🔥 Hotspots Overview

Hotspot profiling identifies which functions are using the most CPU time, helping you understand where your program spends most of its processing power.

⏱️ Elapsed Time vs. CPU Time

Metric	Meaning
Elapsed Time	Total time the application ran (wall clock) → **0.162 seconds**
CPU Time	Time CPU actually worked on the program (all threads summed) → **0.150 seconds**
Effective Time	Time spent doing useful work (same as CPU time here) → **0.150 seconds**
Spin Time	Time threads were active but not doing useful work (e.g., waiting in a loop) → **0 seconds**
Overhead Time	Time spent due to profiling itself or system overhead → **0 seconds**
Paused Time	Time when VTune paused data collection → **0 seconds**
Total Thread Count	Threads used by your application → **4 threads**

🔍 Top Hotspots (functions using the most CPU time)

Function	Module	CPU Time	% of Total CPU Time
`func@0x31d9d0`	libcuda.so.1	**0.071s**	**47.0%**
`memset`	libc-dynamic.so	0.050s	33.3%
`func@0x328160`	libcuda.so.1	0.021s	13.9%
`func@0x328340`	libcuda.so.1	0.009s	5.7%

✅ **What this tells you**:

- Most CPU time (47%) is being spent in a CUDA function (`libcuda.so.1`) — which means a GPU-related part of your code (or the CUDA runtime) is taking time.

- `memset` from the standard C library also takes a good chunk — this could be due to initializing memory buffers.

If you expected your CPU to do more work (and not the GPU), this might point to performance tuning opportunities.

📊 CPU Utilization Histogram

This shows **how many logical CPUs were being used simultaneously** during the profiling run:

Logical CPUs Simultaneously Used	Elapsed Time	Utilization Category
0 CPUs	0.003s	Idle
1 CPU	0.159s	Poor
2–6 CPUs	0s	Poor–Ideal

❗ **What this means**:

- Your app is mostly **single-threaded** — most of the time, **only 1 CPU was active**, even though your system has 6 logical CPUs.
- If your app is designed to use multiple threads or cores, this indicates **low parallelism**.
- Optimization tip: consider adding multithreading (OpenMP, TBB, etc.) if appropriate.

🖥️ System Info

Property	Value
OS	Ubuntu 22.04.5 LTS
Kernel	6.8.0-57-generic
VTune Command Line	`/home/it/23620006/vect`
CPU	Intel Coffee Lake @ 2.8 GHz
Logical CPUs	6

> “Coffee Lake” is your CPU’s microarchitecture — common in 8th/9th Gen Intel Core processors.

🧠 Cache Allocation Technology

- **Not detected** — This feature helps tune cache use (like Level 3 cache partitioning), but your system or CPU doesn't support it or it's not enabled.

✅ TL;DR Summary

- Your app ran for 0.162 seconds and used 4 threads, but **only 1 thread (CPU) was doing most of the work.**
- Almost half the time was spent in CUDA code (`libcuda.so.1`) — indicating GPU usage.
- There is **low CPU core utilization**, suggesting **no real parallelism**.
- Memory operations (`memset`) took a significant chunk — maybe optimize memory usage or avoid large unnecessary initializations.

📦 Memory Consumption Overview

Metric	Explanation
Elapsed Time	Total run time of the application → **0.193 seconds**
Allocation Size	Total memory allocated by the program → **9.3 GB**
Deallocation Size	Memory freed during execution → **289.4 MB**
Allocations	Number of separate memory allocation calls → **7,252**
Total Thread Count	Threads used by your app → **4**
Paused Time	Time when VTune paused data collection → **0 seconds**

> ⚠️ Big warning: You allocated **9.3 GB**, but only freed **289.4 MB** — this shows a potential **memory leak** or very short-lived execution (or manual memory not freed properly).

🔝 Top Memory-Consuming Functions

Function Hash	Memory Used	Net Increase	Allocations	Module
`libcudart_static_5449...`	**4.7 GB**	**+4.5 GB**	6,759	`vect`
`libcudart_static_aa4a...`	**4.4 GB**	**+4.4 GB**	203	`vect`
`libcudart_static_5382...`	100.7 MB	+384 B	57	`vect`
`libcudart_static_8b73...`	12.0 MB	0 B	57	`vect`
`libcudart_static_3874...`	12.0 MB	12.0 MB	3	`vect`
`[Others]`	209.8 KB	195.3 KB	173	N/A

📌 Key Observations:

- All top allocations are happening **inside CUDA runtime** (`libcudart_static...` functions).
- **~9.1 GB of memory is still held at the end of the run** → these memory blocks are not deallocated (leak or lifetime too long).

