

体系结构调研

体系结构相关实验及性能测试

杜忱莹 周辰霏

2021 年 3 月

安祺 曾泉胜

2022 年 2 月

贺祎昕

2023 年 2 月

朱璟钰

2024 年 3 月

马浩祎

2025 年 3 月

目录

1 关于 AI 的使用	4
2 使用 LaTeX 撰写文稿	4
2.1 分级标题和段落	4
2.2 枚举和序号	4
2.2.1 有序枚举	4
2.2.2 无序枚举	5
2.3 公式	5
2.3.1 行内公式	5
2.3.2 行间公式	5
2.4 插入图片	5
2.5 插入代码	6
2.6 参考文献与交叉引用	6
2.6.1 交叉引用	6
2.6.2 参考文献	6
2.6.3 如何找文献的 BibTeX 引用信息	6
3 报告结构与内容组织（参考）	7
3.1 实验报告类	7
3.2 文献调研类	10
3.3 实验报告（科技论文）写作注意事项	10
3.3.1 实验环境	10
3.3.2 实验设计	10
3.3.3 实验结果呈现	11
3.3.4 实验结果分析	11
4 体系结构相关实验分析——矩阵与向量内积	13
4.1 实验介绍	13
4.2 实验设计指导	13
4.2.1 题目分析	13
4.2.2 算法设计与编程	13
4.3 程序编译和运行	14
5 体系结构相关实验分析——n 个数求和	14
5.1 实验介绍	14
5.2 实验设计指导	14
5.2.1 题目分析	14
5.2.2 算法设计与编程	15
5.3 程序编译、运行、结果分析及更多思考	15

6 使用 VTune 剖析程序性能	15
6.1 VTune 简要介绍	16
6.2 VTune 的安装及使用	16
6.2.1 VTune 安装	16
6.2.2 VTune 使用	16
6.3 体系结构相关的程序性能剖析	17
6.3.1 cache 命中率	17
6.3.2 超标量	18
7 使用 Perf 剖析程序性能	21
7.1 Perf 简要介绍	21
7.2 Perf 的安装及使用	21
7.2.1 Perf 安装	21
7.2.2 Perf 使用	21
7.3 体系结构相关的程序性能剖析	23
7.3.1 cache 命中率	23
7.3.2 超标量	24
8 使用 Godbolt 分析汇编程序	26

1 关于 AI 的使用

最近，我们学校也部署好了 deekseek。关于大模型的使用一直饱受争议，我们在此建议同学们，关于算法设计和性能分析部分，大家尽可能多思考，避免使用 AI，不得不的时候也请标注一下。如果你的报告中关键章节大幅使用 AI 生成，可能会影响你的分数（大家可以通过 AI 了解一些 API 的使用，然后灵活运用到算法里，这是没有问题的）。

2 使用 LaTeX 撰写文稿

我们会为同学们提供几份 latex 的模板压缩包，其中包含了大部分撰写报告时需要的使用案例（分段、插入图片、插入代码、参考文献和引用等等），使用 overleaf 进行编译时记住选择 XeLaTeX 编译器。同学们可以根据我们提供的简单模板和查阅资料多进行尝试，观察编译效果，总之是熟能生巧。

下面简单介绍几种常用的排版方法。

2.1 分级标题和段落

一般用到的标题和段落组织方式非常简单，如下所示：

```
\section{一级标题}
\subsection{二级标题}
\subsubsection{三级标题}
\paragraph{一级段落} 并行程序设计
\subparagraph{二级段落} SIMD...
```

(a) latex 代码

```
1 一级标题
1.1 二级标题
1.1.1 三级标题
一级段落 并行程序设计
二级段落 SIMD...
```

(b) 排版效果

2.2 枚举和序号

文章中分点、分步骤陈述时需要用到枚举和序号。

2.2.1 有序枚举

```
\begin{enumerate}
\item **
\item **
\item **
\end{enumerate}
```

(c) latex 代码

```
1. **
2. **
3. **
```

(d) 排版效果

2.2.2 无序枚举

```
\begin{itemize}
  \item **
  \item **
  \item **
\end{itemize}
```

