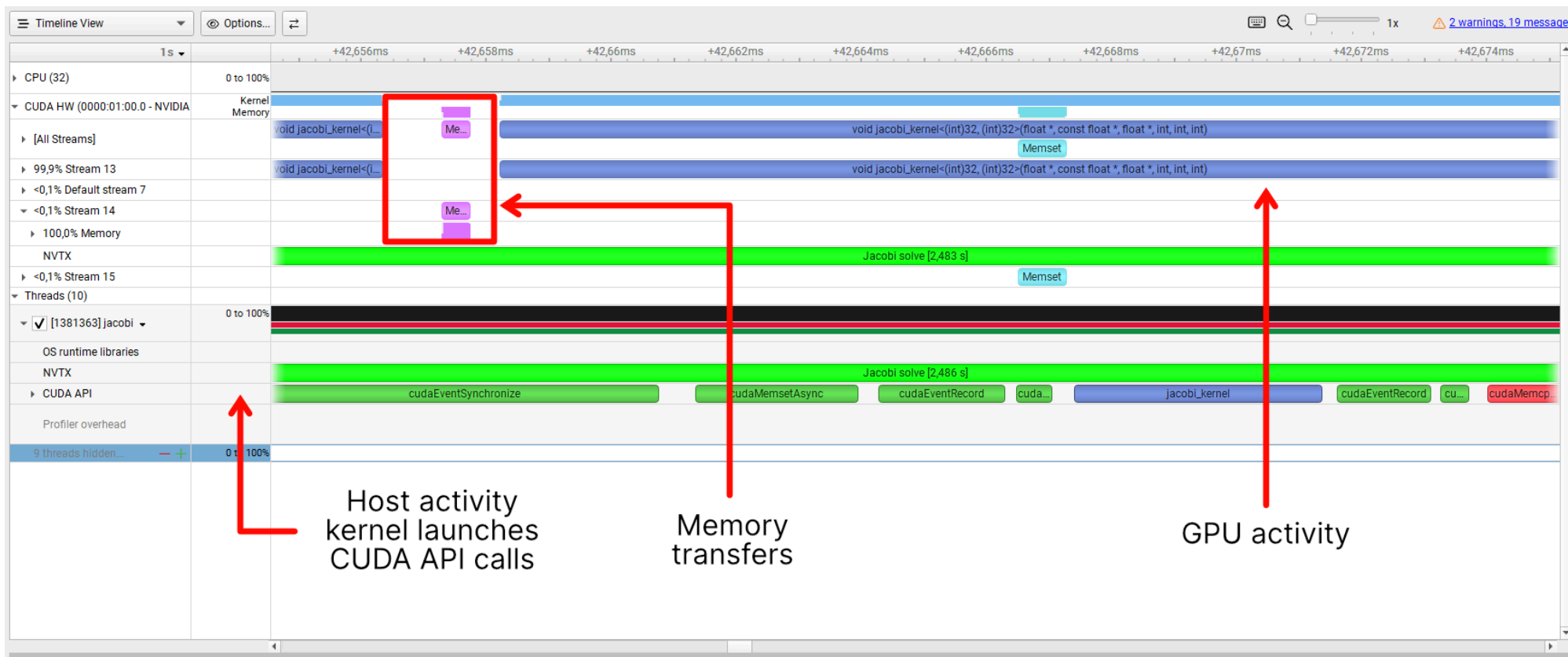
To capture a performance trace with Nsight System, run the following command

```
nsys profile -o <output_name> <application> <args>
```

After profiling, you can generate statistical summaries of kernel launches, memory operations, and API calls:

```
nsys stats <output_name>.nsys-rep
```

This provides detailed information such as GPU kernel execution times, API call counts, and memory transfer statistics. For a more comprehensive visualization, you can download the `nsys-rep` file and open it in the [Nsight Systems GUI](#)



NVIDIA Nsight Compute is a GPU-focused kernel profiling tool designed to help developers analyze and optimize CUDA kernel performance. Key features include:

- Detailed metrics for GPU kernel execution, memory utilization, and instruction efficiency
- Identifies performance bottlenecks at the kernel level, such as memory stalls or low occupancy
- Supports per-kernel analysis, allowing comparison of multiple kernel launches

Before running your application with NSight Compute, compile your application with the `-lineinfo` compiler flag

```
module load CUDA
nvcc -lineinfo <other_compiler_flags>
```

Then, run your application with NSight Compute

```
ncu --set full -o <output_name> <application> <args>
```

To collect all the metrics, Nsight Compute runs the kernel multiple time (replay): **limit your application to 10 timesteps**

To visualize the result, you can download the `ncu-rep` file and open it in the [Nsight Compute GUI](#)

