VTune 使用

李奕琛

February 2020

目录

1 VTune 简要介绍

VTune 是 Intel 推出的一款可视化的性能剖析（profiling）工具，可以在 linux，Windows 系统上运行。所谓 profiling 是指通过对目标收集采样或快照来归纳目标特征。例如分析 CPU 的使用率时，可以通过对程序计数器采样，或者跟踪栈来找到消耗 CPU 周期的代码路径，从而找到程序中的占用 CPU 使用率高的函数。通过对程序的性能分析，可以帮助开发人员针对系统资源的使用来优化代码。常见的剖析工具有：DTrace、perf、VTune 等。这里我们介绍 VTune 的使用。

2 VTune 的安装及使用

2.1 VTune 安装

VTune 在 Intel 官网即可免费下载安装：https://software.intel.com/en-us/vtune/choose-download，这里展示使用的是 VTune_Amplifier_2019 版本在 windows 系统上的使用。

2.2 VTune 使用

VTune 需要以管理员的身份打开，打开进入页面并创建新项目。

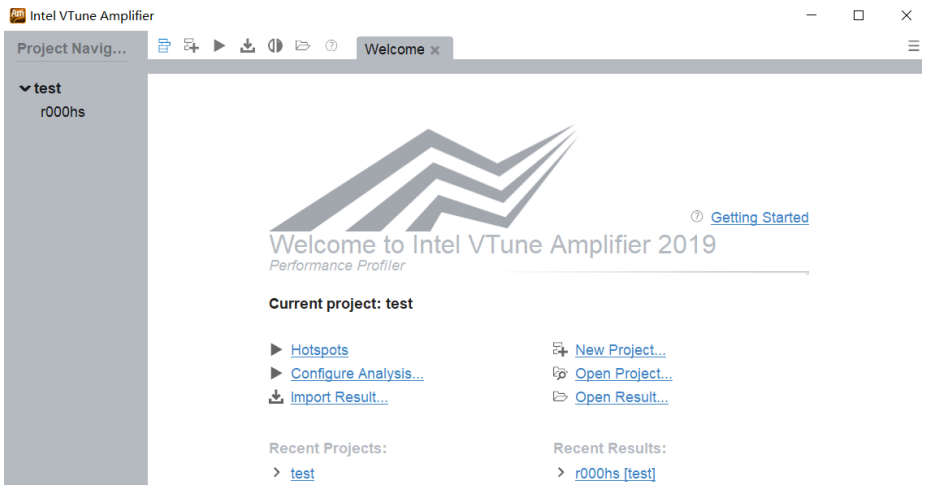


图 2.1: VTune 主页面

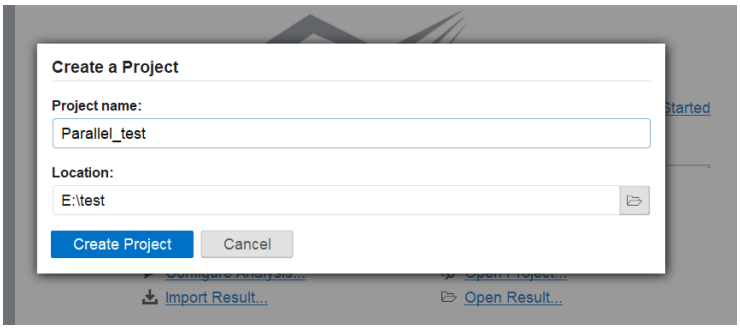


图 2.2: 创建项目

点击 Configure Analysis 进入分析界面，左下角 Launch Application 中选中需要进行测试的程序以及输入参数，右侧提供了：Hotspots, Microarchitecture, Parallelism 等多种分析类型。设置 CPU 的采样间隔时间，以及额外的测试内容即可测试对应程序性能。

