MULTI-CORE PROGRAMMING ASSIGNMENT

Dongrui She

December 9, 2010

Goal of the Assignment

The purpose of this assignment is to

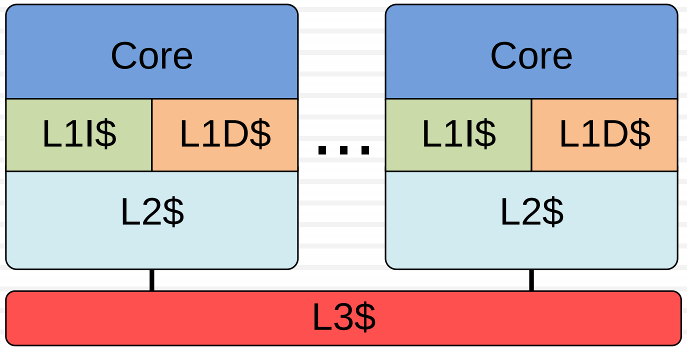
- Have in-depth understanding of the architectures of real-world multi-core CPUs
- Learn about how to develop parallel applications on such architectures, and how to analyze the performance in a real environment

Outline

- Parallelism in Mainstream CPUs
- Exploiting Parallelism in CPUs
- Methods to analyze application performance
- Introduction to VTune
- Example: Matrix Multiplication

Parallelism in Typical Mainstream CPUs

- 1-12 cores with shared memory
  - Large on-chip cache
  - Both private and shared cache
- Inside a core:
  - ILP: 3-4 issue out-of-order superscalar core
  - DLP: 128-bit SIMD instructions (SSE)
  - TLP: 2-way SMT (Intel's hyper-threading)
  - Typical core frequency: 2-3 GHz



Ways to Exploit Parallelism

- ILP: cannot be controlled directly
  - Compiler optimization and proper coding style can help
- TLP and multi-core: multi-threaded programming
  - Logically they are the same for the OS
  - Many programming models available, e.g., OpenMP, Cilk, pthread.
    - We will introduce OpenMP in more detail latter

Exploiting DLP

- Two ways to do vectorization
  - Auto-vectorization by compilers
    - The Intel compiler is considered the best
    - Most compilers are limited to (simple) inner-most loops
    - Pragmas can be use to tell compilers more information to enable more aggressive optimization
  - Intrinsics or inline assembly
    - Vectorization by programmers, more information about app.
    - Examples / Documentation will be on the assignment website
- Two most common obstacle
  - Cross-iteration dependency
  - Alignment issues

Vectorization Example – FIR

- Basic idea to vectorize a loop: unroll and pack multiple scalar iterations into one vector iteration
- Inner-most loop is an obvious choice, but
  - Packing and unpack can be costly, especially if the trip count is not aligned with the machine vector length
  - Inner loops may have low trip count

```
for(i=0; i< N; i++){
  s = 0;
  for(j=0; j < 4; j ++)
    s += x[i+j] * c[j]
  y[i] = s
}
```

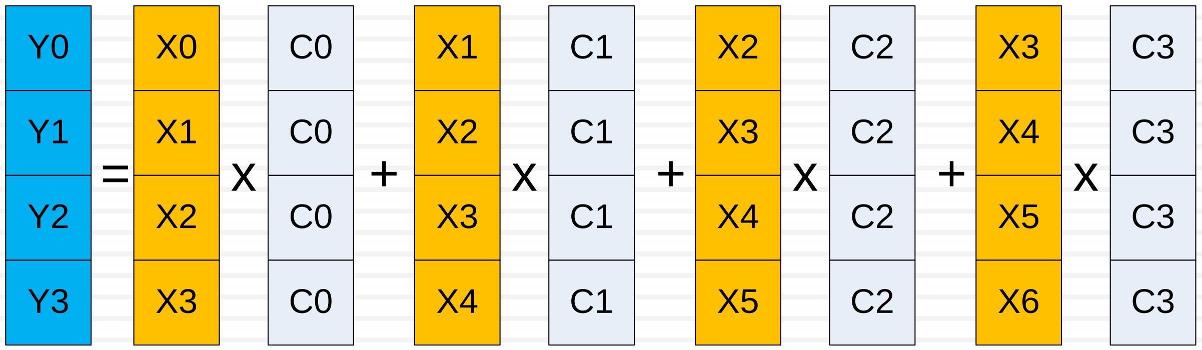
```
vc[0:3] = {c[0], c[1],
           c[2], c[3]}
for (i=0; i< N; i++){
  vs[0:3] = x[i:i+3] * vc
  y[i] = sum(vs[0:3])
}
```

