# guppy v1.2.0

exaberries

October 24, 2026

## Contents

**1 Introduction** **3**

**2 Options** **3**
2.1 Memory Requirements . . . . . 4

**3 General Optimization Advice** **4**

**4 HPC, Memory Access, and Caching** **5**
4.1 Stencil Computations in HPC . . . . . 5
4.2 Address Translation and the TLB . . . . . 5
4.3 Prefetching . . . . . 6
4.4 Stencil Computations and the Roofline Model . . . . . 6
4.5 Superscalar Execution, Data Dependencies, and SMT . . . . . 7
4.5.1 Superscalar Pipelines and Out-of-Order Execution . . . . . 7
4.5.2 Data Dependencies and Memory-Level Parallelism . . . . . 8
4.5.3 Simultaneous Multithreading (SMT) . . . . . 8

**5 Finite Difference Time Domain** **8**
5.1 Maxwell's Equations in Differential Form . . . . . 8
5.1.1 Vector Expansion . . . . . 9
5.2 The Yee Grid . . . . . 9
5.2.1 Timestep . . . . . 9
5.2.2 Numerical Stability (CFL Condition) . . . . . 9

**6 guppy** **9**
6.1 The Scaling Challenge . . . . . 10
6.1.1 Mitigation Strategies . . . . . 10
6.1.2 Throughput and Multithreading . . . . . 10
6.2 Solver implementation . . . . . 10
6.2.1 Loading from memory . . . . . 11
6.2.2 Shuffle/Swap and FLOPs . . . . . 11
6.2.3 Arithmetic Intensity . . . . . 11
6.3 Workloads . . . . . 12
6.3.1 standing_wave . . . . . 12
6.3.2 soft_gaussian . . . . . 12
6.4 Verification in Detail . . . . . 12
6.5 Scoring: MGups . . . . . 13

6.6 Timers . . . 13
6.7 Topology Detection . . . 15
6.8 Large Pages . . . 15
6.9 Thread Model . . . 15
6.10 Expected Scaling Characteristics . . . 15
6.11 Compiling . . . 16
6.12 Test Harness . . . 16

**A Testing Data** **17**

# 1 Introduction

guppy is based on the finite difference in time domain (FDTD) method of electromagnetic simulations of Maxwell's equations (Eq. 1). This workload, like some other similar 3d simulation HPC workloads ends up being very memory access bottlenecked with larger problems, which ends up being an interesting benchmark for competitive overclocking. This benchmark is not designed to be either an accurate fdtd solver, or simulate how many threads can work on a single simulation. Rather, it represents more of a upper throughput bound of multiple simultaneous simulations. This documentation will cover the rationale behind why FDTD is a good workload for competitive benchmarking, and the other choices made related to multithreading, and scoring.

# 2 Options

```
USAGE:
  guppy-1.2.0 [-?|-h|--help] [-n|--numthreads <numthreads>] [-d|--duration <duration>]
  [-s|--size <size>] [--workload <workload>] [--isa <isa>] [--strict-isa-check]
  [--threading <threading>] [--large-pages] [--bypass-timer-check] [--topo-debug]
  [--timer-debug] [--guppy]

Display usage information.

OPTIONS, ARGUMENTS:
  -?, -h, --help
  -n, --numthreads <numthreads>
                           Number of threads to use [default: 0]
  -d, --duration <duration>
                           Bench duration (s) between 1-1000s [default: 30]
  -s, --size <size>        Grid size, one of [64, 128, 192, 256, 320, 384, 448, 512,
                                              640, 768, 1024, 2048, 4096, 8192]
                                              [default: 192]
  --workload <workload>    Workload to use
                               (results with different workloads aren't comparable),
                               one of [standing_wave, soft_gaussian]
                               [default: standing_wave]
  --isa <isa>              Instruction set to use for workload/solver,
                               one of [sse2, avx, avx2, fma, fma-apx, avx512f,
                                           avx512vl, avx10-2-256, avx10-2-512]
                                     [default: sse2]
  --strict-isa-check       Don't try to run if the ISA support check fails
  --threading <threading>  Threading manager to use [default: os]
  --large-pages            Use large pages allocator
  --bypass-timer-check     Run the bench even if the timer is not HWBOT compliant
  --topo-debug             Print detected topology for debug
  --timer-debug            Printed detected timers
  --guppy
```

## 2.1 Memory Requirements

FDTD is quite memory intensive as you start to simulate a larger and larger area. To simplify, and since the simulation for the benchmark is essentially arbitrary, the entire simulation space is a large cube, therefore we only need to specify a single side of the cube. As will be explained in more detail below in the FDTD section, our large cube is split up into many smaller cubes, and at each point in the cube we have the vectors for both the electric (E) and magnetic (H) fields. The vectors are composed of 3 double precision floating point numbers, with a fourth used for alignment / padding. The alignment / padding will be explained more in Section 4. The size parameter determines how many mini cells (finite differences) makes up our simulation space, with an extra layer of cells around as the simulation boundary. Therefore, to store our fields, we need for a field of size $N$, 4 double precision floats $\cdot$ 8 bytes per double $\cdot$ 2 fields $\cdot$ $(N+2)^3$ bytes. Since each thread works on its own copy of the data, this memory usage is per thread. Pretty quickly that leads to a lot of memory used as either the problem size increases or more threads are used. The current default is a size of 192 which leads to 445.64 MiB per thread. If large pages are used the memory allocation will be rounded up to a multiple of the large page size, with 2 MiB large pages this shouldn't be a problem but 1 GiB may substantially increase memory usage at the smaller sizes. Since allocation is still done per thread, it isn't a single round up across all threads but each threads allocation. That is at the standard size each thread is then using 1 GiB not 445.64 MiB essentially doubling memory allocation size.

