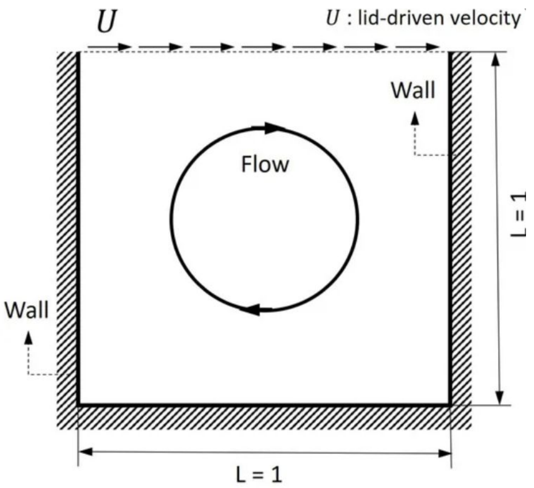
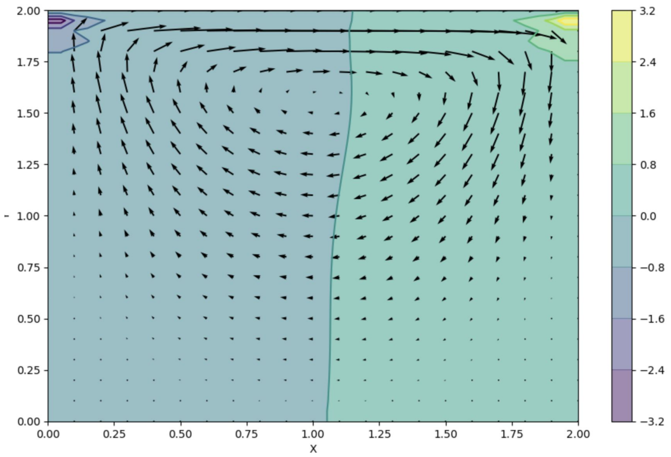
Optimizing Cavity Flow with Navier-Stokes

Team 22: Filip Jaksic, Pavel Kliska, Peter Horcic, Selin Barash

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cavity flow with Navier-Stokes





Input:
nt := the number of simulation steps, nt ∈ N
nx, ny := the sizes of the grid which will be simulated, nx, ny ∈ N

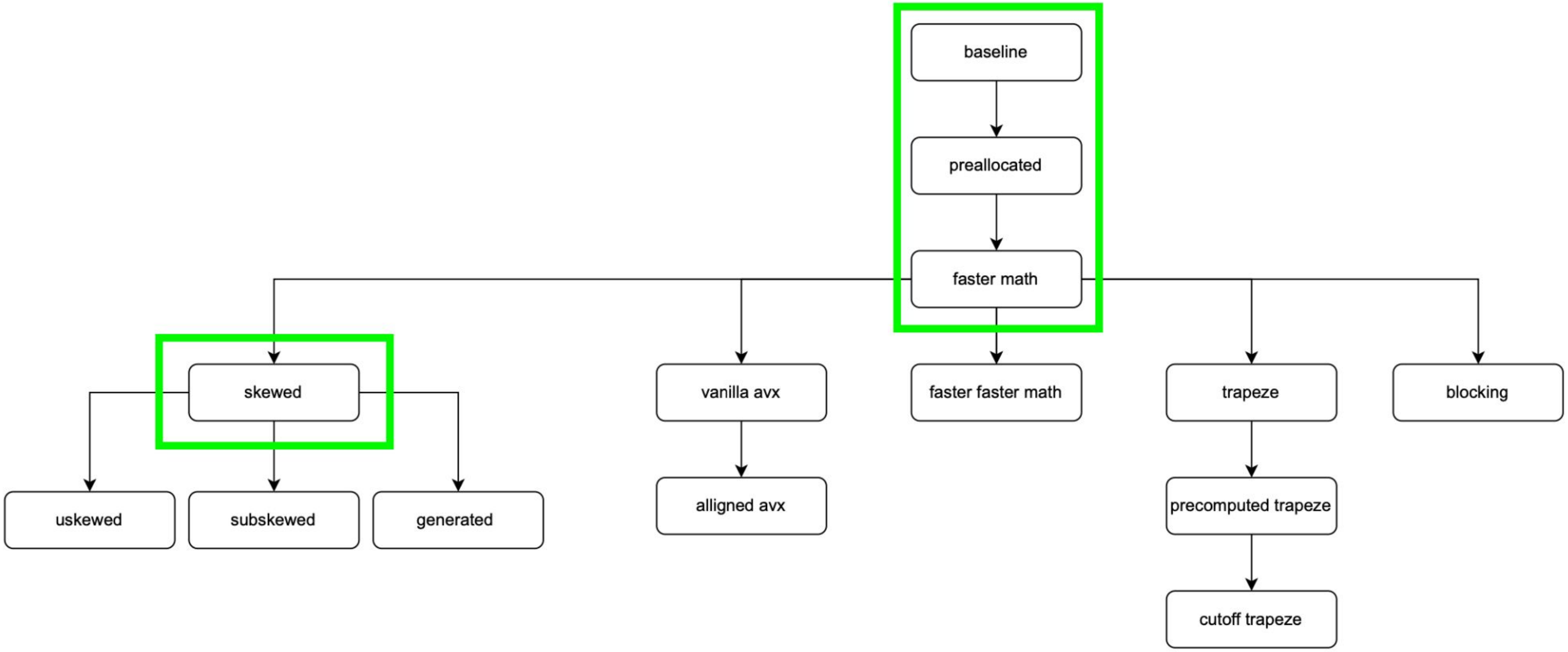
Output:
u := velocity along the x-axis, u ∈ R^(nx*ny)
v := the velocity along the y-axis, v ∈ R^(nx*ny)
p := the pressure of the simulation, p ∈ R^(nx*ny)