Vectorization Example – FIR (2)

- Outer-loop vectorization can be more efficient
  - However, most compiler cannot do it

```
for(i=0; i<N; i++){
   s = 0;
   for(j=0; j<4; j++)
      s += x[i+j] * c[j]
   y[i] = s
}
```

```
for (i=0; i< N; i+=4){
   vs[0:3] = {0, 0, 0, 0}
   for (j =0; j < 4; j ++) {
         vc[0:3] = vsplat(c[j])
         vs[0:3] += x[i+j:i+j+3] * vc
   }
   y[i:i+3] = vs[0:3]
}
```



Analyzing Application Performance

- Understand and optimize application performance
  - Is the performance good or bad?
  - Which part should run in parallel?
  - Where to optimize?
- Static analysis and execution time measurement are not enough
  - They are not enough to understand the dynamic behavior of complex applications
  - We need profiling

Ways to Profile an Application

- Emulation/Simulation
  - Accurate (if the model is accurate enough) but slow
- Intrusive profilers:
  - The profiling codes may change the program (timing) behavior
- Statistical profilers:
  - Periodically halt the program and sample the PC and other data. Less overhead and better overall accuracy
  - Most commonly used profilers are based on this approach, e.g., Intel VTune, AMD CodeAnalyst.

Profiling with Intel VTune

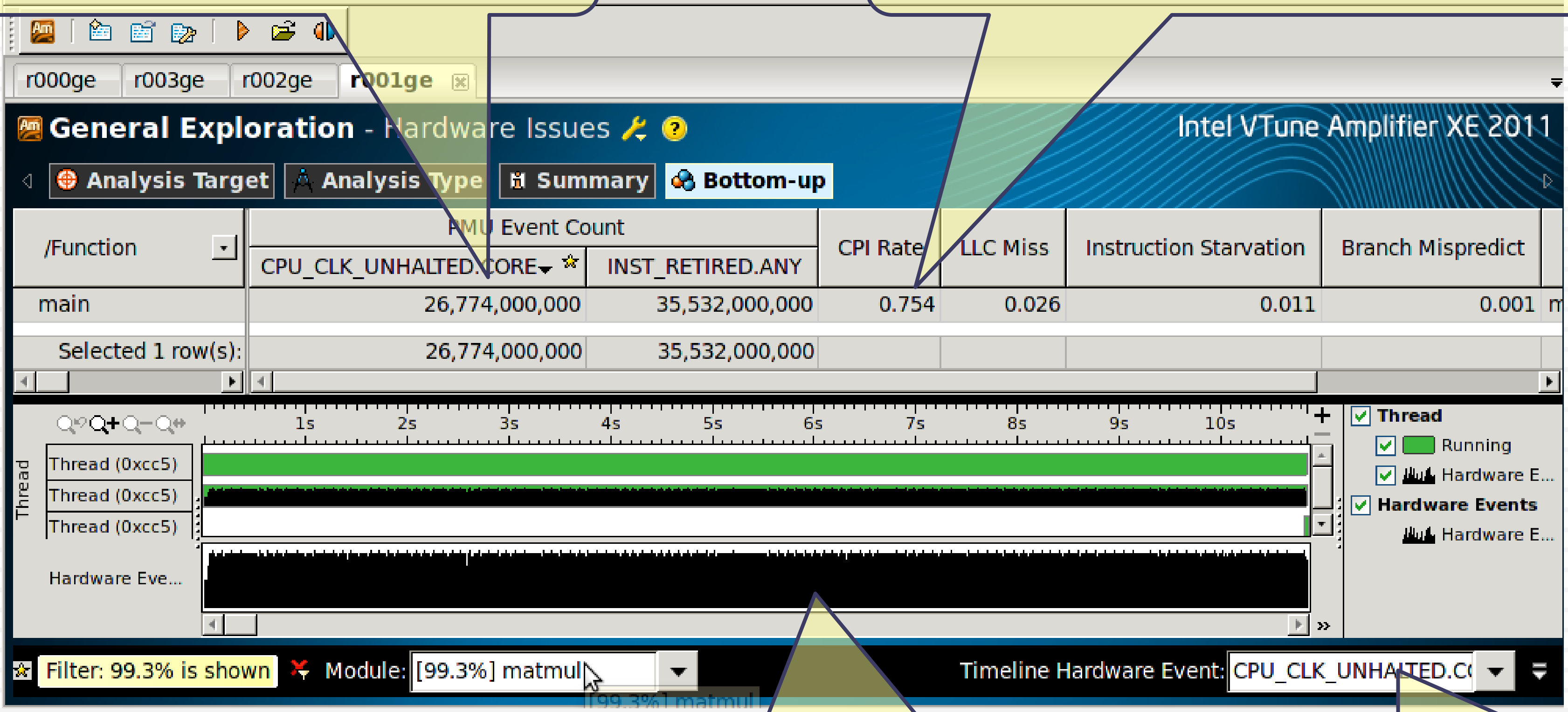
- Features
  - Based on sampling
  - Supports event counters in the PMU (Performance Monitoring Unit) of Intel CPUs
- Requirements
  - Intel CPUs (Core or newer) running Linux or Windows
  - Program compiled with debug symbols (-g)
- Alternatives
  - For AMD CPUs: AMD CodeAnalyst is similar to VTune
  - Open source solution for Linux: pfmon
    - May require some effort to get it running