Table 1: Memory Requirements

Memory Scaling with Thread Count

| **Threads** | **Total Mem** | |
|---|---|---|
| 1 | 445.64 | MiB |
| 2 | 891.28 | MiB |
| 3 | 1.33 | GiB |
| 4 | 1.78 | GiB |
| 5 | 2.23 | GiB |
| 6 | 2.67 | GiB |
| 7 | 3.12 | GiB |
| 8 | 3.57 | GiB |
| 9 | 4.01 | GiB |
| 10 | 4.46 | GiB |
| 11 | 4.90 | GiB |
| 12 | 5.35 | GiB |
| 13 | 5.79 | GiB |
| 14 | 6.24 | GiB |
| 15 | 6.68 | GiB |
| 16 | 7.13 | GiB |

Memory Scaling per Thread with Grid Size

| **Grid size** | **Mem/Thread** | |
|---|---|---|
| 64 | 17.55 | MiB |
| 128 | 134.09 | MiB |
| 192 | 445.64 | MiB |
| 256 | 1.02 | GiB |
| 320 | 1.99 | GiB |
| 384 | 3.43 | GiB |
| 448 | 5.43 | GiB |
| 512 | 8.09 | GiB |
| 640 | 15.77 | GiB |
| 768 | 27.21 | GiB |
| 1024 | 64.38 | GiB |
| 2048 | 513.50 | GiB |
| 4096 | 4.01 | TiB |
| 8192 | 32.02 | TiB |

# 3 General Optimization Advice

This section is a brief outline of general advice for optimization `guppy`. While reading the rest of the document is highly recommended, this advice should work well in the vast majority of circumstances. By default the thread count will be the number of actual cores since SMT is unlikely to help unless either your uarch allows many simulataneous loads from memory or your core count is very low compared to your available bandwidth. Use –threading=numa to automatically lock core affinity in an intelligent way, it will do p cores, e cores than hyperthreads spreading across multiple NUMA nodes if they are available. Alternatively, –threading=phte prioritizes p cores, then p core hyperthreads, then e cores, which may be

preferable on systems where you want to maximize single-thread performance before spreading to e cores. If enabled on your operating system –large-pages will also most likely give a small bump in performance, especially with larger sizes though if you have 1GiB large pages enabled it will round the memory allocation in each thread up to 1GB which can increase the memory footprint substantially. Denormalized floating point values are automatically flushed to zero during field initialization to prevent potential performance degradation on x86 platforms. The benchmark will not care much about the more latency focused timings but will substantially care about any timings to do with back to back reads and memory clock. IMC clock (uclock) does not seem to matter, as in run the highest possible memory speed you can with all channels regardless of the memory clock to imc clock ratio (gear) required. The difference between different ISAs is very small on most cpus since the clock completing work faster doesn't matter if its spending most of its time waiting for data to arrive. Generally the sse2, fma, or avx512vl are the fastest though test thoroughly for maximum efficiency.

# 4 HPC, Memory Access, and Caching

## 4.1 Stencil Computations in HPC

In scientific and engineering HPC computing, simulations of physical systems governed by Partial Differential Equations (PDEs) is extremely common. These numerical PDE solvers usually evolve the state in discrete timesteps. FDTD is just one such example of these kinds of solvers. As mentioned in Section 5, as the solver updates the state from time $t$ to $t+1$ we update the simulation state at each point in a 3D grid based on its neighbours. This common data dependency relationship, which is shared in most 3D simulation software, is known to HPC programmers as a stencil, specifically iterative stencil loops. While this is computationally straightforward, it presents a significant challenge for the memory subsystem. Because 3D grids are linearized into 1D memory arrays, neighbors in the $x$ direction are adjacent in memory, but neighbors in the $y$ and $z$ directions are separated by large strides. That is, to access the E field at $i$, $j$, $k$ (integer indexes into the grid), we access the E field array at $(i \cdot L + j) \cdot L + k$ where $L$ is the axis length. This creates a spatial locality (how far apart the data were accessing is) that heavily taxes the CPU's ability to manage data movement. This becomes important as we discuss prefetching and caching.

## 4.2 Address Translation and the TLB

Applications running inside an operating system deal with what are known as virtual memory addresses which abstracts away the complex reality of pages, swapping to/from disk, and other operating system features. This is also an important security feature in modern operating systems to prevent stack smashing, or any other attempts to write into either the operating system's or other program's memory. Every memory address initiated by a thread requires the CPU to translate a virtual memory address (what the progress sees) into a physical hardware address (the actual location in memory). This translation is managed by the page table, which is stored in main memory. To accelerate this process, CPUs utilize the Translation Lookaside Buffer (TLB), a specialized high-speed cache for recent translations. For a standard 4 KiB page size, a large FDTD grid spans hundreds of thousands of pages. As the solver sweeps through the grid, the number of active pages quickly exceeds the capacity of the TLB, leading to a phenomenon known as TLB thrashing. In this state, the CPU spends a disproportionate number of cycles "walking the page tables" in RAM rather than processing simulation data. By utilizing the `-large-pages` option to enable 2 MiB pages, the memory "coverage" of each TLB entry is increased by orders of magnitude, reducing the effect.

## 4.3 Prefetching

Modern processors employ hardware prefetchers to mitigate memory latency by predicting linear access patterns and pulling data from DRAM into the L2 and L3 caches before it is explicitly requested. While prefetching is effective at hiding latency, it cannot increase the physical width or frequency of the memory bus. Therefore, in order to make a memory latency bound benchmark, you have to basically 'defeat' branch prediction and prefetching, however if the volume of data being moved is significantly larger than the cache, even if the prefetcher perfectly anticipates the next set of cells, hiding the latency, the memory controller / cache subsystem eventually reaches a saturation point where it cannot transfer data any faster. This scenario immediately evicts older data from the cache, turning the cache hierarchy into a transient buffer rather than a reuse store. At this limit, the execution units are starved of data, and wider instruction sets such as AVX-512 provide little to no performance gain because the bottleneck is strictly the throughput of getting data from memory through the cache to the cpu and back.

## 4.4 Stencil Computations and the Roofline Model

The performance limits of grid-based numerical solvers are best understood through arithmetic intensity (AI), defined as the ratio of floating-point operations (FLOPs) performed per byte of data transferred from main memory. The Roofline Model provides a visual framework for this relationship, plotting attainable performance against AI to determine whether a kernel is memory-bound" or compute-bound. In the memory-bound regime, the workload has a low AI; the CPU execution units are frequently idle while awaiting data from DRAM. In such scenarios, increasing the core clock frequency or utilizing wider vector instructions (e.g., AVX-512) yields diminishing returns because the performance ceiling is dictated by the memory bandwidth (GiB/s). Conversely, a compute-bound workload has high AI, where data remains in cache long enough to perform many operations, allowing performance to scale linearly with frequency. Stencil-based codes typically fall deep into the memory-bound region.