Figure on the left from Huang and Lim "Simulation of Lid-Driven Cavity Flow with Internal Circular Obstacles" (2020).
Figure on the right from L. A. Barba, G. F. Forsyth "12 steps to Navier–Stokes" (2017)

Baseline Implementation

- Translate Python code to C code
- We keep time steps nt = 100 constant
- Consider only square matrices
- Runtime is squared wrt the dimension of the grid
- Avoid overflow by choosing other parameters of the simulation appropriately
- Correctness check in the testing infrastructure

Overview of all Optimizations



Standard C Optimizations I

```c
static void pressure_poisson(struct baseline_simulation* sim,
            unsigned int pit, double* b){
    const size_t d = sim->d;
    const double ds = sim->size / (d - 1);
    // const double ds = 0.025;
    const size_t bytes = d*d*sizeof(double);
    double* p = sim->p;
    double* pn = malloc(bytes);

    for(unsigned int q=0; q < pit; q++){
        memcpy(pn, p, bytes);

        for(size_t i=1; i < d-1; i++){
            for(size_t j=1; j < d-1; j++){
                const double pn_left =  pn[d*i     + j-1];
                const double pn_right = pn[d*i     + j+1];
                const double pn_below = pn[d*(i+1) + j  ];
                const double pn_above = pn[d*(i-1) + j  ];

                p[i*d+j] = (
                    ((pn_right + pn_left) * sq(ds)
                            +(pn_below + pn_above) * sq(ds)) /
                        (2 * (sq(ds) + sq(ds))) -
                        sq(ds) * sq(ds) / (2 * (sq(ds) + sq(ds))) *
                        b[i*d+j]
                );
            }
        } // FLOPS: 14 mul, 2 div, 5 add, 1 sub   (d-2)*(d-2)*pit
```

Baseline Poisson

```c
static void pressure_poisson(struct preallocated_simulation* sim,
            unsigned int pit){
    double *restrict b = sim->b;
    const size_t d = sim->d;
    const double ds = sim->size / (d - 1);
    // const double ds = 0.025;

    // 2 flops
    for(unsigned int q=0;q<pit;q++){
        //Swap p and pn
        double *tmp = sim->p;
        sim->p = sim->pn;
        sim->pn = tmp;

        double *restrict p = sim->p;
        double *restrict pn = sim->pn;
        for(size_t i=1;i<d-1;i++){
            for(size_t j=1;j<d-1;j++){
            const double pn_left =  pn[d*i     + j-1];
            const double pn_right = pn[d*i     + j+1];
            const double pn_below = pn[d*(i+1) + j  ];
            const double pn_above = pn[d*(i-1) + j  ];
            p[i*d+j] = ((pn_right + pn_left) * sq(ds)
                        +(pn_below + pn_above) * sq(ds)) /
                    (2 * (sq(ds) + sq(ds))) -
                    sq(ds) * sq(ds) / (2 * (sq(ds) + sq(ds))) *
                    b[i*d+j];
            }
        }
```