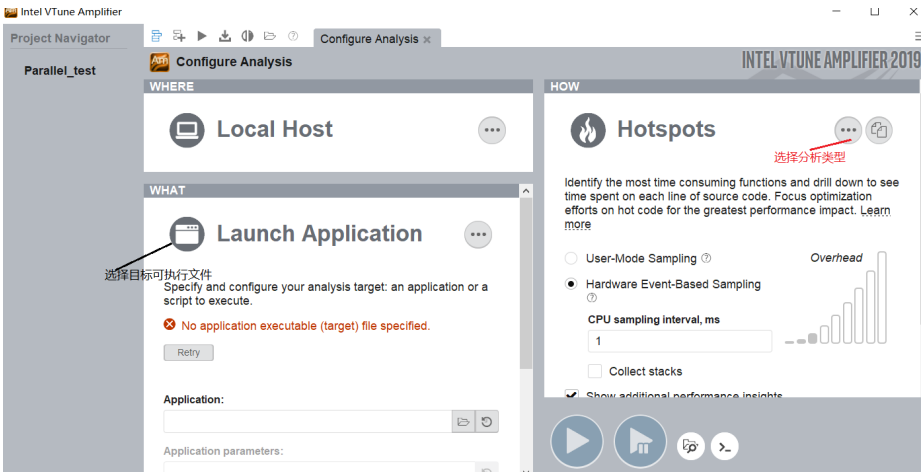


图 2.3: 配置分析项目

以 Hotspots 为例，选中并进行测试，可以采集到 collection log, Summary, Bottom-up, Caller/Callee, Top-down Tree Plaform 数据。如图 2.4所示，Summary 主要分析的数据有：执行时间，高热点部分，CPU 使用直方图以及收集信息和平台信息。在这里可以看到总开销时间，程序中最耗时的部分等内容。Bottom-top 可以查看函数/线程调用时间。具体各数据类型大家可以自己进行查看，在这里不一一列举。接下来我们举两个例子来展示

VTune 的基本使用。

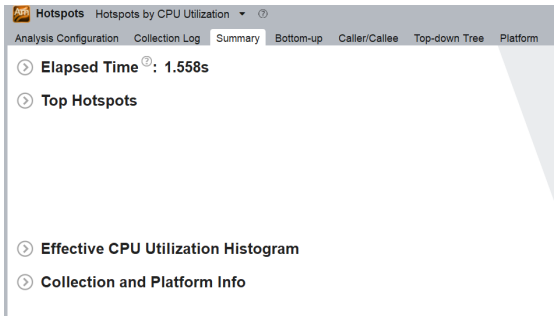


图 2.4: Hotspot 简单分析结果

3 体系结构相关的程序性能分析

3.1 cache 命中率

接下来我们以数组列求和的例子展示 VTune 分析程序性能。

```
1 for (i = 0; i < 1000; i++)
2     column_sum[i] = 0.0;
3     for (j = 0; j < 1000; j++)
4       column_sum[i] += b[j][i];
```

二维数组在 C/C++ 中为行主存储方式，这样按列访问数组可能会造成 cache 频繁 miss 的从而影响到程序执行的时间。

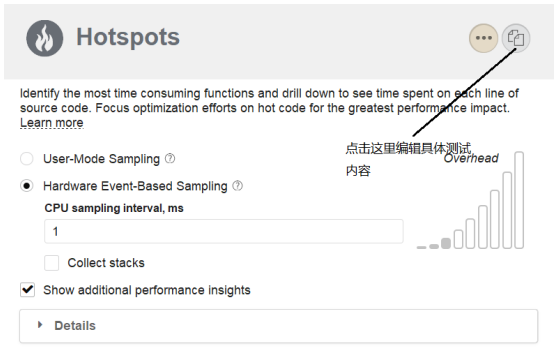
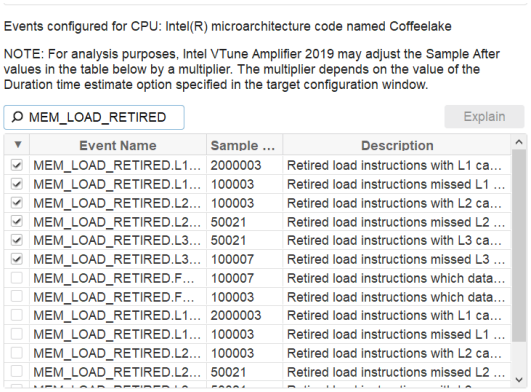