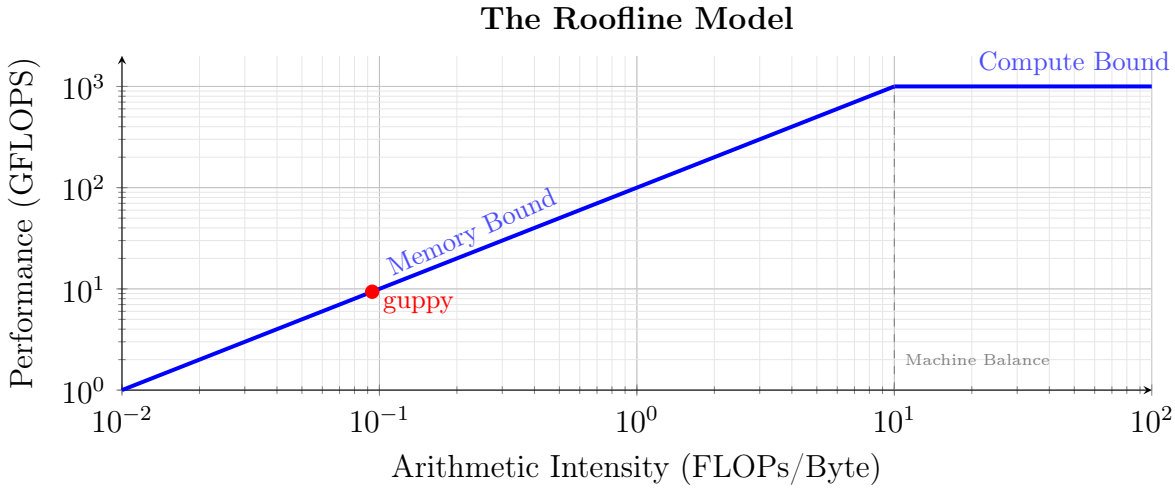


Figure 1: Visual representation of the Roofline Model assuming 100GB/s. Stencil operations typically reside on the steep memory-bound slope, far to the left of the machine balance point.

The calculations for the AI of `guppy` as shown in Fig. 1 will be calculated later in Section 6.2.3 If we know the AI of a specific workload we can then calculate the required FLOPs to saturate a given bandwidth. As shown in 2, the required GFLOPs is actually quite low. Lets examine some outliers AI. A Pentium G4400 is one of the slower dual channel DDR4 cpus, at a base frequency of 3.3 GHz, it lacks avx support which leaves us with a max of 2 FP64 ops per cycle per core with SSE and 2 FP pipelines which gives us $2 \cdot 4 \cdot 3.3 \cdot 10^9 = 26.4$ GFLOPs. The machine balance of the cpu is 51.2/26.4, that gives a extremely low machine balance of just 0.52 FLOP/byte, with most modern cpus being above 5 (see Fig. 3). Even

Table 2: Required Performance for Memory Bandwidth Saturation ($AI \approx 0.09375$)

| Memory Configuration | Rate (MT/s) | BW (GB/s) | Required GFLOPs |
|---|---|---|---|
| Dual Channel DDR2 | 800 | 12.8 | 1.20 |
| Dual Channel DDR4 | 3200 | 51.2 | 4.80 |
| Quad Channel DDR3 | 1600 | 51.2 | 4.80 |
| Dual Channel DDR4 | 4400 | 70.4 | 6.60 |
| Dual Channel DDR5 | 4800 | 76.8 | 7.20 |
| Dual Channel DDR5 | 6000 | 96.0 | 9.00 |
| Quad Channel DDR4 | 3200 | 102.4 | 9.60 |
| Dual Channel DDR5 | 8000 | 128.0 | 12.00 |
| Octa Channel DDR5 | 4800 | 307.2 | 28.80 |

a Pentium G7400 with dual channel ddr5 has a machine balance of about 1.5 FLOP/byte. In fact, the G4400 almost has enough theoretical FLOPs to handle octa channel ddr5. In practice there are a lot more considerations and inefficiencies that leaves this level of analysis rather naive. The key detail is that low AI (significantly less than 1) leaves a workload extremely memory bandwidth bound.

Table 3: Machine Balance Comparison for Various Architectures (FP64 sse/avx/avx2)

| Architecture / Model | Memory Config | BW (GB/s) | GFLOPS | Machine Balance |
|---|---|---|---|---|
| Pentium G4400 (2C) | 2-Ch DDR4-3200 | 51.2 | 26.4 | 0.52 |
| Pentium G7400 (2C) | 2-Ch DDR5-6400 | 102.4 | 76.8 | 0.75 |
| Pentium 4 (Northwood) | 2-Ch DDR-400 | 6.4 | 6.4 | 1.00 |
| EPYC 7262 (8C) | 8-Ch DDR4-3200 | 204.8 | 230.4 | 1.13 |
| Core 2 Quad Q6600 | 2-Ch DDR2-1066 | 17.1 | 38.4 | 2.25 |
| i7-8700K (6C) | 2-Ch DDR4-3200 | 51.2 | 259.2 | 5.06 |
| i9-10980XE (18C) | 4-Ch DDR4-3200 | 102.4 | 1036.8 | 10.13 |
| Ryzen 9 5950X (16C) | 2-Ch DDR4-3600 | 57.6 | 588.8 | 10.23 |
| EPYC 9654 (96C) | 12-Ch DDR5-4800 | 460.8 | 4608.0 | 10.00 |
| i9-14900K (24C) | 2-Ch DDR5-8000 | 128.0 | 1088.0 | 8.50 |
| RTX 4090 | GDDR6X (384-bit) | 1008.0 | 1288.0 | 1.28 |
| H100 (SXM5) | HBM3 (5120-bit) | 3350.0 | 33 500.0 | 10.00 |

## 4.5 Superscalar Execution, Data Dependencies, and SMT

Modern CPUs achieve high performance not only through high clock frequencies, but through deep instruction pipelines, superscalar issue width, out-of-order execution, and simultaneous multithreading (SMT). While these architectural techniques dramatically improve performance for compute-bound workloads, their impact on low arithmetic-intensity stencil codes which are fundamentally limited by memory throughput is essentially negligible.