Preallocated

Standard C Optimizations II

```c
for(unsigned int q=0;q<pit;q++){
    //Swap p and pn
    double *tmp = sim->p;
    sim->p = sim->pn;
    sim->pn = tmp;

    double *restrict p = sim->p;
    double *restrict pn = sim->pn;
    for(size_t i=1;i<d-1;i++){
        for(size_t j=1;j<d-1;j++){
            p[i*d+j] = ((pn_right + pn_left) * sq(ds)
                            + (pn_below + pn_above) * sq(ds))
                        / (2 * (sq(ds) + sq(ds)))
                        - sq(ds) * sq(ds) / (2 * (sq(ds) + sq(ds))) * b[i*d+j];
        }
    }
    for(size_t i=0;i<d;i++) p[d*i     + d-1] = p[d*i + d-2];
    for(size_t i=0;i<d;i++) p[d*i     + 0  ] = p[d*i + 1];
    for(size_t j=0;j<d;j++) p[d*0     + j  ] = p[d*1 + j];
    for(size_t j=0;j<d;j++) p[d*(d-1) + j  ] = 0;
}
```

Preallocated

```c
for(unsigned int q=0;q<pit;q++){
    //Swap p and pn
    double *tmp = sim->p;
    sim->p = sim->pn;
    sim->pn = tmp;

    double *restrict p = sim->p;
    double *restrict pn = sim->pn;
    for(size_t i=1;i<d-1;i++){
        for(size_t j=1;j<d-1;j++){
            p[i*d+j] = (pn_right + pn_left + pn_below + pn_above
                        - b[i*d + j]) / 4.0;
        }
    }
    for(size_t i=0;i<d;i++) p[d*i     + d-1] = p[d*i + d-2];
    for(size_t i=0;i<d;i++) p[d*i     + 0  ] = p[d*i + 1];
    for(size_t j=0;j<d;j++) p[d*0     + j  ] = p[d*1 + j];
    for(size_t j=0;j<d;j++) p[d*(d-1) + j  ] = 0;
}
```

Faster Math

Blocking the Time Loop (Skewed)

- Elements are re-used for each new iteration of the time loop in *pressure_poisson*
- Improve cache-friendliness by blocking the time loop:

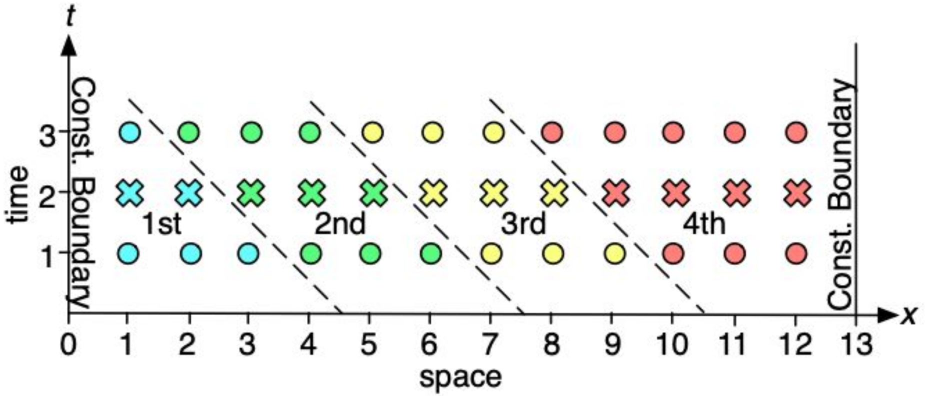
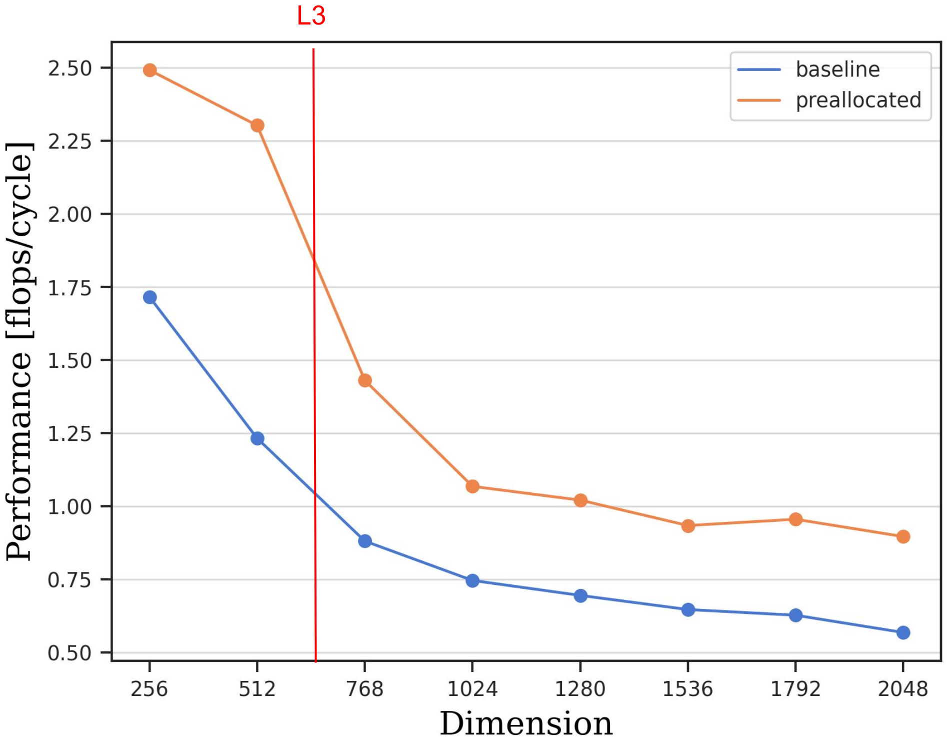