VTune User Interface

- Information of each function

Total number of cycles (for main);
Note: these are statistical values

Cycles per instruction ratio(for main);
Note: these are derived values



/Function	PMU Event Count		CPI Rate	LLC Miss	Instruction Starvation	Branch Mispredict
	CPU_CLK_UNHALTED.CORE	INST_RETIRED.ANY				
main	26,774,000,000	35,532,000,000	0.754	0.026	0.011	0.001
Selected 1 row(s):	26,774,000,000	35,532,000,000				

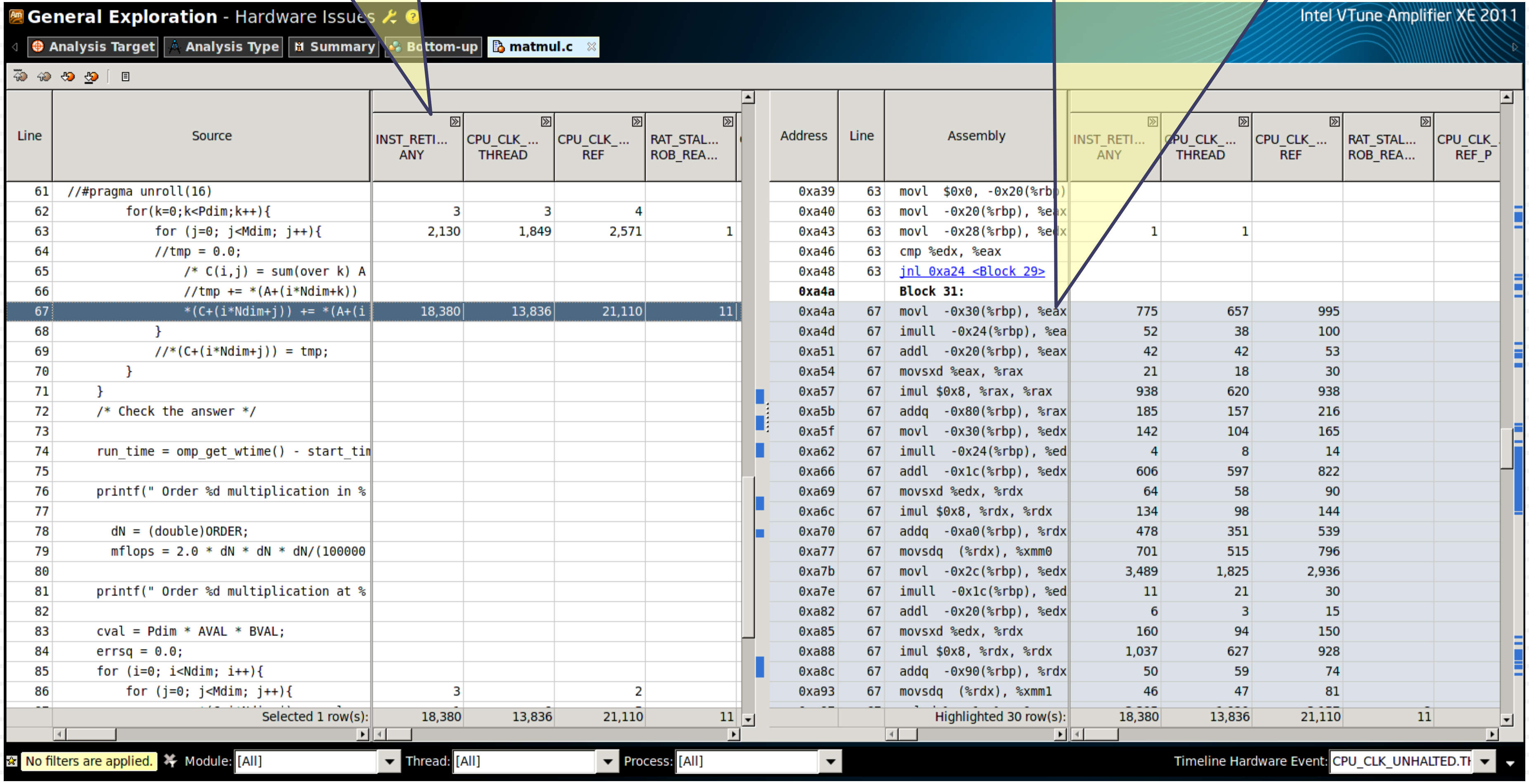
Distribution of events over time

Choose event to show in the distribution diagram

VTune Detailed View

Counter values

Shaded lines are corresponding disassembly of the selected line in the original source



Line	Source	INST_RETI... ANY	CPU_CLK_... THREAD	CPU_CLK_... REF	RAT_STAL... ROB_REA...
61	`//#pragma unroll(16)`				
62	`for(k=0;k<Pdim;k++){`	3	3	4	
63	`for (j=0; j<Mdim; j++){`	2,130	1,849	2,571	1
64	`//tmp = 0.0;`				
65	`/* C(i,j) = sum(over k) A`				
66	`//tmp += *(A+(i*Ndim+k))`				
67	`*(C+(i*Ndim+j)) += *(A+(i`	18,380	13,836	21,110	11
68	`}`				
69	`//*(C+(i*Ndim+j)) = tmp;`				
70	`}`				
71	`}`				
72	`/* Check the answer */`				
73					
74	`run_time = omp_get_wtime() - start_tin`				
75					
76	`printf(" Order %d multiplication in %`				
77					
78	`dN = (double)ORDER;`				
79	`mflops = 2.0 * dN * dN * dN/(100000`				
80					
81	`printf(" Order %d multiplication at %`				
82					
83	`cval = Pdim * AVAL * BVAL;`				
84	`errsq = 0.0;`				
85	`for (i=0; i<Ndim; i++){`				
86	`for (j=0; j<Mdim; j++){`	3		2	
	Selected 1 row(s):	18,380	13,836	21,110	11