图 3.5: 选择硬件事件采集

为分析此程序，我们希望看到程序执行时间以及 cache miss 次数。后者需要额外加入额外的 Event，如图 3.5，在 Hotspots 窗口中选择 “Hardware Event-Based Sampling”。然后编辑希望采样的事件，如图 3.6所示，在 Events configured for CPU 中勾选 MEM_LOAD_RETIURED.L1_HIT、MEM_LOAD_RETIURED.L1_MISS、MEM_LOAD_RETIURED.L2_HIT、MEM_LOAD_RETIURED.L2_MISS、MEM_LOAD_RETIURED.L3_HIT、MEM_LOAD_RETIURED.L3_MISS。VTune 就会采样 CPU L1,L2,L3 cache 的 HIT 以及 MISS 的次数。图 3.7给出了测试结果。

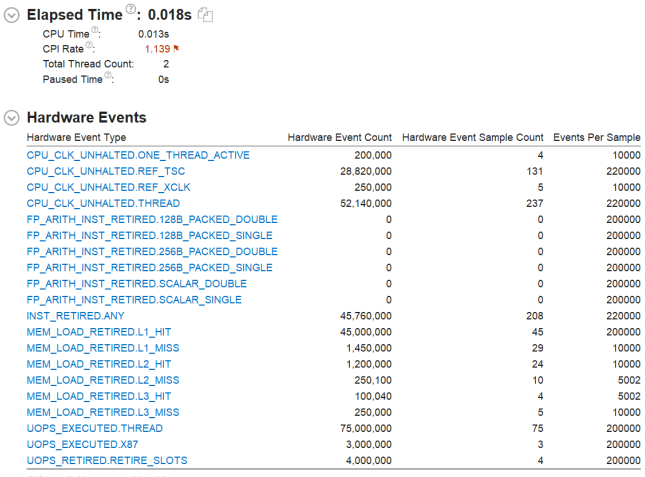

Events configured for CPU: Intel(R) microarchitecture code named Coffeelake

NOTE: For analysis purposes, Intel VTune Amplifier 2019 may adjust the Sample After values in the table below by a multiplier. The multiplier depends on the value of the Duration time estimate option specified in the target configuration window.

MEM_LOAD_RETIRED Explain

Event Name	Sample ...	Description
MEM_LOAD_RETIRED.L1...	2000003	Retired load instructions with L1 ca...
MEM_LOAD_RETIRED.L1...	100003	Retired load instructions missed L1 ...
MEM_LOAD_RETIRED.L2...	100003	Retired load instructions with L2 ca...
MEM_LOAD_RETIRED.L2...	50021	Retired load instructions missed L2 ...
MEM_LOAD_RETIRED.L3...	50021	Retired load instructions with L3 ca...
MEM_LOAD_RETIRED.L3...	100007	Retired load instructions missed L3 ...
MEM_LOAD_RETIRED.F...	100007	Retired load instructions which data...
MEM_LOAD_RETIRED.F...	100003	Retired load instructions which data...
MEM_LOAD_RETIRED.L1...	2000003	Retired load instructions with L1 ca...
MEM_LOAD_RETIRED.L1...	100003	Retired load instructions missed L1 ...
MEM_LOAD_RETIRED.L2...	100003	Retired load instructions with L2 ca...
MEM_LOAD_RETIRED.L2...	50021	Retired load instructions missed L2 ...