Figure from S. Kamil et al. “Implicit and Explicit Optimizations for Stencil Computations” (2006).

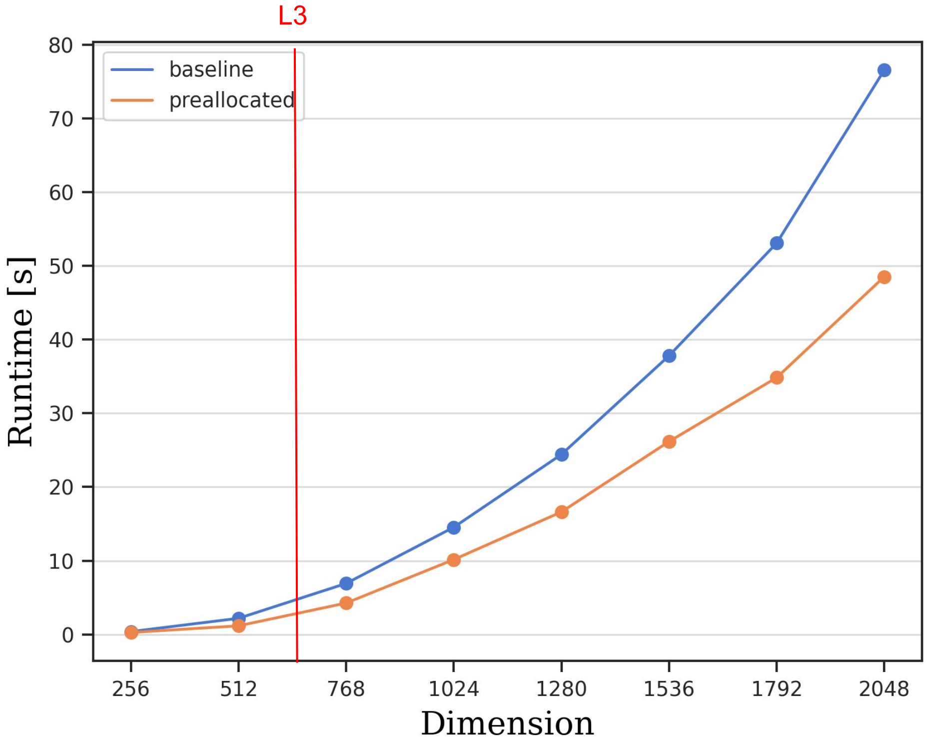
Experimental Setup

- Platform: Skylake (linux-x86)
- Compiler: gcc-12.3.0
- CPU: Intel(R) Core(TM) i7-6700
- Ports 0 and 1 can schedule flops including FMAs, hence theoretical peak performance:
  - Without SIMD, with FMAs: 4 flops/cycle
  - Without SIMD, without FMAs: 2 flops/cycle