GPU Speed Of Light Throughput

Roofline Single Precision

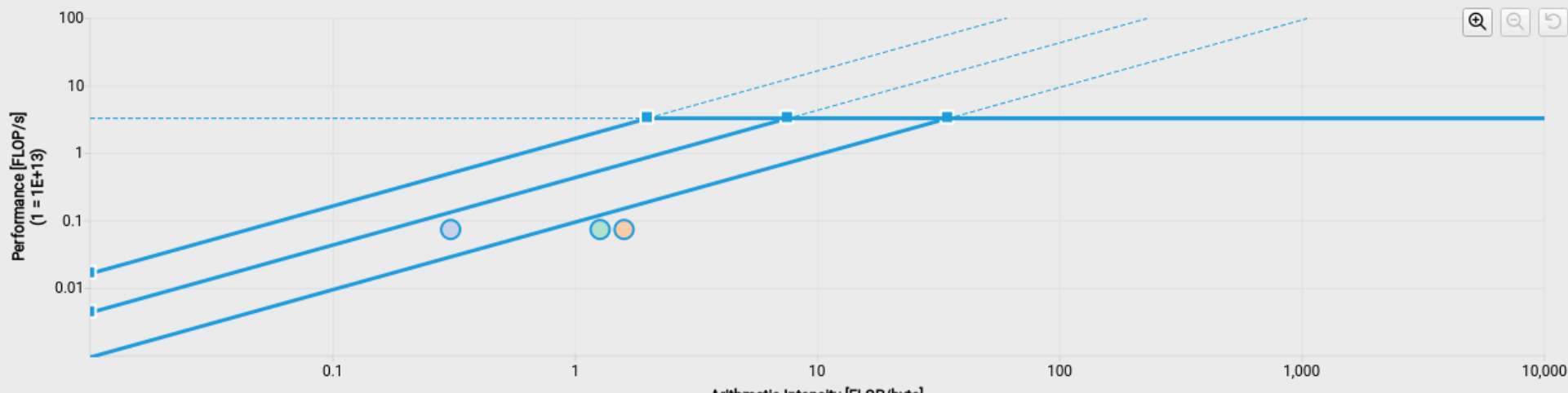
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.


Compute (SM) Throughput [%]	59.20	Duration [ms]	3.77
Memory Throughput [%]	59.20	Elapsed Cycles [cycle]	3,448,587
L1/TEX Cache Throughput [%]	65.86	SM Active Cycles [cycle]	3,099,797.48
L2 Cache Throughput [%]	21.46	SM Frequency [Mhz]	914.99
DRAM Throughput [%]	58.24	DRAM Frequency [Ghz]	9.99


Latency Issue This workload exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues. Look at [Scheduler Statistics](#) and [Warp State Statistics](#) for potential reasons.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The workload achieved 2% of this device's fp32 peak performance and 0% of its fp64 peak performance. See the [Kernel Profiling Guide](#) for more details on roofline analysis.

Floating Point Operations Roofline (Single Precision)