(e) latex 代码

- **
- **
- **

(f) 排版效果

当然除了以上还有许多样式的枚举，可以通过修改一些参数实现，感兴趣的同学们可以查阅资料进行尝试和了解。灵活的运用枚举，可以让大家的文稿看起来条理清晰，结构美观，同学们可以在后续的编写中自行体会！

2.3 公式

2.3.1 行内公式

一般用于公式或者特殊变量嵌入在文字中的情况，用成对的单 \$ 符号限定范围，限定为公式的部分会以特殊的字体突出显示。如函数 $f(x) = a + b$

2.3.2 行间公式

一般用于公式需要单行突出的情况，用成对的 \$\$ 符号限定范围，限定为公式的部分会被单独另起一行，单行突出显示。如函数

$$f(x) = a + b$$

如果公式需要全文编号，或者需要输入一些复杂的公式，可以使用 equation 包，用 begin 和 end 限定范围。如函数

```
\begin{equation}
  f(x)=a+b
\end{equation}
```

将被渲染为

$$f(x) = a + b \quad (1)$$

注意 latex 中很多字符有特殊意义，不能直接使用，使用特殊字符时一般需要加反斜杠进行转义。这里提供一个[latex 特殊符号汇集](#)。

另外，在这里介绍一个非常好用的公式编辑工具：[latex 公式编辑器（在线版）](#)。可以手动编辑或者图片识别，并且图片识别的准确率很高，非常好用。

2.4 插入图片

可以插入单张图片 (figure)，也可以插入多张图片并共用一个 caption (subfigure)。使用的代码非常固定，参考给出的模板以及[这个链接](#)使用即可。

小提醒，注意插入图片时设置的排版位置参数（方括号中的）（试试 H 参数有什么效果?），如果插入图片以后图片不见或者乱跑一般是这个参数出了问题。

2.5 插入代码

分为代码（源码）和伪代码，模板中都有，参照使用即可。

2.6 参考文献与交叉引用

模板中已有参考文献和引用样例，可以参照使用，这里再做详细一些的使用说明。

2.6.1 交叉引用

在 LaTeX 中，`label` 和 `ref` 是用于交叉引用的两个关键命令，它们使得在文档中引用图表、章节、公式等元素变得更加方便。

`label` 命令通常用于为某个元素（如章节、图表、公式等）添加一个标签，以便后续可以通过引用这个标签来引用该元素。而 `ref` 命令用于在文档中引用具有 `label` 标签的元素。二者一般语法分别为：`\label` 标签名与 `\ref` 标签名。如引用图 2.1，无论后续图片的顺序如何变化，均能保证正确引用对应 `label` 的图片标号。表格、公式、伪代码等同理。

具体到编译出的 PDF 文件，大家可以通过点击引用来灵活的跳转，这是一般科技论文写作的规范，希望同学们熟练使用！

2.6.2 参考文献

模板中采用 BibTeX 的方式建立参考文献数据库，引用的时候调用所需要的参考文献。BibTeX 文件的后缀名为 `.bib`，对应模板中即为 `references.bib` 文件，该文件中会规范地列出引用文献的信息，引用时只需在引用位置 `\cite{ 引用文章名称 }`（文章名称在 `.bib` 文件对应文献的第一个参数中规定）。

2.6.3 如何找文献的 BibTeX 引用信息

使用 [百度学术](#)（有条件的同学也可以使用 Google 学术），搜索你需要引用的文献，点击“引用”：



图 2.1: 百度学术搜索界面

选择 BibTeX：

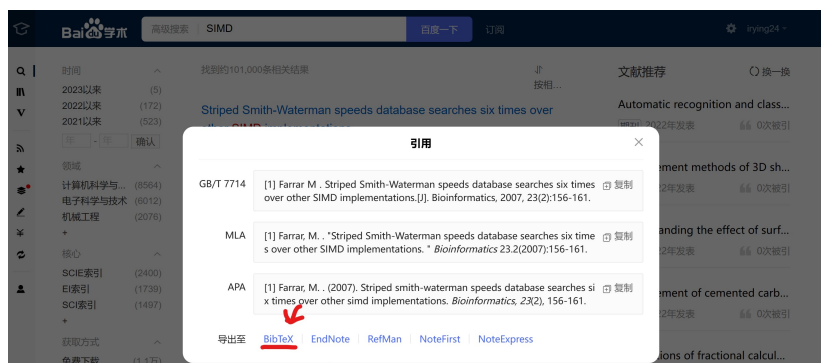


图 2.2: 百度学术搜索界面

选择后页面自动跳转到含有 bibtex 格式的引用信息的新页面:

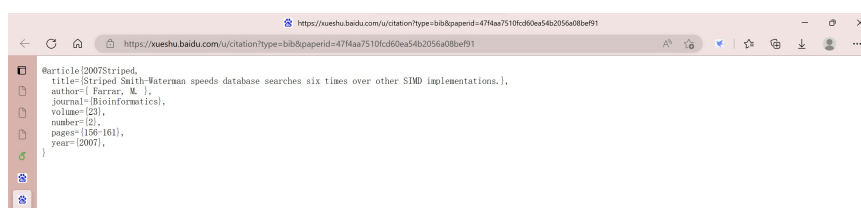


图 2.3: 百度学术搜索界面

复制内容粘贴到.bib 即可。

3 报告结构与内容组织（参考）

3.1 实验报告类

完成一份合格的实验报告，需要做到逻辑结构、条例清晰，内容详细且重点突出。特别是对于这门课程，每次实验都包括基础部分和拓展选做两部分，并且支持同学们自由选题，这就意味着每位同学的实验内容和实验完成的程度都不相同。如何在实验报告中体现出你所学、所做实验内容的完整性（证明你圆满完成实验的基础内容）和独特性（你所做的深入探究），如何充实和组织你的报告内容，非常重要。

在介绍前先做说明，以下总结并呈现的内容仅为个人经验，并且也是在学习过程中不断改进的，所以目的只是为之前没有写过较长篇幅实验报告的同学提供一个抓手，仅作为参考，还是希望同学们可以在实践过程中形成自己的观点和思考，并不断改进。

结构组织 对于这门课来说，每一个平时实验老师都会提供一份非常详细的实验内容和实验要求。实验的要求一般分为**基础要求**和**进阶要求**两部分，实验内容因人而异，可能不会在作业要求中明确列出，以高斯消元选题为例，如果你同时实现了基础消元和特殊消元，那么你的内容就可以分为这样的两部分。

了解实验要求后，报告结构一般可以按实验内容或实验要求为主干展开（选择一个比较明确的）。以本次实验为例，实验内容的分类比较明确，可以按内容展开：

目录	
一、实验一：n*n 矩阵与向量内积	2
(一) 算法设计	2
1. 平凡算法设计思路	2
2. cache 优化算法设计思路	2
(二) 编程实现	2
1. 平凡算法	2
2. cache 优化算法	2
(三) 性能测试	2
1. 平凡算法	2
2. cache 优化算法	2
(四) profiling	2
1. 平凡算法	2
2. cache 优化算法	2
(五) 结果分析	2
二、实验二：n 个数求和	2
(一) 算法设计	2
1. 平凡算法设计思路	2
2. cache 优化算法设计思路	2
(二) 编程实现	2
1. 平凡算法	2
2. cache 优化算法	2
(三) 性能测试	2
1. 平凡算法	2
2. cache 优化算法	2
(四) profiling	2
1. 平凡算法	2
2. cache 优化算法	2
(五) 结果分析	2
三、实验总结和思考	2
(一) 对比 2 个实验的异同	2
(二) 总结	2