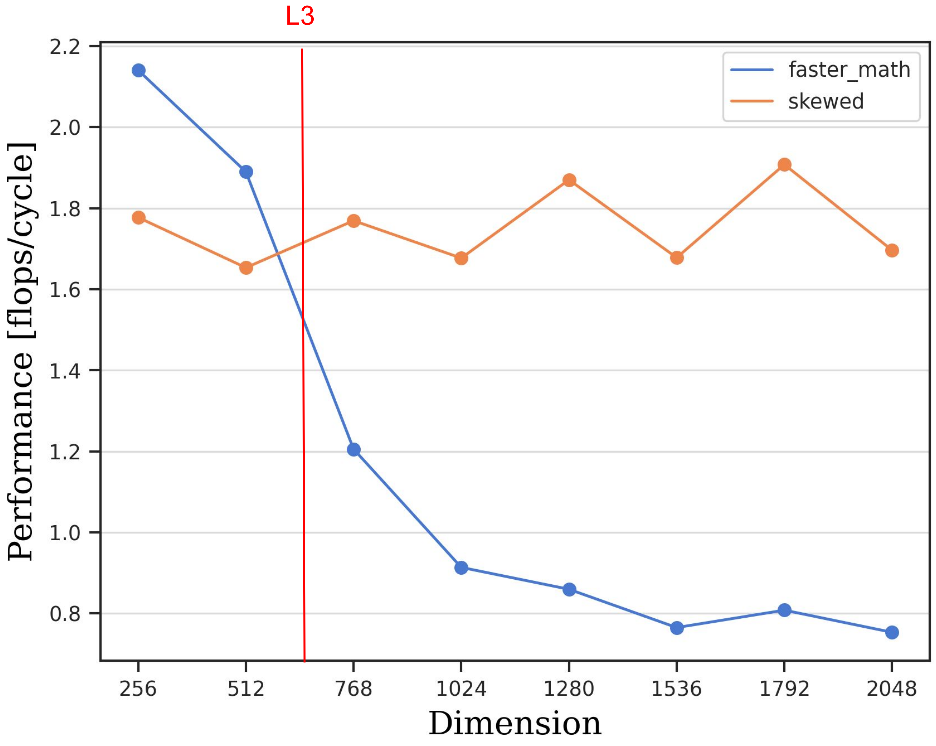
Experimental Results: Performance Plot (I)



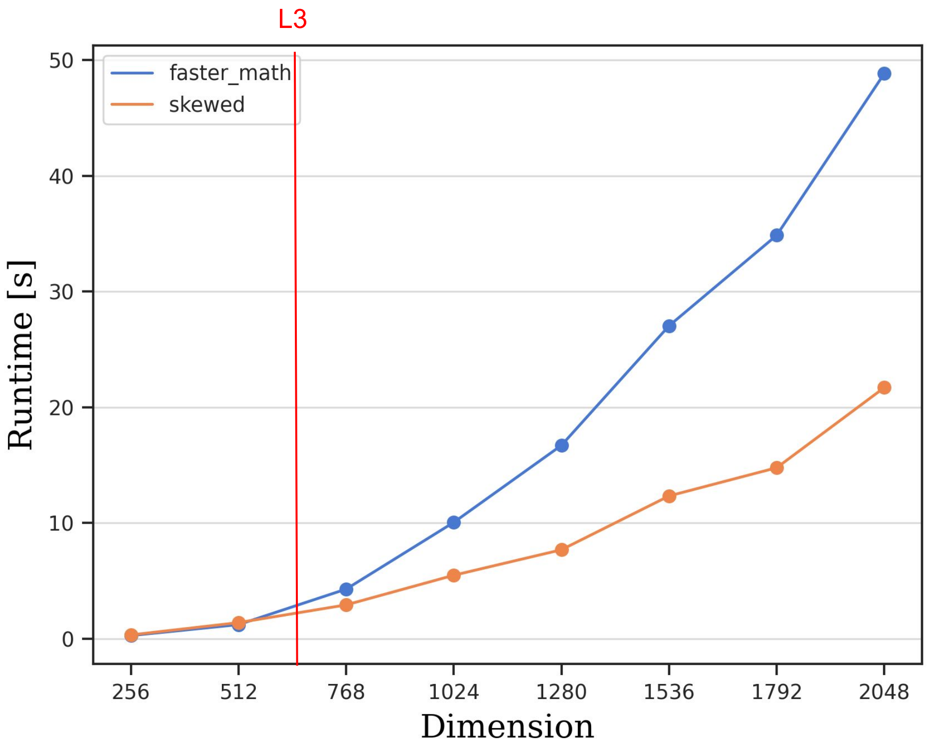
Experimental Results: Runtime Plot (I)