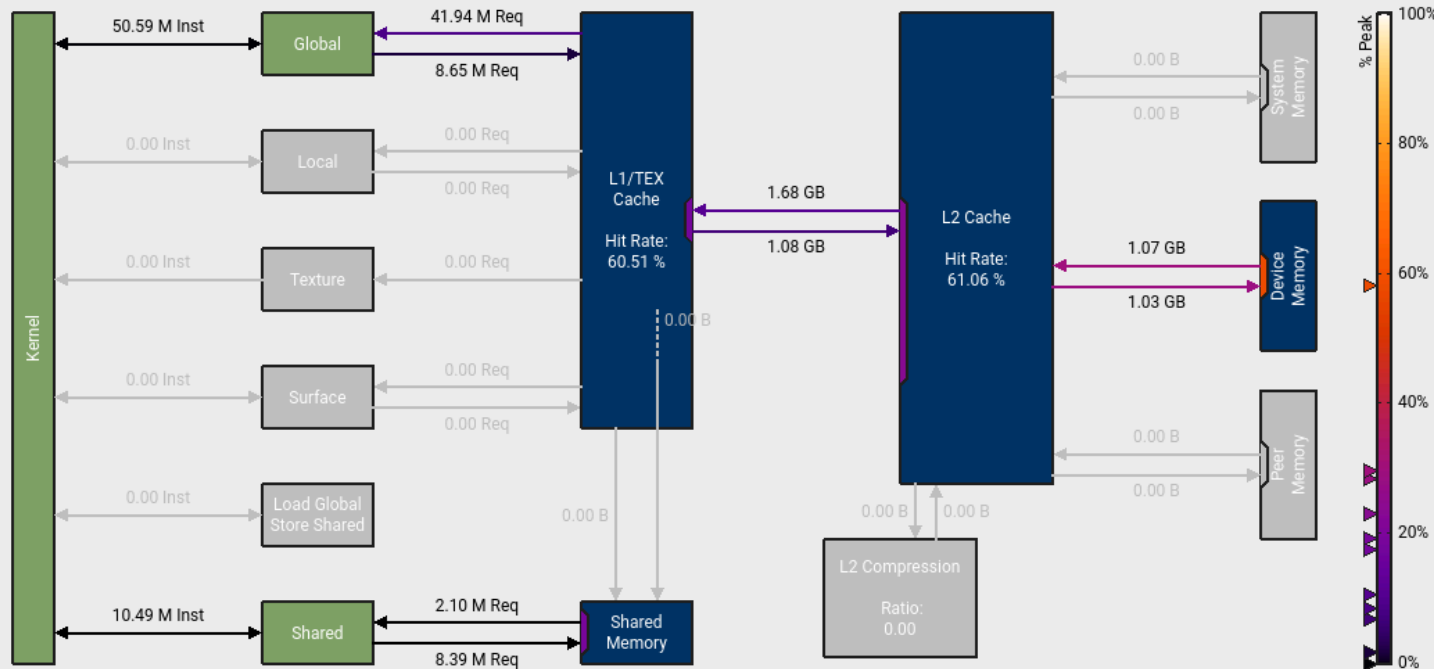


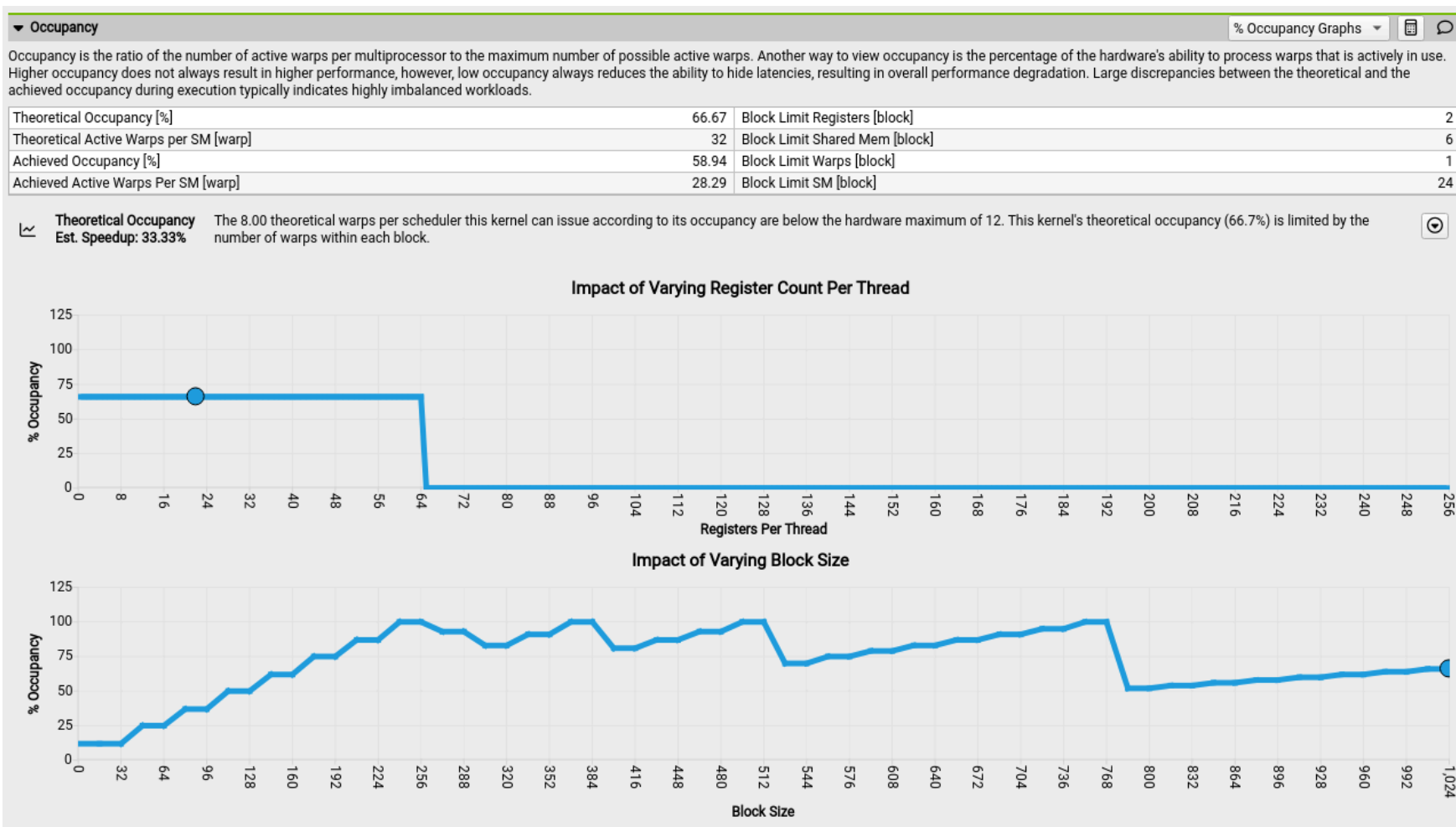
 **L1TEX Global Load Access Pattern** Est. Speedup: 5.38% The memory access pattern for global loads from L1TEX might not be optimal. On average, only 29.1 of the 32 bytes transmitted per sector are utilized by each thread. This could possibly be caused by a stride between threads. Check the [Source Counters](#) section for uncoalesced global loads.