图 3.4: 按实验内容组织报告结构

做了实验拓展部分的同学，写报告也可以按“基础要求—进阶要求”实验难度递进的方式展开：

目录	
一、 基础要求	1
(一) 算法设计	1
1. $n \times n$ 矩阵与向量内积	1
2. n 个数求和	1
(二) 编程实现	1
1. $n \times n$ 矩阵与向量内积	1
2. n 个数求和	1
(三) 性能测试	1
1. $n \times n$ 矩阵与向量内积	1
2. n 个数求和	1
(四) profiling	1
1. $n \times n$ 矩阵与向量内积	1
2. n 个数求和	2
(五) 结果分析	2
1. $n \times n$ 矩阵与向量内积	2
2. n 个数求和	2
二、 进阶要求	2
(一) 循环展开优化	2
1. 算法设计	2
2. 性能测试	2
3. profiling	2
4. 与原算法对比	2
(二) ...	2
三、 实验总结和思考	2
(一) 异同对比	2
(二) 总结和思考	2

图 3.5: 按实验难度递进组织报告结构

注意每个实验内还是要分别按两种算法（平凡算法和 cache 优化算法）展开。

内容组织

摘要 最好有摘要部分，简要介绍实验的内容、经过和结果。

实验平台信息 由于本课程的实验性能测试结果和实验平台（你实验所用的计算机）密切相关，所以在介绍实验之前，需要提供实验平台的详细信息。

报告各部分内容组织 在上一段的例子中已经展示了本课程实验报告最基础的结构组织，下面详细说明各部分的内容组织。

1. **算法设计** 对实验内容的实现思路进行文字性的描述，也可以用伪代码进行辅助描述。
2. **编程实现** 包含实验内容的具体实现（代码级）。注意如果代码部分太长，最好不要大篇幅粘贴代码，可以省去这个部分并在算法设计部分用详细的伪代码进行描述。
3. **性能测试** 包含你实现的代码在不同平台、不同编译选项下的性能测试结果。最好给予测试方法的说明（可靠性说明），测试数据制成表格或图的形式呈现（前者呈现实验结果具体数值，后者展现实验结果变化趋势），注意图的大小和位置。
4. **profiling** 使用性能分析工具进行更细粒度的代码性能剖析，通过 cache 表现（不同层次的 miss 和 hit 率）、代码编译结果、运行时时间片占用结果等等进行结果与性能表现间关系的分析。
5. **结果分析** 总结性能测试和 profiling 的结果，一般能够用 profiling 的结果去解释性能测试的结果，并提出自己的想法和思考。

3.2 文献调研类

文献调研类报告也可以参考上面介绍的根据内容展开的组织结构，具体内容根据同学们自己调研的结果而定。这类报告的撰写格式没有非常刻板的要求，同学们可以根据自己想调研的内容，选择最合适的结构组织方式进行报告的撰写。不过这里可以提供一种比较好上手的内容组织方式（调研思路也可以按此展开）。

按时间线展开 按时间顺序组织文章结构是比较简单，且容易使文章看起来结构鲜明、层次清晰的方式。按照时间顺序，可以调研一种事物的历史、现状以及大众对该事物未来发展的看法，也可以调研一种事物的发展和演变过程等等。以调研**超算**为例，文章结构可以按以下组织：

目录	
一、超算的历史	1
(一) 超算的发展历史概述	1
1. 超算的 x 个发展阶段	1
二、超算的现状	1
(一) 超算 x	1
1. 性能	1
2. 硬件特点	1
3. 软件特点	1
4. ...	1
(二) 超算 y	1
1. 性能	1
2. 硬件特点	1
3. 软件特点	1
4. ...	1
(三) ...	1
三、超算的未来	1

图 3.6: 按时间顺序组织调研报告结构

3.3 实验报告（科技论文）写作注意事项

如无特殊情况，对于要求提交报告的作业，同学们应提交 pdf 文档，这样会节省老师、助教很多时间（在长江雨课堂可直接预览批改，而 word、压缩文件等格式还需下载）。同样地，文件名中最好加上学号和姓名，以节省老师助教修改文件名的时间。

在技术报告/科技论文中，实验部分的撰写需注意：

3.3.1 实验环境

实验部分应清楚地描述进行实验的软硬件平台：包括硬件系统的参数（CPU（型号、核数、主频）、内存容量、外存型号容量（如涉及外存访问）、网络（如涉及网络传输）...），操作系统及版本，上层系统及版本（如涉及的话，如数据库系统等），编译器及版本、编译选项（特别是优化力度）等等。对实验环境的说明应基于实验的需求，如 4 节中，还需额外补充 CPU 各级 cache 大小。