图 3.6: 选择采样事件


Elapsed Time: 0.018s

CPU Time: 0.013s
CPI Rate: 1.139
Total Thread Count: 2
Paused Time: 0s

Hardware Events

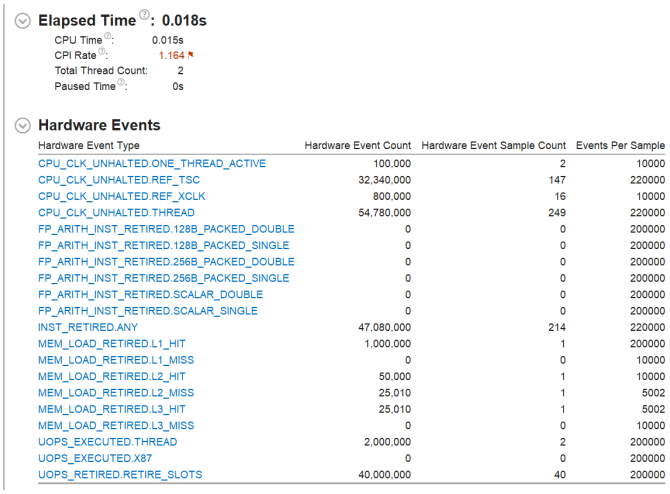
Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	200,000	4	10000
CPU_CLK_UNHALTED.REF_TSC	28,820,000	131	220000
CPU_CLK_UNHALTED.REF_XCLK	250,000	5	10000
CPU_CLK_UNHALTED.THREAD	52,140,000	237	220000
FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	0	0	200000
FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	0	0	200000
FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	0	0	200000
FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	0	0	200000
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	0	0	200000
FP_ARITH_INST_RETIRED.SCALAR_SINGLE	0	0	200000
INST_RETIRED.ANY	45,760,000	208	220000
MEM_LOAD_RETIRED.L1_HIT	45,000,000	45	200000
MEM_LOAD_RETIRED.L1_MISS	1,450,000	29	10000
MEM_LOAD_RETIRED.L2_HIT	1,200,000	24	10000
MEM_LOAD_RETIRED.L2_MISS	250,100	10	5002
MEM_LOAD_RETIRED.L3_HIT	100,040	4	5002
MEM_LOAD_RETIRED.L3_MISS	250,000	5	10000
UOPS_EXECUTED.THREAD	75,000,000	75	200000
UOPS_EXECUTED.X87	3,000,000	3	200000
UOPS_RETIRED.RETIRE_SLOTS	4,000,000	4	200000

图 3.7: 列主次序访问算法的分析结果

上述的代码中按列访问会导致较高的 cache miss 情况，我们重写代码：

```
1 for (i = 0; i < 1000; i++)
2   column_sum[i] = 0.0;
3 for (j = 0;  j < 1000; j++)
4   for (i = 0; i < 1000; i++)
5     column_sum[i] += b[j][i];
```

这样的访问方式与行主存储器匹配，进行测试可以得到结果 cache miss 的次数更少。测试结果如图 3.8所示。



Elapsed Time: 0.018s

CPU Time:	0.015s
CPI Rate:	1.164
Total Thread Count:	2
Paused Time:	0s

Hardware Events

Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	100,000	2	10000
CPU_CLK_UNHALTED.REF_TSC	32,340,000	147	220000
CPU_CLK_UNHALTED.REF_XCLK	800,000	16	10000
CPU_CLK_UNHALTED.THREAD	54,780,000	249	220000
FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	0	0	200000
FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	0	0	200000
FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	0	0	200000
FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	0	0	200000
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	0	0	200000
FP_ARITH_INST_RETIRED.SCALAR_SINGLE	0	0	200000
INST_RETIRED.ANY	47,080,000	214	220000
MEM_LOAD_RETIRED.L1_HIT	1,000,000	1	200000
MEM_LOAD_RETIRED.L1_MISS	0	0	10000
MEM_LOAD_RETIRED.L2_HIT	50,000	1	10000
MEM_LOAD_RETIRED.L2_MISS	25,010	1	5002
MEM_LOAD_RETIRED.L3_HIT	25,010	1	5002
MEM_LOAD_RETIRED.L3_MISS	0	0	10000
UOPS_EXECUTED.THREAD	2,000,000	2	200000
UOPS_EXECUTED.X87	0	0	200000
UOPS_RETIRED.RETIRE_SLOTS	40,000,000	40	200000