Experimental Results: Performance Plot (II)

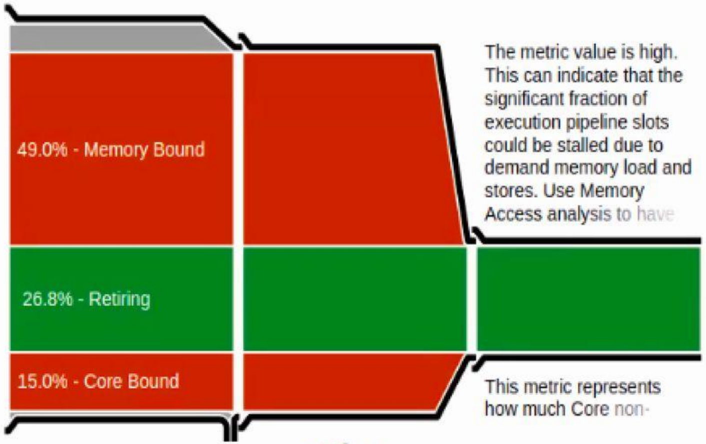


Experimental Results: Runtime Plot (II)



Backup Slide: Baseline Profiler

Clockticks:	734,474,800,000	
Instructions Retired:	613,196,800,000	
CPI Rate:	1.198	
MUX Reliability:	0.992	
Retiring:	26.8%	of Pipeline Slots
Front-End Bound:	7.1%	of Pipeline Slots
Bad Speculation:	2.0%	of Pipeline Slots
Back-End Bound:	64.0%	of Pipeline Slots
Memory Bound:	49.0%	of Pipeline Slots
L1 Bound:	4.0%	of Clockticks
L2 Bound:	3.2%	of Clockticks
L3 Bound:	2.3%	of Clockticks
DRAM Bound:	45.6%	of Clockticks
Memory Bandwidth:	69.2%	of Clockticks
Memory Latency:	12.1%	of Clockticks
Store Bound:	0.4%	of Clockticks
Core Bound:	15.0%	of Pipeline Slots
Average CPU Frequency:	3.4 GHz	
Total Thread Count:	1	
Paused Time:	0s	

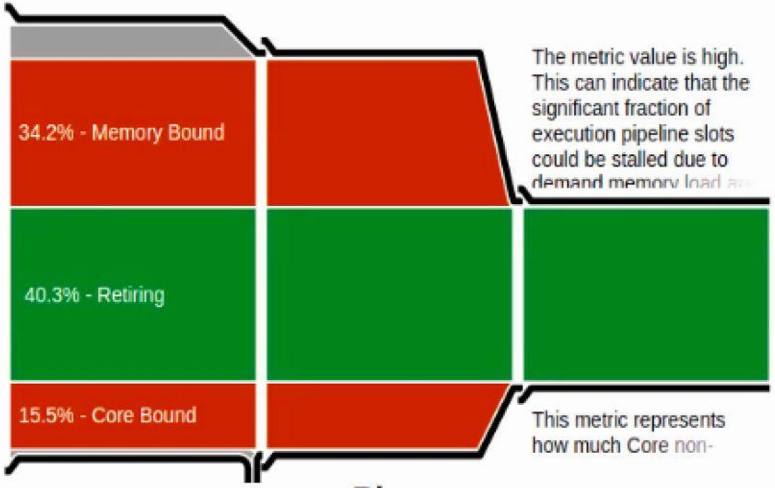


µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Backup Slide: Preallocated Profiler