### 4.5.1 Superscalar Pipelines and Out-of-Order Execution

Contemporary processors are superscalar, meaning they can issue multiple micro-operations per cycle to different execution units (e.g., integer ALUs, floating-point units, load/store units). Combined with deep pipelines and out-of-order scheduling, this allows the processor to overlap independent instructions and hide instruction latency. In compute-bound kernels with high arithmetic intensity, these mechanisms

enable near-linear scaling with clock frequency and vector width. However, in memory-bound workloads, execution units frequently stall waiting for data to arrive from the cache hierarchy or main memory. When load buffers and miss status holding registers (MSHRs) are saturated, additional execution width provides little benefit because the bottleneck is no longer instruction throughput but data movement.

#### 4.5.2 Data Dependencies and Memory-Level Parallelism

Stencil computations exhibit structured spatial data dependencies. As mentioned above each simulation update depends on neighboring field components. Within a single timestep, these dependencies are local and deterministic, allowing spatial parallelism across different regions of the grid. Modern CPUs attempt to exploit memory-level parallelism by issuing multiple outstanding cache misses simultaneously. The effectiveness of this mechanism depends on the availability of independent memory operations. However, the maximum number of outstanding memory transactions is finite. Once this hardware limit is reached, additional instructions cannot increase concurrency, and cores stall waiting for memory responses. At this point, performance becomes strictly bandwidth-bound rather than latency-bound. Increasing core frequency may reduce arithmetic latency, but it does not increase the number of bytes transferred per second from DRAM.

#### 4.5.3 Simultaneous Multithreading (SMT)

Simultaneous multithreading (e.g., Intel Hyper-Threading or AMD SMT) allows multiple logical threads to share a single physical core. SMT can improve utilization when execution units would otherwise be idle due to branch mispredictions or instruction dependencies. In memory-bound workloads, SMT has a more nuanced effect. On lightly loaded systems, an additional logical thread may increase memory-level parallelism and improve bandwidth utilization by filling otherwise idle load/store slots. However, once memory bandwidth is near saturation, SMT threads compete for shared core resources such as load/store queues, L1/L2 cache capacity, TLB entries, and reorder buffer space.

# 5 Finite Difference Time Domain

The Finite Difference Time Domain (FDTD) method, solves the partial differential equations for electromagnetics, Maxwell's equations (Eq. 1, with a small discrete grid (the Finite Difference) over small discrete timesteps (the Finite Time). While proposed back in the 60s it is quite memory intensive and not as applicable to lower frequency simulations as other techniques, therefore it did not see widespread adoption until later.

## 5.1 Maxwell's Equations in Differential Form

The FDTD method is a direct solution to Maxwell's two time-dependent curl equations. In a linear, isotropic, and non-dispersive medium (like the vacuum or simple dielectric), these are defined as:

$$\begin{aligned} \nabla \times \mathbf{E} &= -\mu \frac{\partial \mathbf{H}}{\partial t} \\ \nabla \times \mathbf{H} &= \epsilon \frac{\partial \mathbf{E}}{\partial t} + \mathbf{J} \end{aligned} \tag{1}$$

Where:

- $\mathbf{E}$ is the Electric field vector (V/m)
- $\mathbf{H}$ is the Magnetic field vector (A/m)
- $\epsilon$ is the permittivity of the medium (F/m)

- $\mu$ is the permeability of the medium (H/m)
- **J** is the electric current density

### 5.1.1 Vector Expansion

To implement these on a CPU, the curl operators ($\nabla\times$) are expanded into six coupled partial differential equations in Cartesian coordinates $(x, y, z)$. For example, the change in the $x$-component of the Electric field depends on the spatial derivatives of the $y$ and $z$ components of the Magnetic field:

$$\frac{\partial E_x}{\partial t} = \frac{1}{\epsilon}(\frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z}) \tag{2}$$

The FDTD method replaces these continuous derivatives ($\partial$) with central-difference approximations ($\Delta$) over the Yee grid (explained in Section 5.2).

## 5.2 The Yee Grid

The core of FDTD is the **Yee Cell**. Instead of calculating all field components at the same points in space, the electric field (**E**) and magnetic field (**H**) components are staggered.

- **E** components are situated on the edges of the cell.
- **H** components are situated on the faces of the cell.

To update $\mathbf{H}_{ijk}$ we need the $\mathbf{E}_{ijk}$, and its next neighbours: $\mathbf{E}_{i+1,j,k}$, $\mathbf{E}_{i,j+1,k}$, $\mathbf{E}_{i,j,k+1}$. Conversely, to update $\mathbf{E}_{ijk}$ we need $\mathbf{H}_{ijk}$ and its previous neighbours: $\mathbf{H}_{i-1,j,k}$, $\mathbf{H}_{i,j-1,k}$, $\mathbf{H}_{i,j,k-1}$. The outer boundary of $i, j, k = 0$ or $i, j, k = L$ is not updated to create a perfectly reflecting boundary.

### 5.2.1 Timestep

Each FDTD timestep first updates the H field based on the E field, then updates the E field based on the H field. Therefore, while you can think of the whole simulation taking place in integer timesteps, each field is actually 1/2 a timestep apart.

### 5.2.2 Numerical Stability (CFL Condition)

For the simulation to remain stable and not diverge to infinity, the timestep $\Delta t$ must be bounded by the grid spacing $\delta$. This is known as the Courant-Friedrichs-Lewy (CFL) condition. In 3D:

$$\Delta t \leq \frac{\delta}{c\sqrt{3}} \tag{3}$$

where $c$ is the speed of light in the medium.

# 6 guppy

The core design philosophy of `guppy` is centered on five primary goals:

1. Bandwidth saturation
2. Support a broad range of hardware without requiring different work sizes
3. Low computational overhead
4. Minimizing score scaling purely from CPU frequency
5. Maintaining even CPU and memory load throughout the entire duration of the run

## 6.1 The Scaling Challenge

A persistent challenge for benchmarks on HWBOT is the massive performance delta between legacy hardware (e.g., Pentium 4) and modern high-core-count enterprise silicon. In benchmarks like SuperPi 32M, we have seen runtimes shrink from over an hour on a Willamette-based Pentium 4 to under three minutes on a Core i9-14900KF. When benchmark durations become excessively short, the nature of the test changes. The level of system stability required to complete the run decreases significantly, and the benchmark can turn into a "slot-machine" where benchers simply re-run the test repeatedly until a lucky score passes. This issue is compounded in multicore scenarios where the gap in theoretical FLOPs and memory bandwidth between consumer and enterprise platforms is significantly larger.