3.3.2 实验设计

在实验设计部分，需要清晰地交代实验使用的数据（测试用例）的来源（公开数据集/自己生成/仿真等等）、规模、数据集特征等信息，实验的性能指标（时间、RMSE、AUC 等等，根据不同的问题自

行设置)，用于比较的方法（baseline）及本文提出的方法，并最好简单介绍本文方法与 baseline 的异同及优势所在。

总之，**实验的目的是为了说明本文提出的方法有好的性能**，应围绕这一目的设计实验，明确每个实验想要说明什么，如 A 方法比 B 方法速度快（测试不同问题规模下两种方法的时间）、A 方法加入算法策略 X 后可加速（测试不同问题规模下 A 方法和 A 方法 + 策略 X 的时间），再如对于本问题设置对应各级 cache 大小的问题规模来研究 cache 对性能的影响等。

3.3.3 实验结果呈现

实验结果应通过尽可能清晰的方式呈现，不要用截屏数据、测试工具截图等方式呈现实验结果，要收集实验结果数据，再绘制图、表格来呈现。用 excel 等工具绘制的图不要保存为 jpg 等图片格式插入到 pdf、word 文档中，要用 pdf 打印机（如微软的）保存为 pdf 文件插入到 latex 中，或直接拷贝绘图对象插入到 word 文档中。除了基本的 excel 制图外，如果希望绘制更为专业的图像，可以借助/通过 Origin, Python, R 等工具/方式绘制。图、表中要清晰描述参数及单位，图、表标题、坐标轴等信息要准确、简明。如图 3.7 即为一张不标准的图示，相同形式而较为完整的图应参考 3.8。

图的表现形式应是依照文章想要表述或强调的内容而变化的，即使是相同的数据，也会存在不同的图像表达形式。比如 3.9 相较 3.8，虽然同样包含折线图，但后者额外加入了数据点标识，并将图例移至图像框外。这样做可以更清晰地表现出实验图的绘制实际上使用了多少数据点，同时也保证数据不被图例遮挡。而 3.10 相较图 3.7-3.9，曲线下的额外的填充则让图像表达的重心从曲线变化转变到分布上（注意，此处主要为强调数据应根据其特征，及作者的表达重心选择合适的绘制方式，而即使是相同的数据也会产生不同的表达效果。故，此处用以示例的图像，其数据的表达方式的并不一定总是选取了最恰当的。如图 3.10，对于时间-速度数据，此处类似于分布的展示并非是足够合适的。）进一步地，图 3.11 相较 3.10，额外绘制了不同 Object 过程中达到的最高速度，相较前述图像，其强调重心是存在差异的。而对于 3.12，则通过小框放大部分数据的方式，将重心放在了部分希望额外强调的数据上，同时减少了对其他数据的绘制笔墨（只绘制了折线图，没有绘制对应的数据点）。

3.3.4 实验结果分析

在科技论文中，实验结果的分析是非常重要的。

分析实验结果时要做到“**看图说话**”。即，对图、表中的实验结果进行描述。如不同方法的性能对比、文章提出的新方法较之 baseline 性能提升幅度 xx%、随着问题规模变化性能变化趋势、最高性能最低性能是多少、一些算法策略对性能影响幅度等等。

更重要的，需要**对实验结果进行合理分析解释**，如为什么 A 方法比 B 方法好、为什么加入策略 X 后 A 方法性能有明显提升、为什么随着问题规模增大 A 方法时间增加幅度比 B 方法慢等等。从算法设计和计算机硬件的角度，对产生这样的实验结果的原因进行**理论分析**。另外，采用 profiling 工具分析程序运行过程中硬件、系统软件的一些监测值来说明产生程序性能的**底层原因**，也是一种好的方法。最好能与算法设计和分析小节中对算法的理论分析呼应上，这样，理论和实验相互印证会有更强的说服力。总之，实验结果的分析一般是为了说明本文提出方法的确有好的性能，并理论与理论分析呼应等方式增强说服力。