图 3.8: 行主次序访问算法的分析结果

3.2 超标量

以课堂介绍的 n 个数求和问题为例，两类算法设计思路：逐个累加的平凡算法（链式）；超标量优化算法，如最简单的两路链式累加，再如递归算法——两两相加、中间结果再两两相加，依次类推，直至只剩下最终结果。

我们比较两路链式累加算法和普通的链式算法的性能。对比普通的链式算法，两路链式算法能更好地利用 CPU 超标量架构，两条求和的链可令两条流水线充分地并发运行指令。有一个评价指标 IPC（Instruction Per Clock），即每个时钟周期执行的指令数，可以直观地比较这两种算法的区别。我们可以想到，两种算法所需的指令数大致相同，且两路链式算法同一时间令两条流水线充满，那么其 IPC 应该明显优于链式算法。接下来我们利用 VTune 来分析这两种算法的性能，验证我们的分析。

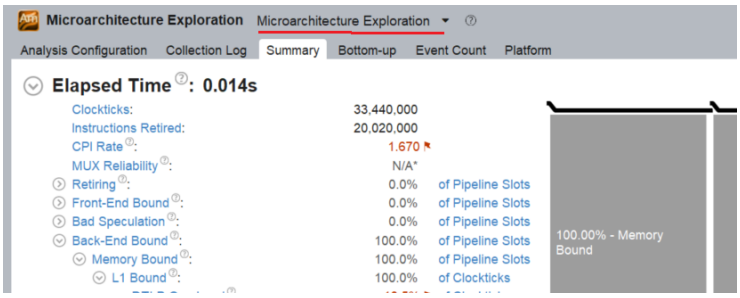


图 3.9: 超标量分析结果

如图 3.9所示, 我们选择 Microarchitecture Exploration 类型(在 Bottom-up 中可以看到具体执行的周期数), Summary 数据下我们可以看到总体执行的周期数(Clockticks), 执行指令数(Instructions Retired)以及 CPI(IPC 的倒数, 每条指令执行的周期数)。接下来我们进入 Bottom-up 数据栏, 如图 3.10所示, 在这一页面中我们可以看到具体每个函数执行的 CPU 时间, 周期数以及执行指令数 (gcc 编译时记得-g 来加入调试程序使用的附加信息)。在 chain_unroll (链式算法实现函数) 一栏, 我们可以看到执行的指令数为 3,640,000 (4096 个元素求和 500 次), 而执行的周期数为 2,080,000。其 CPI 为 0.571。