### 6.1.1 Mitigation Strategies

`guppy` addresses the scaling problem through two key architectural choices:

- Running for a minimum duration (defaulting to 30 seconds)
- Executing an isolated copy of the simulation in each thread, with the final score representing the aggregate throughput.

The choice of a 30-second window provides a balance between the required stability to run the test and quick iteration while testing. Furthermore, this actually increases the load on faster cpus, since each iteration is faster.

### 6.1.2 Throughput and Multithreading

The multithreading strategy is specifically designed to force a memory bandwidth bottleneck. By spawning an independent simulation per thread, the total memory footprint scales linearly with the thread count. In a single-threaded scenario, a core has exclusive access to the system's entire L3 cache and a disproportionate share of the memory bus. In this state, the workload is often compute-bound (limited by CPU FLOPs). However, as threads are added, the aggregate computational potential of the CPU quickly outpaces the memory controller's ability to feed it. By saturating all cache levels and flooding the memory controller, `guppy` shifts from a test of the feeding a single-core and per-core FLOPs to a test of total system throughput. This model accurately reflects high-performance computing (HPC) environments where clusters run multiple simultaneous simulations (parameter sweeps) in parallel. This approach also ensures that the step time remains a function of single-core efficiency rather than total core count, allowing the benchmark to scale to 100+ cores without requiring an unmanageable increase in grid size. The multithreading strategy also ensures good NUMA scaling since the memory for each simulation doesn't have to be access across multiple NUMA nodes. The main thread acts as worker thread 0, participating in the benchmark alongside the spawned threads. In a single-threaded run, no additional threads are spawned and the main thread performs all the work directly.

## 6.2 Solver implementation

The core loop of `guppy` iterates for 30s by default or whatever duration in seconds is specified. Each iteration represents one physical timestep ($\Delta t$). Because the goal is to measure maximum throughput, the solver is stripped of complex features like different materials, complex boundaries, etc..., focusing purely on the raw 3D stencil update across the grid in a tight loop. Because this benchmark isn't trying to perform actual simulations, normalized constants are used, that is $c = 1$, $\mu = 1$, $\epsilon = 1$. Each yee cell's size, $\delta$, is given a size of $10 \cdot 10^{-3}$, from which the $\Delta t$ is calculated with Eq. 4 to satisfy the CFL stability criterion as specified in Eq. 3:

$$\Delta t = \frac{\delta}{2c} = \frac{10 \cdot 10^{-3}}{2(1)} = 5 \cdot 10^{-}3 \tag{4}$$

As usual for a FDTD implementation we update the yee cells first for the H field then the E field. Since were using a staggered field, and the H field update looks to the next cells in each direction we iterate with $i$, $j$, $k$ from 1 to $L2$. While for the E field update we look at the previous cells, therefore iterate from 2 to $N-1$. The inside of each loops looks essential identical, therefore we will only example the H field update. The loop can be slip into 3 parts: loading from memory, shuffle/swap, and FLOPs.

### 6.2.1 Loading from memory

To update one cell's H field, the CPU must load the current E field component ($E_{i,j,k}$) and its three spatial neighbors ($E_{i+1,j,k}$, $E_{i,j+1,k}$, $E_{i,j,k+1}$). The field at each point is represented by a vector of 3 double-precision floats (doubles) + one extra to pad out to 32 bytes. The 32 byte padding is important for vectorization since it allows for aligned instead of unaligned loads. Since the solver is designed to be as memory bandwidth limited as possible, requiring to load more data, via padding and double precision is an intentional choice. In total, we have 5 loads of 4 doubles which leads to $5 \cdot 4 * \cdot 8 = 128 bytes$.

### 6.2.2 Shuffle/Swap and FLOPs

$$\mathbf{H}_{i,j,k} - = \frac{\Delta t}{\mu_0 \Delta x} [ \begin{pmatrix} E_{z,i,j+1,k} \\ E_{x,i,j,k+1} \\ E_{y,i+1,j,k} \end{pmatrix} - \begin{pmatrix} E_{z,i,j,k} \\ E_{x,i,j,k} \\ E_{y,i,j,k} \end{pmatrix} - \begin{pmatrix} E_{y,i,j,k+1} \\ E_{z,i+1,j,k} \\ E_{x,i,j+1,k} \end{pmatrix} + \begin{pmatrix} E_{y,i,j,k} \\ E_{z,i,j,k} \\ E_{x,i,j,k} \end{pmatrix} ] \tag{5}$$

To calculate the curl and update the H field, there is only one vector multiply, by a scalar, three vector subtractions and one vector addition. As descrete floating point values, without vectorization, this becomes 15 operations since we don't need to operate on the padding, just $x$, $y$, $z$. In practice, on modern cpus these 5 FLOPs are essentially negligible, especially since while there is a data dependency chain through thme, there is some level of instruction level parallelism that can be utilized. Furthermore, since it is far slower to load from memory into registers and re-arrange the data than to actually do the floating point math, the actual advantage of FMA/AVX2 add/subtract/multiply/fma is limited compared to support for advanced shuffle operations such as with avx512vl. Though overall, this still is significantly less important than the time taken to load into registers from memory, even with the prefetcher. Since the memory acces pattern is not particuilarily difficult to follow it is assumed that the prefetcher is able to load values into cache and the throughput simply has to be high enough that it is overloaded.