另外，如作业中所描述，可以尝试不同的算法设计和实验方式。

最后需要说明的是，希望同学们是在认真做完实验后，满腹收获与心得时，水到渠成地完成每一份实验报告，不要一味地复制粘贴不属于自己的内容来拼凑字数。实验报告只是一份你实验成果的展示，完成好这份报告确实重要，但是更重要、能让你收获更多的一定是认真踏实地学习和实践的过程。

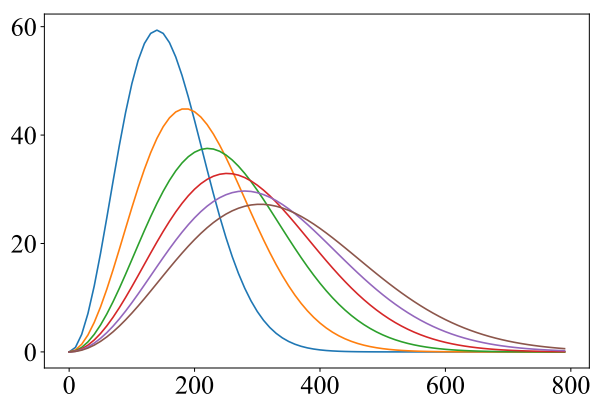


图 3.7: 不规范的实验图

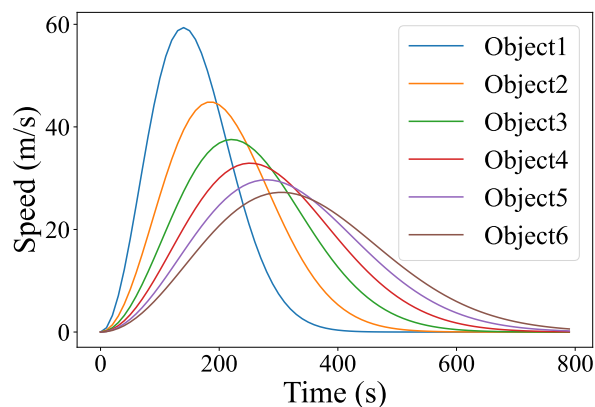


图 3.8: 较为完整的实验图

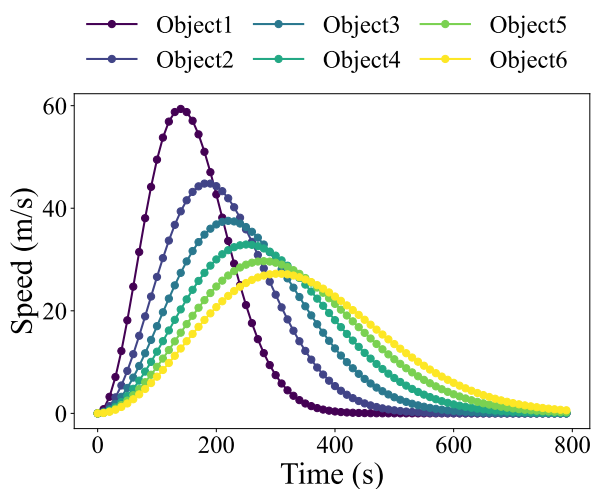


图 3.9: 加入散点、移出图例的实验图

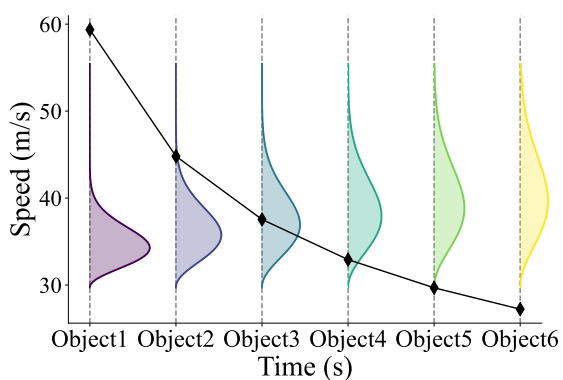


图 3.11: 折线表示不同 Object 的最大速度, 每条垂直线表示不同 Object 各自的速度变化情况

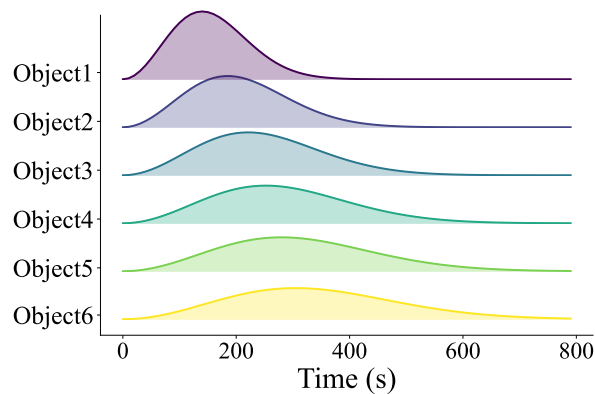


图 3.10: 区分不同 Object 数据分布的实验图

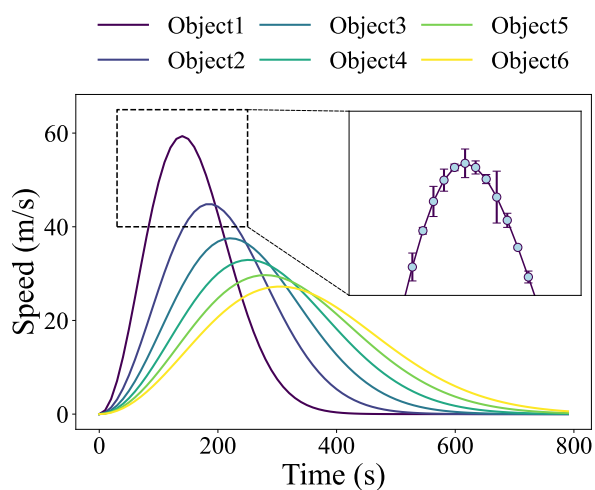


图 3.12: 强调部分数据的实验图

4 体系结构相关实验分析——矩阵与向量内积

4.1 实验介绍

给定一个 $n \times n$ 矩阵，计算每一列与给定向量的内积，考虑两种算法设计思路：逐列访问元素的平凡算法和 **cache 优化算法**，进行实验对比：