- Only **289 MB was released** → dangerous for long runs or large datasets.

🧠 Analysis Summary

Insight	Implication
🔴 **High Allocation Size (9.3 GB)**	Your program demands **a lot of memory**, possibly due to image/video data, deep learning tensors, etc.
⚠️ **Low Deallocation Size (289.4 MB)**	Suggests **memory leak** or **not freeing memory** properly after use.
🔁 **Many Allocations (7252)**	Frequent calls to `malloc/new/cudaMalloc` — overhead from lots of small allocations.
💡 **Concentration in CUDA functions**	Most allocations happen in GPU-related code — this might be fine if managed correctly, but memory reuse or cleanup should be verified.

🖥️ System Info

Component	Info
OS	Ubuntu 22.04.5 LTS (kernel 6.8.0-57-generic)
CPU	Intel Coffee Lake @ 2.8 GHz
Logical CPUs	6 cores (hyperthreaded or physical)
Cache Allocation Tech	Not supported or not enabled (L2/L3 not detected)
VTune Mode	Fast mode (limits sample processing for speed)

🛠️ Suggestions for Improvement

✅ If You're Using CUDA:

- Reuse memory (e.g. **use memory pools** instead of frequent `cudaMalloc`).
- Explicitly **free GPU memory** with `cudaFree()` after use.
- Use **CUDA Unified Memory** carefully — it can hide leaks if not managed right.

✅ General Best Practices:

- Use **memory profiling tools like `valgrind` or `cuda-memcheck`** for deeper leak detection.
- Monitor **peak memory usage** in larger inputs.
- Minimize small allocations — combine buffers if possible.

🧾 TL;DR:

- 🚨 You're allocating 9.3 GB but only freeing 289 MB → high chance of **memory leak** or **not releasing GPU memory**.
- 🔍 Most allocations are from **CUDA runtime** (`libcudart_static_*`) — GPU work.
- 🧠 Optimize memory reuse and ensure all GPU allocations are properly released.

Here's a comprehensive analysis and interpretation of the **VTune Profiler Memory Access report** you provided:

📋 Summary:

Metric	Value
Elapsed Time	0.105s
CPU Time	0.093s
Threads Used	5
Loads	108.8 million
Stores	58.7 million
Memory Bound	16.5%
LLC Miss Count	0
Avg Latency (cycles)	14
Max Bandwidth (DRAM)	18.0 GB/s (hardware limit)
Collection Method	Driverless Perf sampling
Paused Time	0s
Finalization Mode	Fast

🧠 Interpretation & Insights

🔸 1. Memory Bound: 16.5%

This means that **16.5% of CPU pipeline slots were stalled** due to waiting on memory operations. A memory-bound value under 20% is generally **not a bottleneck**, so your application is **not heavily limited by memory latency or bandwidth**.

✅ **Good news:** Your code is likely **CPU compute-bound**, not limited by memory throughput.

🔸 2. Load & Store Analysis

- **Loads**: 108,809,477
- **Stores**: 58,705,698

You performed almost **twice as many reads as writes**, which is typical for algorithms involving large data processing, like image manipulation, matrix computation, or numerical simulations.

🔸 3. Latency Histogram (Load Latency in CPU cycles)

Latency (cycles)	Loads
6–8	~950K
9	129,856
24	129,856
51	129,856
77	30,162

Most memory accesses are **very fast (6–9 cycles)** → indicates **cache hits** (likely L1 or L2).

A small number of accesses show **higher latency (24–77 cycles)** → might represent **LLC (Last-Level Cache)** or even **DRAM** accesses.

✅ **LLC Miss Count: 0** — Suggests **excellent cache locality** or efficient prefetching. Nice!

🔸 4. Bandwidth Utilization Histogram

Bandwidth (GB/s)	Time Spent (seconds)	Level
0–5 GB/s	< 0.1s (total)	Low/Medium
>6 GB/s	~0.03s (peaking)	Medium to High

🚨 **Observation:** Despite DRAM having a max bandwidth of **18 GB/s**, your app never utilized more than **~7–8 GB/s**, and for most of the time, it's **well under 5 GB/s**.

✅ Again, confirms **not memory bandwidth bound**.