### 6.2.3 Arithmetic Intensity

Using these values, we can derive the maximum arithmetic intensity as defined in Section 4.4:

$$AI = \frac{\text{Total FLOPs}}{\text{Total Bytes Transferred}} \approx \frac{15 \text{ FLOPs}}{160 \text{ Bytes}} = 0.09375 \text{ FLOPs/Byte} \tag{6}$$

Referring back to Table 3 and Table 2, this is a very low AI. As such, the solver will remain extremely memory bandwidth bottlenecked except in very unusal circumstances. In practice since the non strided memory access is likely to be cached without transiting the memory bus, and more instructions than just the FLOPs being required, the effective AI will be higher, perhaps closer to 0.25. Even then, outside of some more extreme examples, the solver should remain bottlenecked. In testing with going from 5ghz to 5.7ghz on a 13900ks (8p), a 14% increase in per thread performance, having a measured 0.17% score increase averaged across 3 runs. Adding more threads onto different cores however does increase performance, since it enables more load/stores to better utilize available memory bandwidth. However, once memory bandwidth is already very well utilized with many threads, more threads may decrease performance as the memory controller, especially prefetcher and TLB has to work harder to keep all cores fed with data. (Data in Tab. 4 in Appendix A)

## 6.3 Workloads

Currently `guppy` supports selectable workloads. The workloads define the how the fields are initalized, as well as the pre and post steps to provide a source of signals and verify that the simulation is physically plausible. Currently, there are two possible workloads, `standing_wave` and `soft_gaussian`.

### 6.3.1 standing_wave

The `standing_wave` workload is designed to provide a spatially uniform and constant computational load across the entire grid. The grid is initialized with a sinusoidal distribution of electric fields in the $x$ and $y$ dimensions with a fixed period of 32 cells. The initialization ensures that the energy density is distributed throughout the simulation space. Every loop the difference in energy is checked against the value after initialization at the nodes and antinodes, while after the workload is finished the total energy is compared to initalization. Since the per iteration check is only at the nodes/antinodes this method somewhat relies upon bitflips disrupting the standing wave and spreading to the nearest node/antinode over subsequent iterations. With larger grids, a small disruption in total energy is much harder to catch since it becomes small realtive to total energy.

### 6.3.2 soft_gaussian

The `soft_gaussian` workload is more realistic to an actual FDTD simulation with a soft gaussian pulse over the first 10s of iterations that then bounces off the boundaries and self interferes. However, this means that it doesn't scale to larger grid sizes as until the wave propagates through the environment most of the grid will be zero. After every iteration the total energy in the grid is compared against the energy injected into the grid, known values, and defined thresholds. The total grid energy calculation is quite cpu and time intensive which reduces time in the solver.

## 6.4 Verification in Detail

Since `guppy` is designed to be a competitive benchmark it will frequently be run on unstable systems. However, complex verification is in conflict with the design goals since that would add a heavy cpu load. Energy of the total grid and a specific point in the grid is based off the Hamiltonian energy.

$$H = \frac{1}{2} \int_V (\epsilon |\mathbf{E}|^2 + \mu |\mathbf{H}|^2) dV \tag{7}$$

Given that we are operating with a static discrete difference, the Hamiltonian for a single point becomes.

$$H = \frac{\epsilon |\mathbf{E}|^2 + \mu |\mathbf{H}|^2}{2} \cdot \delta^3 \tag{8}$$

However, its important to note that the E and H fields are half a timestep apart and the E and H fields are staggered. While we can't do anything about the temporal difference, we can tackle the spatial offset by averaging the H and the current point with its previous neighbours. Checking for energy conservation in the system is a good way to check for any egregious errors in the simulation that caused significant divergence. Due to imperfections in the simulation, the energy will usually converge and slightly oscillate around some scalar multiple of the energy put into the system.

Because of the realities of floating point math small changes in the mantissa need to not produce an error, this also means that bit flips in the mantissa that only slightly affect the value will go unnoticed, especially if there are not many iterations left. However, the primary goal of verification is to catch large errors that make the simulation physically implausible. The number of iterations run in each of the solver loops is also verified instability causing less work being completed than intended. Each workload step reports back a specific error message instead of just a pass/fail so that the exact reason for a failure can be shown. If any

of the verification fails before 30 seconds has elapsed, the failing thread prints that error, raises a flag with all threads, and stops doing work immediately; the other threads stop after they get through their current iteration, and the process exits with a failure return code. Therefore, especially with very slow iterations it may take a while for all threads to stop by themselves. However, this is a tradeoff with adding more checks either in the loop (adding more software overhead) or more aggressively terminating the program instantly (potentially leading to dangling resources).

## 6.5 Scoring: MGups

The primary metric for `guppy` is Mega-Grid-Updates per second (MGups). A "grid update" is defined as one update of one grid point's field component. Since each iteration (timestep) sweeps both the electric ($E$) grid and the magnetic ($H$) grid, which are staggered half a timestep apart, one iteration performs $2N^3$ grid updates. The score is calculated as:

$$\text{Score} = \sum \frac{2N^3 \times I}{10^6 \times \text{seconds}} \tag{9}$$

Where $N$ is the grid size and I is the total number of iterations (timesteps) completed. The score will always be displayed up to at least 4 significant figures or up to 2 decimal places whichever is greater. During the benchmark run, a real-time progress display shows the current elapsed time, MGups, and iteration count.

By totaling the score across all threads the end result represents the total possible throughput of all cores, which ensures that the peak score will be reached when memory bandwidth is maximally utilized. Only the time taken in the solver is used for the score calculation, pre and post steps are part of the overhead time. This is especially important for the soft_guassian workload since the computation limitation of calculating the sum of hamiltonian at every point to verify the total energy pushes it to being a compute bound benchmark if the workload post step was included in the benchmark time. In effect, this also averages across every iteration run which should reduce the impact from any small bits of bandwidth fluctuation or operating system overhead.

## 6.6 Timers

`guppy` scores based on the total throughput, therefore we need to know not just how much work was done by each thread but exactly long it spend doing the work. Since the 30 second duration is the minimum duration, not the exact duration, final scoring depends on the exact amount of time each thread took to do their work. However, all timers have flaws. High frequency timers are generally affected by runtime changes to a cpu's base clock. Other timers are far slower to access to not available to usermode applications. To try to detect timer skew or any other way that the amount of time measured is incorrect or the bench did not run for its proper duration, we use two separate timers that ideally have separate sources:

The **fast timer** is the primary clock for the bench duration, and the timer that is used for scoring. On Windows it is QueryPerformanceCounter (QPC), on Linux it is `clock_gettime` with `CLOCK_MONOTONIC_RAW`, and on any other platform `std::chrono::high_resolution_clock` is used as a fallback.
`CLOCK_MONOTONIC_RAW` is used deliberately in preference to `CLOCK_MONOTONIC` because the raw variant is not affected by NTP slew or `adjtime` adjustments, which could otherwise shift the measured work time by small amounts that do not correspond to work actually performed. Elapsed work time is always taken from the fast timer, including the loop condition that decides whether the benchmark duration has elapsed, so the run length and the score are derived from the same clock and can never disagree with each other.