Clockticks:	468,333,000,000	
Instructions Retired:	627,194,600,000	
CPI Rate:	0.747	
MUX Reliability:	0.988	
Retiring:	40.3%	of Pipeline Slots
Front-End Bound:	7.8%	of Pipeline Slots
Bad Speculation:	2.2%	of Pipeline Slots
Back-End Bound:	49.7%	of Pipeline Slots
Memory Bound:	34.2%	of Pipeline Slots
L1 Bound:	5.2%	of Clockticks
L2 Bound:	7.8%	of Clockticks
L3 Bound:	2.6%	of Clockticks
DRAM Bound:	24.2%	of Clockticks
Memory Bandwidth:	65.0%	of Clockticks
Memory Latency:	17.1%	of Clockticks
Store Bound:	0.1%	of Clockticks
Core Bound:	15.5%	of Pipeline Slots
Average CPU Frequency:	3.4 GHz	
Total Thread Count:	1	
Paused Time:	0s	

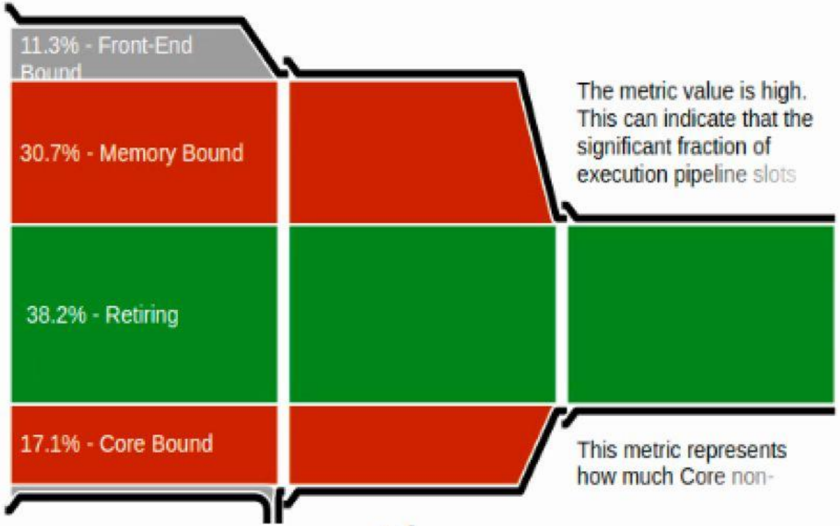


µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Backup Slide: Std. Opt. II Profiler

Clockticks:	503,829,000,000	
Instructions Retired:	631,335,800,000	
CPI Rate:	0.798	
MUX Reliability:	0.990	
Retiring:	38.2%	of Pipeline Slots
Front-End Bound:	11.3%	of Pipeline Slots
Bad Speculation:	2.6%	of Pipeline Slots
Back-End Bound:	47.8%	of Pipeline Slots
Memory Bound:	30.7%	of Pipeline Slots
L1 Bound:	4.6%	of Clockticks
L2 Bound:	7.8%	of Clockticks
L3 Bound:	2.3%	of Clockticks
DRAM Bound:	23.6%	of Clockticks
Memory Bandwidth:	61.6%	of Clockticks
Memory Latency:	16.2%	of Clockticks
Store Bound:	0.1%	of Clockticks
Core Bound:	17.1%	of Pipeline Slots
Average CPU Frequency:	3.4 GHz	
Total Thread Count:	1	
Paused Time:	0s	

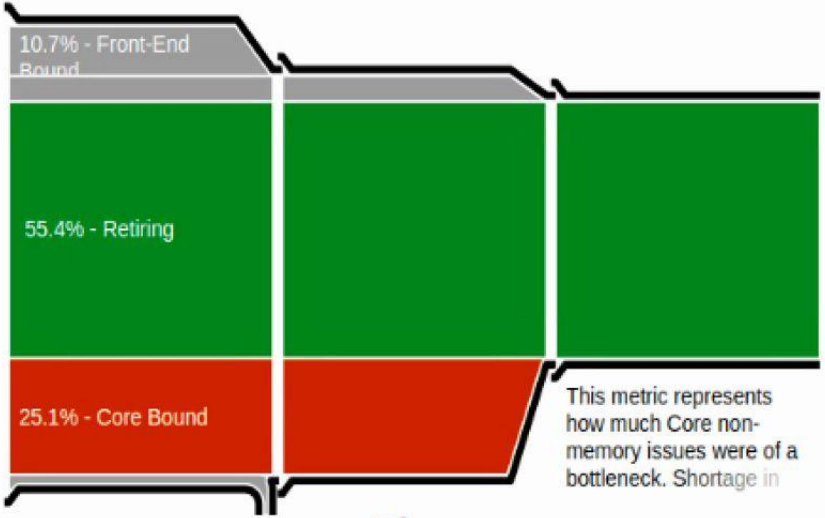


μPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Backup Slide: Skewed Profiler