1. 对两种思路的算法编程实现；
2. 练习使用高精度计时测试程序执行时间，比较平凡算法和优化算法的性能。

4.2 实验设计指导

4.2.1 题目分析

本题与讲义中矩阵列求和的例子非常相似，都是关于如何设计算法，令访存模式具有更好的空间局部性，从而发挥 cache 的能力。

平凡算法逐列访问矩阵元素，一步外层循环（内存循环一次完整执行）计算出一个内积结果；cache 优化算法则改为逐行访问矩阵元素，一步外层循环计算不出任何一个内积，只是向每个内积累加一个乘法结果。后者的访存模式具有很好空间局部性，令 cache 的作用得以发挥。

需要注意以下几点：

1. 实验设计（问题规模的确定、测试数据的生成等）

对于本问题，测试数据人为设定固定值即可，例如 $A[i][j] = i + j$ ，方便程序正确性检查。对本题而言，程序性能与 cache 相关，而现代 CPU cache 有多个层次，每个层次有固定规模（结合体系结构调研作业）。因此，观察问题规模与 cache 大小不同关系时的程序性能变化，是有意义的。性能测试中可设置一系列不同的问题规模，规模可对应于实验 CPU 设备各级 cache 大小，以研究问题规模变化对程序性能产生的影响及变化趋势。并且，我们可以创建一定大小的数组来占据内存空间，来进一步观察更小的内存对程序性能产生的影响。

2. 程序分析

一方面，可对算法进行理论分析（包括程序性能随问题规模变化的趋势，以及问题达到与各级 cache 大小相对应规模时程序性能变化等），计算时间复杂度及空间复杂度。此外，这里除了考虑基本的计算次数外，还需额外考虑程序的访存开销。