The **coarse timer** is a second, independent, low resolution clock that is used purely as a cross-check. On Windows it is `GetTickCount64`, which ticks once per millisecond, and on Linux it is `clock_gettime` with `CLOCK_MONOTONIC_COARSE`, which has a substantially lower resolution than the raw clock. Each worker records its total work time on both timers, and when the results are gathered the two totals are compared.

Because the coarse timer only resolves whole milliseconds, the fast total only has to agree with the coarse total to within a 32 millisecond tolerance, which allows for at most a couple of timer ticks of slew on windows. If the two disagree by more than that, the result is marked invalid, the fast timer still reports what it measured, and the run exits with a failure rather than a score. Requiring two independent sources to agree makes it substantially harder for a run to report a work time that nothing else corroborates, which matters when the benchmark is expected to be run on systems that may be unstable.

Both timer frequencies, along with the detected source of the fast timer, are printed when the benchmark starts. On Windows there is no way to directly ask what QPC is implemented by, so the source is inferred from the QPC frequency and cross checked against the TSC information reported by CPUID, and is reported as TSC or iTSC (an invariant TSC), HPET, ACPI (the ACPI power management timer), PIT, or RTC. On Linux the source is read from the current clocksource in sysfs. This is reported mostly for debug value, since the speed of the fast timer has essentially no effect on the score: a slow but still accurate enough clock will measure the same work as a fast one, and only timers accurate to within an iteration matter. Unfortunately, RTC and TSC can both by skewed together on some platforms which means they cannot be used to cross check one another, HPET is resilient to skew but not available in usermode. Therefore, for platforms that do exibit skew for RTC/TSC, HPET must be used as QPC to ensure accurate timekeeping. Timer calls are minimized to try to mitigate the higher overhead of using HPET as QPC, it is still very noticeable unfortunately...

To enforce the HWBOT timer rules, `guppy` checks the fast timer's source against the hwbot rules as of October 15th, 2026; before starting a run. On Windows 8 and newer, processors that are not recent Intel (Skylake or later, excluding Broadwell-E) are required to use HPET as their fast timer source; otherwise the run is refused by default and exits with an error. This includes AMD processors, as well as older Intel parts. This check is only performed on Windows; on non-Windows platforms the timer is considered compliant regardless of source. This check can be overridden with the `-bypass-timer-check` command line option, which allows the bench to run even when the fast timer is not HWBOT compliant, printing a warning banner that asks the user to check that the detection is erroneous in terms of windows version, processor, or timer before submitting the result. This option is planned to be removed in future versions if its determined that the check doesn't produce false positives, or benchmate becomes the only way to run `guppy`. The option `-timer-debug` prints debug info related to timer detection.

Since both timers read 64 bit tick counters, a sufficiently long running process or windows sitting idle for a long time can eventually wrap the counter, especially if its sufficiently high frequency. Before any work starts, both timers are checked to make sure that neither counter will overflow within the requested duration, plus two extra seconds of headroom so the in-flight iteration can finish, and if either would overflow the run aborts with an error instead of measuring across a wrap.

Once a full iteration has been complete, that is: pre-step, step, and post-step; as long as the fast timer has not measured the requested duration, another iteration will be started and run until it is complete. It is entirely expected that if the first iteration takes 29.5s of measured work time, another iteration will be started and the total time will become about 59 seconds instead of the intended 30 seconds. However, this is mostly an issue for extremely slow hardware and/or very large simulation sizes, where a very small number of iterations will occur within 30 seconds. For most users, the most they may ever see is 31 or 32 seconds instead of 30 seconds. Since actual time in the workload is tracked on the fast timer, there should be no effect on the actual end score because of it, unless the reason it occured was due to the operating system pre-empting the thread, in which case the score will drop dramatically, since fewer iterations will be completed in a larger amount of time. Since score bad is unfortunate but not an issue that can end up with a higher score and this can be easily fixed by increasing process priority, which is expected for a competitive benchmark, this is not considered a bug.

## 6.7 Topology Detection

`guppy` automatically detects the cpu topology: NUMA nodes, cores, hyperthreads, and L3/LLC grouping to better determine the default number of threads to spawn, and to automatically assign threads to cores (on linux and windows only) when `-threading=numa` or `-threading=phte` is specified. The default thread count (shown as `auto` in the help output) will be the number of physical cores detected, not the number of logical cores, further limited by however many threads can actually be allocated within the available system memory for the chosen grid size. If the user overrides the default thread count with -n and `-threading=numa` is set, core affinity will be assigned as such: p cores first spread across numa nodes, then e cores, then hyperthreads. While not always the most efficient, this should be the most efficient on average. If `-threading=os` is passed (the default) then affinity won't be set by `guppy` and the user can use other software to set core affinity in a more advantageous way. The other threading option, `-threading=phte` first prioritizes p cores then p core hyperthreads then e cores, in this case increasing the thread count to match the number of total logical threads is recommended. To see the detected topology pass `-topo-debug`.

## 6.8 Large Pages

When –large-pages is specified on linux and windows, the benchmark will attempt to allocate with large pages (huge pages in linux). As such the allocation size for each thread has to be rounded up to the nearest multiple of the large page size. For 2 MiB large pages this isn't a big issue since that's relatively small compared to the allocation, but with 1 GiB pages that would be an effective doubling of the allocation for the standard 192 grid size. Since that increase is per thread that represents a doubling of overall memory allocation, for 8 threads that takes it from 3.57 GiB to 8 GiB. In general, 1 GiB pages would most likely be overkill for the standard 192 grid size, with 1 GiB pages making sense only for much larger grids.