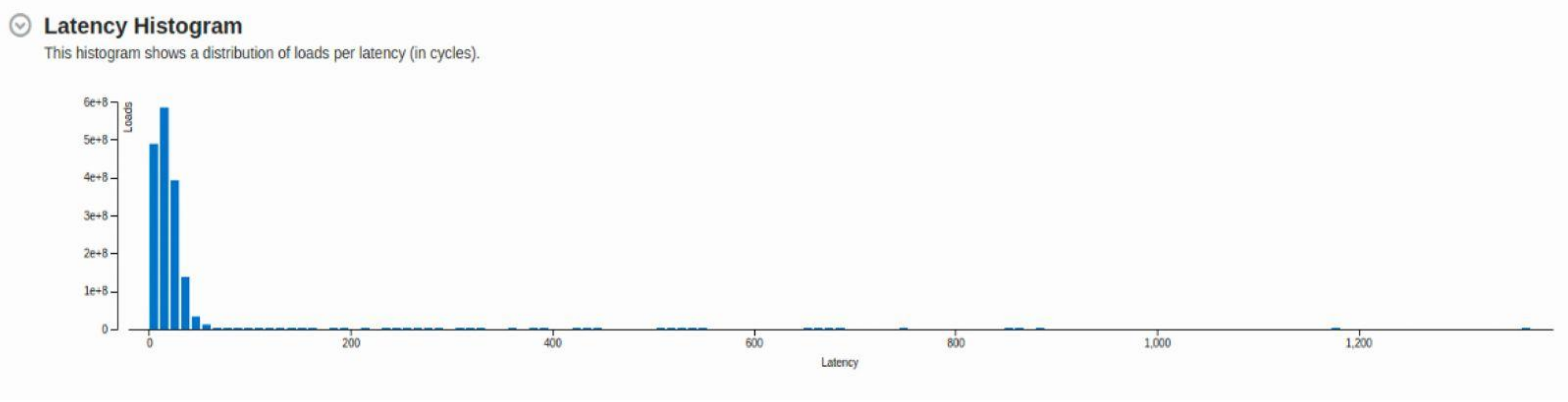
Clockticks:	286,008,000,000	
Instructions Retired:	547,090,600,000	
CPI Rate:	0.523	
MUX Reliability:	0.988	
Retiring:	55.4%	of Pipeline Slots
Front-End Bound:	10.7%	of Pipeline Slots
Bad Speculation:	3.2%	of Pipeline Slots
Back-End Bound:	30.7%	of Pipeline Slots
Memory Bound:	5.6%	of Pipeline Slots
Core Bound:	25.1%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	19.5%	of Clockticks
Cycles of 0 Ports Utilized:	8.7%	of Clockticks
Cycles of 1 Port Utilized:	7.7%	of Clockticks
Cycles of 2 Ports Utilized:	16.5%	of Clockticks
Cycles of 3+ Ports Utilized:	61.8%	of Clockticks
Vector Capacity Usage (FPU):	25.0%	
Average CPU Frequency:	3.4 GHz	
Total Thread Count:	1	
Paused Time:	0s	



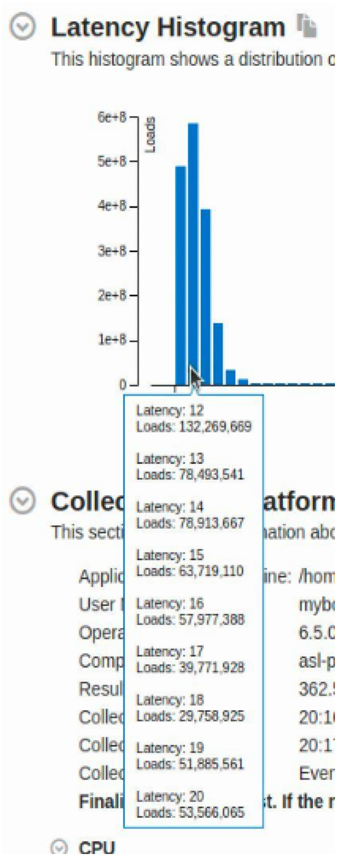
µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Backup Slide: Latency Histogram Skewed



Backup Slide: Latency Histogram Skewed



Backup Slide: Heatmap Skewed Params