 **L1TEX Global Store Access Pattern** The memory access pattern for global stores to L1TEX might not be optimal. On average, only 32.0 of the 32 bytes transmitted per sector are utilized by each thread. This could possibly be caused by a stride between threads. Check the [Source Counters](#) section for uncoalesced global stores.

Memory Chart

Values: Inactivity:





Summary Details **Source** Context Comments Raw Session

View: Source and SASS

Source: jacobi.cu Navigate By: L2 Theoretical Sectors Global Excessive

Redo Resolve

#	Source	Press Size	L1 Wavefronts Shared Excessive	heoretical Sectors Global Excessive
102	#ifdef HAVE_CUB			
103	typedef cub::BlockReduce<real, BLOCK_I			
104	BlockReduce;			
105	__shared__ typename BlockReduce::TempS			
106	#endif // HAVE_CUB			
107	const int iy = blockIdx.y * blockDim.			
108	const int ix = blockIdx.x * blockDim.			
109	real local_l2_norm = 0.0;			
110				
111	if (iy < iy_end) {			
112	if (ix >= 1 && ix < (nx - 1)) {			
113	const real new_val = 0.25 * ((2)		100.00%
114	a[(2)		
115	a_new[iy * nx + ix] = new_val	32		
116				
117	// apply boundary conditions			
118	if (iy_start == iy) {			
119	a_new[iy_end * nx + ix] =	32		
120	}			
121				
122	if ((iy_end - 1) == iy) {			
123	a_new[(iy_start - 1) * nx	32		
124	}			
125				
126	real residue = new_val - a[iy	32		
127	local_l2_norm = residue * resi			

Source: jacobi_kernel Navigate By: L2 Theoretical Sectors Global Excessive

#	Address	Source	Press Size	L1 Wavefronts Shared Excessive	heoretical Sectors Global Excessive
jacobi_kernel					
1	00001555 03268900	MOV R1, c[0x0][0x28			
2	00001555 03268910	S2R R2, SR_CTAID.X			
3	00001555 03268920	MOV R16, c[0x0][0x1			
4	00001555 03268930	ULDC .64 UR6, c[0x0]			
5	00001555 03268940	BSSY B0, 0x155503268			
6	00001555 03268950	S2R R3, SR_TID.X			
7	00001555 03268960	IADD3 R4, R16, -0x1			
8	00001555 03268970	MOV R12, RZ			
9	00001555 03268980	S2R R5, SR_CTAID.Y			
10	00001555 03268990	S2R R0, SR_TID.Y			
11	00001555 032689a0	IMAD R11, R2, c[0x0			
12	00001555 032689b0	ISETP.GE.AND P0, P1			
13	00001555 032689c0	IMAD R2, R5, c[0x0]			
14	00001555 032689d0	ISETP.LT.OR P0, PT			
15	00001555 032689e0	IADD3 R10, R2, 0x1,			
16	00001555 032689f0	ISETP.GE.OR P0, PT			
17	00001555 03268a00	@P0 BRA 0x155503268c10			
18	00001555 03268a10	IMAD R12, R10, c[0x			
19	00001555 03268a20	MOV R15, 0x4			
20	00001555 03268a30	IMAD R8, R2, c[0x0]			
21	00001555 03268a40	IADD3 R6, R12, c[0x			
22	00001555 03268a50	IMAD.WIDE R4, R12,			
23	00001555 03268a60	IMAD.WIDE R6, R6, P			
24	00001555 03268a70	LDG.E.CONSTANT R2,	32		50.00%
25	00001555 03268a80	LDG.E.CONSTANT R13,	32		50.00%
26	00001555 03268a90	IMAD.WIDE R8, R8, P			