## 6.9 Thread Model

The main thread acts as worker thread 0, participating in the benchmark alongside any of the other spawned threads (if thread count is greater than 1). All printing is done in the main thread except for any detected errors garded by a mutex. The main thread's participation as a worker ensures it is subject to the same scheduling and resource constraints as the other threads, providing more consistent results. Each worker publishes its iteration count and running MGups through its own cache-line aligned slot in the shared intermediate result array, so no two workers ever write to the same cache line, and those slots are only ever accessed with relaxed memory ordering since they carry no synchronization semantics. False sharing on the progress slots could introduce and element of core to core latency in the benchmark that is undesireable.

## 6.10 Expected Scaling Characteristics

`guppy` is expected to scale mostly from memory access bandwidth. That is limited by memory bandwidth as long as its not limited in bandwidth from the cpu interconnects. For platforms where there is a bottleneck in bandwidth from the memory controller to the cores, that instead can become the bottleneck. If there is no interconnect bottleneck, then memory bandwidth is the biggest bottleneck. Scaling with cpu frequency should be essentially nil in most circumstances while scaling by adding more threads while there are more cores should see decent scaling while bandwidth is not very saturated. This is not because of increased floating point throughput but because of more load/store units becoming available. As more threads are added the per thread score will decrease but the overall score should increase until memory bandwidth or cpu interconnects becomes very saturated. This also means that the ideal thread count may differ depending on memory bandwidth or interconnect bandwidth. In testing so far, scaling has been observed going from gear 1 to gear 2 with ddr4 even when increasing frequency only a single memory ratio in the process. When not interconnect limited, with the same timings, score scaling with memory frequency is

essentially linear. In general, when interconnects are not the limit, peak throughput is expected to be with peak memory bandwidth, which implies all channels at the highest possible memory frequency.

## 6.11 Compiling

`guppy` makes use of new c++20 features to make the source code both much easier to read and to allow for quick development. However, this requires a rather new version of msvc to be used to compile on Windows, which is in contention with another goal: Windows 7 support. For that reason, a msvc version that supports <barrier>, <format>, and <stop_token> must be used and it must also support Windows 7 as a target. For that reason, msvc toolchain 19.44 is the best compromise between c++20 features and Windows 7 Support, however it lacks support for newer ISAs such as AVX10.2 and APX. In order to allow for the entire compatibility matrix, we need to compile three executables in three different modes, called isa tiers:

- **legacy**: 32-bit x86 build with SSE2 only. Using msvc 19.44 for Windows 7 32 bit compatibility. The `-isa` option is not available.
- **compatibility**: 64-bit build with SSE2, AVX, AVX2, and FMA. Using msvc 19.44 for Windows 7 64 bit compatibility. Allows isa selection of sse2 to fma.
- **modern**: 64-bit build with all ISAs including AVX10.2 and APX. Requires vs2026 for AVX10.2 support but gives up Windows 7 compatibility.

The msvc runtime is linked statically, using meson's built-in `b_vscrt=mt` option, so that the produced executables carry no dependency on the visual c++ redistributable and can run on a clean windows installation. Additionally, _DISABLE_CONSTEXPR_MUTEX_CONSTRUCTOR is defined to prevent ABI compatibility issues with older vs2022 redistributables, which only show up after the benchmark has run, often as a crash with no error, seemingly just not printing the result after the bench finishes.

Two meson options are defined: `imgdump` and `isa_tier`. If enabled, `imgdump` will add libpng as a dependency and add an additional cmdline option to dump the x/y cross section of the energy as a png every 10 iterations. These images will be saved as "dump/<workload_name>/<size>/<iteration>.png". The `isa_tier` option controls which ISA paths are compiled using templates and compile time polymorphism of the solver/workloads to create selectable ISA paths with minimal runtime virtual calls; in the legacy tier the `-isa` option will not be present. While there is some amount sketched out for aarch64 support, in general this is only currently supported for x86_64 (and x86 for the legacy tier). For all other situations, or if you want to compile a special version optimized for a single specific platform, select the appropriate tier for the toolchain of your choice and manually define compiler flags as necessary. There are multiple defines that can be used to use the different handwritten intrinsic paths for fma, avx512f, avx512vl. Dependencies are managed with meson wrap.

## 6.12 Test Harness

Included for download on github/discord but not intended to be bundled with benchmark, are extra executables ending in '-harness' that present a convenient test harness for sweeping variables and averaging multiple runs. By default it will run the benchmark three times and give you the average result along with the max/min to give an idea of spread. The test harness also has an additional option `-harness-var`, which will sweep a variable, either threads, size, duration, or isa across defined values for ease of testing. The number of runs at each sweep point is set with `-harness-sub-runs` and defaults to 3, and where the selected variable supports it the sweep scope can be further narrowed with `-harness-opt`. Normally the test harness abides any of the normal flags passed, though while sweeping a variable it will override whatever is passed in. After the sweep, a result table will be printed with each variable, the 3 runs with

the value, the average, and the max/min. Note: if sweeping isa and `-strict-isa-check` is not passed the exe will crash if any illegal instructions are run not producing the final result.

# A Testing Data

Table 4: Performance Scaling of i9-13900KS (8p) with core frequency; 8000 38-47

| Core Freq | Ring/L3 | Run 1 | Run 2 | Run 3 | Average | Gain |
|---|---|---|---|---|---|---|
| 5000 MHz | 5000 MHz | 933.26 | 935.29 | 935.56 | 934.70 | — |
| 5700 MHz | 5000 MHz | 935.93 | 935.85 | 937.04 | 936.27 | +0.17% |

Table 5: Performance Scaling of i9-13900KS (8p) with memory timings; 5.7 core / 5.0 ring; ddr5-8000

| timings | Run 1 | Run 2 | Run 3 | Average | Gain |
|---|---|---|---|---|---|
| 38-49-loose subtimings | 935.93 | 935.85 | 937.04 | 936.27 | — |
| 34-45-tight subtimings | 1183.8 | 1185.3 | 1184.71 | 1184.60 | +26.5% |

Table 6: Performance Scaling of i9-13900KS (8p) with memory frequency; 5.7 core / 5.0 ring; 34-45-tight subtimings

| Memory Frequency | Run 1 | Run 2 | Run 3 | Average | Gain |
|---|---|---|---|---|---|
| 4000 | 1183.8 | 1185.3 | 1184.71 | 1184.60 | — |
| 4200 | 1252.91 | 1242.3 | 1251.36 | 1248.86 | +5.42% |