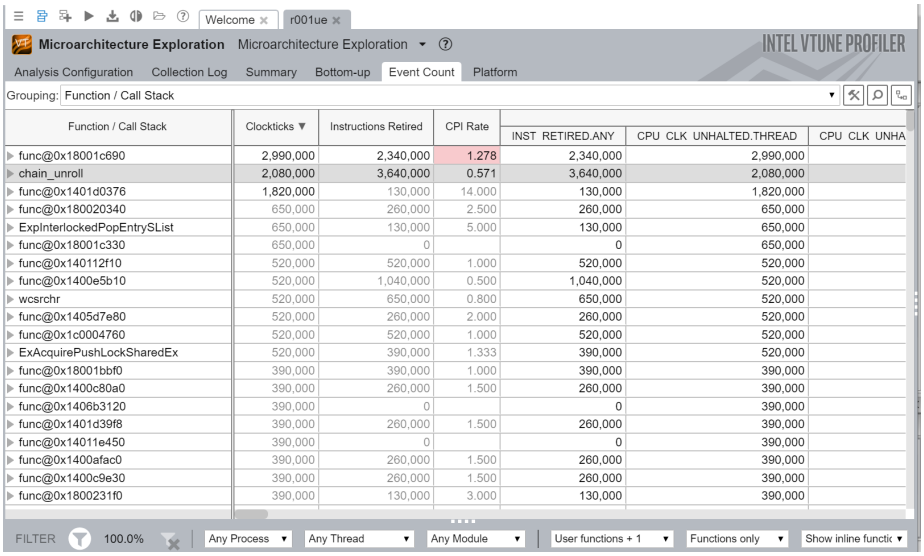


图 3.10: 链式算法详细结果

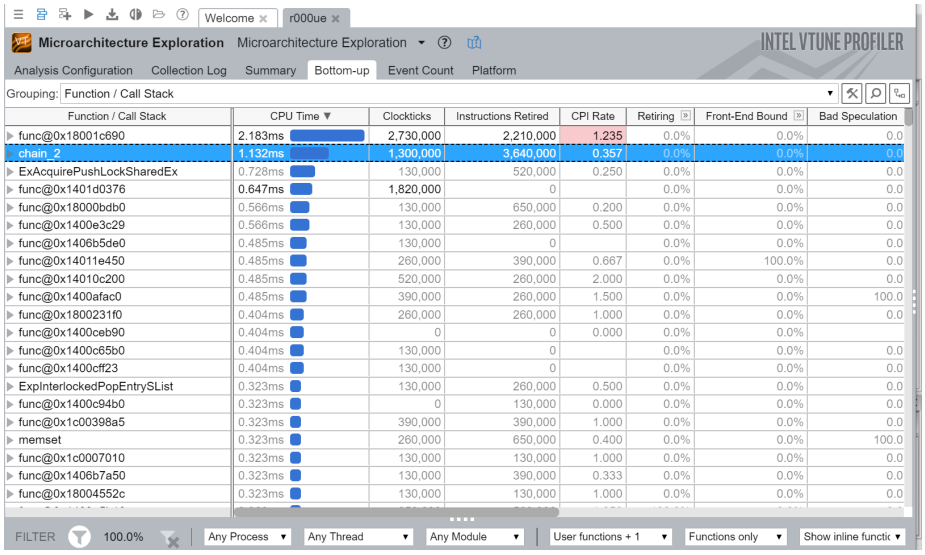


图 3.11: 两路链式算法详细结果

对比两路链式算法分析结果，如图 3.11所示，我们可以看到其执行指令数为 3,640,000，执行周期数为 1,300,000。其 CPI 为 0.357，明显优于链式算法的 0.571，这与我们之前的分析相同。大家可以将自己代码的测试结果与上述结果对比，看看有何异同。

VTune 的功能很强大，不仅能看到每个函数的执行情况，还可以看到具体每段代码以及对应汇编代码的执行情况。我们在图 3.11中双击 chain_2，可以看到 chain_2 函数中具体每段代码执行的次数以及执行周期数：

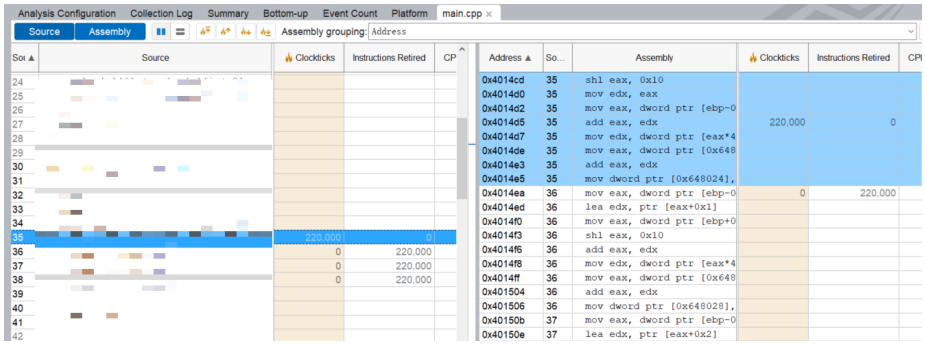


图 3.12: 两路链式算法执行执行分析

同样的，第一个例子中对 cache 命中率的测试也可以查看具体每个函

数每段代码的 cache 命中率。这里只介绍了两个简单的程序性能分析示例，详细的 VTune 使用手册大家可以在网站：https://software.intel.com/en-us/vtune-help 官方文档中查询具体每一种数据的测试和分析。希望大家举一反三，灵活使用 VTune（或其他 profiling 工具）剖析程序性能。