Address	Line	Assembly	INST_RETI... ANY	CPU_CLK_... THREAD	CPU_CLK_... REF	RAT_STAL... ROB_REA...	CPU_CLK_ REF_P
0xa39	63	`movl $0x0, -0x20(%rbp)`					
0xa40	63	`movl -0x20(%rbp), %eax`					
0xa43	63	`movl -0x28(%rbp), %edx`	1	1			
0xa46	63	`cmp %edx, %eax`					
0xa48	63	`jnl 0xa24 <Block 29>`					
0xa4a		**Block 31:**					
0xa4a	67	`movl -0x30(%rbp), %eax`	775	657	995		
0xa4d	67	`imull -0x24(%rbp), %ea`	52	38	100		
0xa51	67	`addl -0x20(%rbp), %eax`	42	42	53		
0xa54	67	`movsxd %eax, %rax`	21	18	30		
0xa57	67	`imul $0x8, %rax, %rax`	938	620	938		
0xa5b	67	`addq -0x80(%rbp), %rax`	185	157	216		
0xa5f	67	`movl -0x30(%rbp), %edx`	142	104	165		
0xa62	67	`imull -0x24(%rbp), %ed`	4	8	14		
0xa66	67	`addl -0x1c(%rbp), %edx`	606	597	822		
0xa69	67	`movsxd %edx, %rdx`	64	58	90		
0xa6c	67	`imul $0x8, %rdx, %rdx`	134	98	144		
0xa70	67	`addq -0xa0(%rbp), %rdx`	478	351	539		
0xa77	67	`movsdq (%rdx), %xmm0`	701	515	796		
0xa7b	67	`movl -0x2c(%rbp), %edx`	3,489	1,825	2,936		
0xa7e	67	`imull -0x1c(%rbp), %ed`	11	21	30		
0xa82	67	`addl -0x20(%rbp), %edx`	6	3	15		
0xa85	67	`movsxd %edx, %rdx`	160	94	150		
0xa88	67	`imul $0x8, %rdx, %rdx`	1,037	627	928		
0xa8c	67	`addq -0x90(%rbp), %rdx`	50	59	74		
0xa93	67	`movsdq (%rdx), %xmm1`	46	47	81		
		Highlighted 30 row(s):	18,380	13,836	21,110	11	

Example: Matrix Multiplication

- Straightforward implementation

```
A = (double *)malloc(N*P*sizeof(double));
B = (double *)malloc(P*M*sizeof(double));
C = (double *)malloc(N*M*sizeof(double));
... // Initialize A, B and C
for (i=0; i < N; i++){
    for (j=0; j < M; j++){
        for(k=0; k <P; k++){
           *(C+(i*N+j))+= *(A+(i*N+k)) * *(B+(k*P+j));
        }
    }
}
```

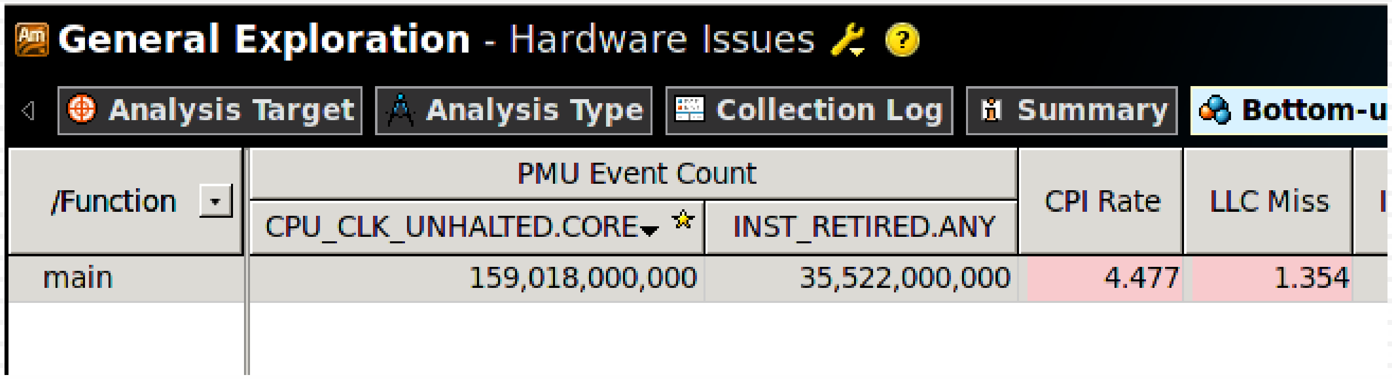
1024x1024 matrices. Program compiled with optimization off (-O0), performance on a Core 2 Quad 8300 with 32bit Linux:

Order 1024 multiplication in 58.7 seconds
Order 1024 multiplication at 36.6 mflops

Number of operations is 2*N*N*N

Initial Profiling Result

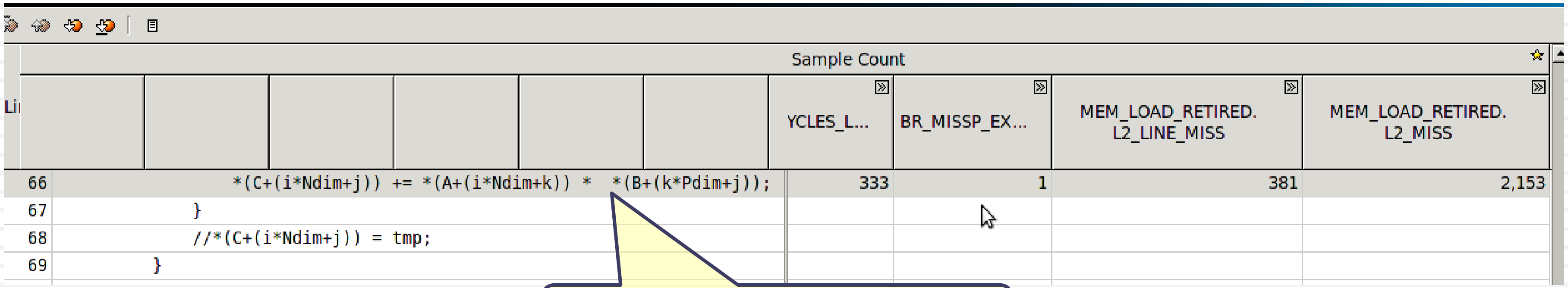
- CPI is very high, and LLC miss is an obvious problem



General Exploration - Hardware Issues

/Function	PMU Event Count		CPI Rate	LLC Miss
	CPU_CLK_UNHALTED.CORE	INST_RETIRED.ANY		
main	159,018,000,000	35,522,000,000	4.477	1.354

- The inner-most loop is causing a lot of cache misses



Li	Source	YCLES_L...	BR_MISSP_EX...	MEM_LOAD_RETIRED.L2_LINE_MISS	MEM_LOAD_RETIRED.L2_MISS
		Sample Count			
66	`*(C+(i*Ndim+j)) += *(A+(i*Ndim+k)) * *(B+(k*Pdim+j));`	333	1	381	2,153
67	`}`				
68	`//*(C+(i*Ndim+j)) = tmp;`				
69	`}`				

This line is the hotspot

Analyze the Problem

- Access pattern of B is the problem

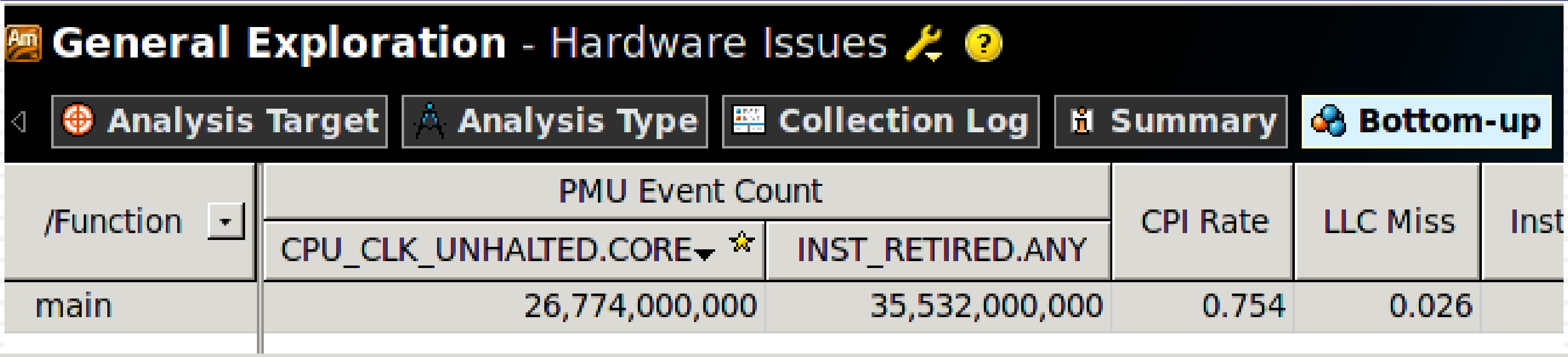
```
A = (double *)malloc(N*P*sizeof(double));
B = (double *)malloc(P*M*sizeof(double));
C = (double *)malloc(N*M*sizeof(double));
... // Initialize A, B and C
for (i=0; i < N; i++){
    for(k=0; k <P; k++){
        for (j=0; j < M; j++){
            *(C+(i*N+j))+= *(A+(i*N+k)) * *(B+(k*P+j));
        }
    }
}
```