⚙️ System and Platform

- **CPU**: Intel Coffeelake, 2.8 GHz, 6 logical cores
- **DRAM Max Bandwidth**: 18.0 GB/s (as per VTune hardware database)
- **VTune Mode**: Driverless collection using `perf` (you didn't install Intel drivers — which is okay for most use-cases)

- **OS**: Ubuntu 22.04.5 LTS
- **App**: `/home/it/23620006/vect`

✅ Final Thoughts & Recommendations

Your app shows:

- ✔️ Efficient memory access patterns
- ✔️ Low memory latency
- ✔️ Very few stalls due to memory
- ✔️ Not bandwidth bound
- ✔️ Zero LLC misses

Suggestions:

- Focus on **compute efficiency**, **thread-level parallelism**, and **vectorization** rather than optimizing memory bandwidth.
- You may use VTune's **"Microarchitecture Exploration"** or **"Hotspots"** analysis to dig deeper into CPU-bound performance bottlenecks (like underutilized ALUs, bad vectorization, or poor ILP).

Based on your Intel VTune Profiler output, here's a **complete summary and performance characterization** of your **HPC (High-Performance Computing)** workload:

🕑 Execution Summary

- **Elapsed Time:** 0.105 seconds
- **CPU Time:** 0.093 seconds
- **Total Threads Used:** 5
- **Paused Time:** 0 seconds

⚙️ CPU Performance Metrics

- **Average CPU Frequency: 3.8 GHz**
- **CPI (Cycles per Instruction): 0.856** → Indicates relatively efficient instruction execution (lower CPI is better).
- **SP (Single Precision) GFLOPS:** 0.000
- **DP (Double Precision) GFLOPS:** 0.000
- **x87 GFLOPS:** 0.000
- **Instruction Mix:**
  - **SP FLOPs:** 0.0%
  - **DP FLOPs:** 0.0%
  - **x87 FLOPs:** 0.0%
  - **Non-FP (Integer/Control):** 100.0%
- **Vectorization:** 0.0% of packed FP operations
  - Suggests no vectorized floating-point math; compute-heavy vector instructions are not being used.

🔄 CPU Utilization

- **Effective CPU Utilization:** 15.3% (very low overall)
- **Average Logical CPUs Used:** 0.918 out of 6
    - Indicates under-utilization of available parallel resources.
- **Utilization Histogram:**

Logical CPUs	Time (s)	Classification
0	0.00898	Idle
1	0.09643	Poor
2–6	0	Poor to Ideal

🧠 Memory System

- **Memory Bound:** 0.0% → Memory is **not** a performance bottleneck overall.
- **Cache Bound:** 43.6% → Significant time spent waiting for **cache**, which may indicate poor data locality.
- **DRAM Bound:** 5.5% → DRAM access is not a major bottleneck.

📊 Bandwidth Utilization Histogram (DRAM, GB/s):

Bandwidth (GB/s)	Elapsed Time (s)	Utilization
0–5	Mostly present	Low
6–13	Minimal	Medium
14–20	0	High

→ **Observation:** Memory bandwidth usage is low to medium throughout. The app is **not saturating DRAM bandwidth**.

📦 Loads and Stores

- **Total Loads:** 108,809,477
- **Total Stores:** 58,705,698
- **LLC (Last Level Cache) Miss Count:** 0 → Very good cache hit performance.
- **Average Memory Access Latency:** 14 cycles
    - Latency histogram shows most memory accesses are relatively fast (6–9 cycles), with some long tails (24–77 cycles).

🔬 Observations and Recommendations

🔍 Aspect	Observation	Recommendation
CPU Utilization	Very low utilization (15.3%), under 1 logical CPU used on average	Parallelize the workload using threads (OpenMP, TBB, or MPI) or increase input problem size.
Floating Point Usage	No floating-point operations are used	If compute-intensive tasks are expected, ensure that vectorized FP operations are implemented properly.
Cache Bound	High cache-bound percentage (43.6%)	Improve **data locality**, reduce cache misses, and consider cache blocking strategies.
DRAM Bandwidth	Low to medium, DRAM not saturated	No optimization needed unless you're optimizing memory-intensive applications.
CPI	0.856 (efficient)	Good CPU execution efficiency, try to maintain or reduce CPI further.

✅ Final Verdict

Your workload is:

- **Not compute-bound** (0 GFLOPS, no vectorization)
- **Not memory-bound** (only 5.5% DRAM bound)
- **Mostly idle on cores**, using only one core efficiently

It appears to be a **lightweight task** or one that's **not yet optimized for HPC**. If you're working on a performance-critical HPC app, focus on:

1. **Parallelism** (threads or processes)
2. **Vectorization (SIMD)** with compiler flags like `-O3 -march=native -ftree-vectorize`
3. **Data locality and cache optimization**

🔍GPU Offload

Metric	Value
⏱ **Elapsed Time**	0.359 seconds
⚙️ **GPU Time (% of Elapsed)**	2.0% (≈ 0.007 seconds)
🧠 **Top CPU Functions While GPU Idle**	`func@0x31d9d0`, `memset`, etc.