Just inter-change the loops

```
Order 1024 multiplication in 10.315131 seconds
Order 1024 multiplication at 208.187717 mflops
```

~5.7x speed-up by a minor change !

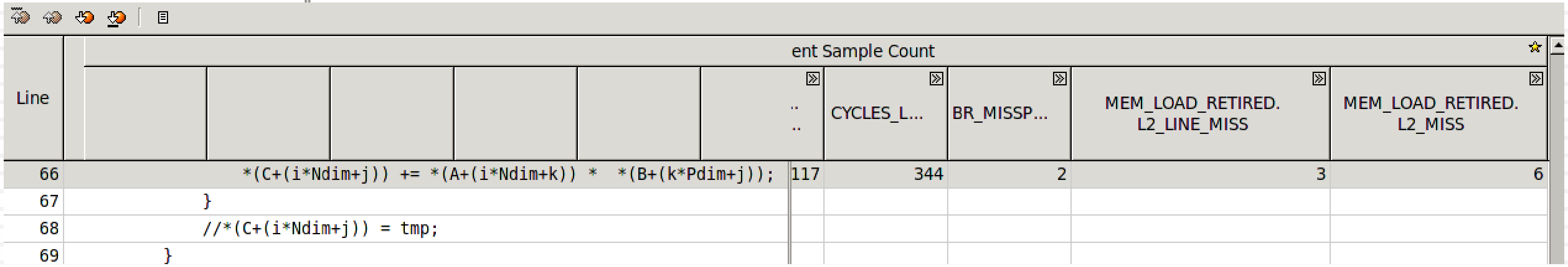
Profiling Result After Optimization



General Exploration - Hardware Issues

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up

/Function	PMU Event Count		CPI Rate	LLC Miss	Inst
	CPU_CLK_UNHALTED.CORE	INST_RETIRED.ANY			
main	26,774,000,000	35,532,000,000	0.754	0.026	



Line		ent Sample Count				
		..	CYCLES_L...	BR_MISSP...	MEM_LOAD_RETIRED. L2_LINE_MISS	MEM_LOAD_RETIRED. L2_MISS
66	`*(C+(i*Ndim+j)) += *(A+(i*Ndim+k)) * *(B+(k*Pdim+j));`	117	344	2	3	6
67	`}`					
68	`//*(C+(i*Ndim+j)) = tmp;`					
69	`}`					

In this case, compilers should be able to interchange the loop automatically (in our experiment, ICC can, but GCC cannot). But further optimizations like tiling still need to be done by hand.

Use *-fast* in ICC and you get:

```
Order 1024 multiplication in 0.507935 seconds
Order 1024 multiplication at 4227.870591 mflops
```

Assignment Setup

- Platform: a PC with multi-core CPU
  - TU/e Notebook 2009 and 2010 are OK
- Software: Intel compiler and VTune Profiler
  - Available on both Windows and Linux
  - A 30-day evaluation license can be obtained from the web-site for free
    - For linux, a 1-year non-commercial license is available
- Assignment can be done in team of two students
  - Make sure at least one has the proper platform

Some General Remarks

- Both GCC and ICC have options to report whether the loops are vectorized and if not what's the reason. It can be quite helpful
- ICC's optimization tends to be quite aggressive, but it doesn't always payoff. So check the manual and use the proper flags and pragmas
- Bear in mind that VTune is based on sampling. So the numbers are NOT exact