📌 Detailed Explanation

1. GPU Usage is Minimal (Only 2%)

- **What it means:** Your application is **barely using the GPU**.
- **Render and GPGPU Engine** (GPU unit responsible for graphics and computation) was active for only **0.007 seconds**, which is **just 2%** of the total run time.

➡️ This suggests your workload is **mostly CPU-bound**, not offloaded to the GPU.

---

2. Top CPU Hotspots While GPU Was Idle

These functions were active when the GPU wasn't doing any work:

Function	Module	Time
`func@0x31d9d0`	`libcuda.so.1`	0.060 s
`memset`	`libc-dynamic.so`	0.050 s

`func@0x37a050`	`libcuda.so.1`	0.010 s

- **`func@0x31d9d0` and `func@0x37a050` (libcuda.so.1):** These are from NVIDIA's CUDA runtime. They're likely driver or runtime-level functions interacting with your GPU, possibly waiting or syncing data.
- **`memset`:** Suggests memory initialization is taking up a noticeable portion of CPU time—common in CPU-intensive workloads.

➡️ **Your GPU is mostly idle**, while CPU functions (especially from `libcuda.so.1`) are running—possibly preparing or transferring data, or waiting on the GPU.

🔧 What Can You Do Next?

✅ If You Intend to Use GPU Acceleration:

You should:

1. **Profile your CUDA code** using `nvprof` or `nsight` to get deeper GPU kernel-level insights.
2. Make sure:
    - You're actually launching CUDA kernels (`cudaLaunchKernel`, etc.).
    - GPU computations are **not too small** (very fast tasks don't benefit from GPU due to overhead).
    - Memory transfers between CPU ↔ GPU are minimized and efficient.

❌ If You Don't Use GPU on Purpose:

Then this report just confirms that most of your application runs on the **CPU**, and the GPU is not doing meaningful work.

📈 Summary

- **2% GPU usage** indicates **very little offloading** is happening.
- **Most work is on the CPU**, especially in CUDA support libraries and memory operations.

Thanks for sharing your **GPU Compute/Media Hotspots** report from Intel VTune. Here's a breakdown of what's happening, explained clearly:

📊 Summary of Your Profiling Report

Metric	Value
⏱ **Elapsed Time**	0.341 seconds
🔧 **GPU Time**	0.005s (1.5%)
🧠 **Top CPU Functions When GPU Was Idle**	`func@0x8982b0`, `memset`, `__memmove_sse2_unaligned_erms`
💾 **GPU Memory Bandwidth Usage**	❌ No significant activity (all in the **Low** range)

🔍 What This Means

1. Very Low GPU Activity

- **Only 1.5%** of total time was spent doing GPU work.
- The GPU engine involved: **Render and GPGPU** — which supports both graphics and general-purpose computing (like CUDA/OpenCL).

2. GPU Was Idle Most of the Time

While the GPU was idle, the CPU was busy running:

Function	Module	CPU Time

`func@0x8982b0`	`libtpsstool.so`	0.063s
`memset`	`libc-dynamic.so`	0.050s
`__memmove_sse2_unaligned_erms`	`libc.so.6`	0.007s

These functions are doing memory operations:

- `memset` and `memmove` are memory initialization and data movement.
- `libtpsstool.so` seems to be part of a **third-party or custom library**, likely CPU-heavy.

➡️ **Conclusion:** The workload is **CPU-bound**, not GPU-accelerated.

---

3. GPU Memory Bandwidth Usage = Very Low

You were provided with a **bandwidth utilization histogram** that shows:

✅ Your application **never exceeds 0 GB/sec** of GPU memory read bandwidth.

This means:

- No **significant data** is being transferred **to or from the GPU memory**.
- Even if the GPU was used briefly, it didn't need much memory bandwidth.

⚙️ Recommendations

✅ If You Intend to Use the GPU:

- Ensure **actual GPU compute kernels** are launched (CUDA/OpenCL, or via DPC++/SYCL if using Intel oneAPI).
- Try **offloading more heavy computation** to the GPU.

- Watch for small kernel launches — GPU overhead can outweigh benefits if your tasks are tiny.

🧪 Run with VTune GPU Tracing + Compute Detail

- Try running with **GPU Compute/Media Hotspots with Kernel Analysis**.
- Check if your kernels are launched, and whether they're memory-bound or compute-bound.

🧵 TL;DR

🔍 What Happened	📋 What It Means
GPU only used for 1.5% of time	Mostly CPU workload
GPU memory bandwidth = 0 GB/s	No significant data sent to GPU
CPU was busy in memory-related tasks	Code is memory-bound on CPU
`libtpsstool.so` may be a bottleneck	Could be optimized or GPU-accelerated