另一方面，可采用 VTune 等工具获取程序运行时一些系统层面的指标（如 cache 命中率、缺页次数等），揭示性能表现的内在原因，同时也可与理论分析对照。VTune 的使用见 6 节。

3. 程序测试（运行时间测量）

参考《实验教学指导书-Lab0》中的程序运行时间测量方法。由于本问题计算较为简单，当矩阵规模较小时，程序运行时间很短。因此，可采用重复运行待测函数、延长计时间隔的方法，来解决计时函数精度不够、影响测量精度的问题。

4.2.2 算法设计与编程

编程思路与讲义中矩阵列求和的例子几乎一致：

● 平凡算法

```
1 // 逐列访问矩阵元素：一步外层循环（内存循环一次完整执行）计算出一个内积结果
2 for(i = 0; i < n; i++){
3     sum[i] = 0.0;
4     for(j = 0; j < n; j++)
```

```
5     sum[i] += b[j][i] * a[j];  
6 }
```

• 优化算法

```
1 // 改为逐行访问矩阵元素：一步外层循环计算不出任何一个内积，只是向每个内积累加一个乘法结果  
2 for(i = 0; i < n; i++)  
3     sum[i] = 0.0;  
4 for(j = 0; j < n; j++)  
5     for(i = 0; i < n; i++)  
6         sum[i] += b[j][i] * a[j];
```

后者的访存模式与行主存储匹配，具有很好空间局部性，令 cache 作用得以发挥。

4.3 程序编译和运行

程序的编译、运行和结果查看可参考《实验教学指导书-Lab0》。在本实验中，我们都是进行本地并行编程练习，不涉及华为鲲鹏服务器。

5 体系结构相关实验分析—— n 个数求和

5.1 实验介绍

计算 n 个数的和，考虑两种算法设计思路：逐个累加的平凡算法（链式）；适合**超标量**架构的指令级并行算法（相邻指令无依赖），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。完成如下作业：

1. 对两种算法思路编程实现；
2. 练习使用高精度计时测试程序执行时间，比较平凡算法和优化算法的性能。

5.2 实验设计指导

5.2.1 题目分析

本题需要关注以下几点：

1. **测试数据的生成**。同上一个题目一样，数据生成人为指定即可，元素个数 n 取 2 的幂即可，方便递归算法设计。
2. **循环处理**。几个算法基本实现方式都是采用循环，但可能带来严重的额外开销——每个元素只进行一次加法，但却需要进行循环判定、归纳变量递增等多个额外操作。可采用循环展开（unroll）策略——每个循环步进行多次加法运算，相当于将多个循环步的工作展开到一个循环步，从而大幅度降低簿记操作的比例。甚至可以采用宏/模板将循环完全去掉。但要注意，不同算法尽量保持相同的展开比例，保证性能对比的公平性。循环展开可结合指令级并行，即，合并到一个循环步中的多个计算通过合理设计令它们相互不依赖，可同时由多条流水线处理。
3. **中间结果处理**。递归算法每个步骤会得到大量中间结果，可在输入数组中原地保存（输入的元素不再被使用的话），也可分配一个辅助数组保存。元素访问顺序要小心设计，注意空间局部性。
4. **问题规模（元素个数）设置**。同样可考虑流水线条数、cache 大小等系统参数来设置实验中的问题规模。
5. **较小问题规模执行**。当问题规模较小时执行时间可能很短，可将核心计算重复多次，以提高性能测试和 profiling 的精度，如上一题。

5.2.2 算法设计与编程

• 平凡算法

```
1 // 链式：将给定元素依次累加到结果变量即可
2 for (i = 0; i < n; i++)
3     sum += a[i];
```

• 优化算法

```
1 // 多链路式
2 sum1 = 0; sum2 = 0
3 for (i = 0; i < n; i += 2) {
4     sum1 += a[i];
5     sum2 += a[i + 1];
6 }
7 sum = sum1 + sum2;
8
9 // 递归：
10 1. 将给定元素两两相加，得到n/2个中间结果；
11 2. 将上一步得到的中间结果两两相加，得到n/4个中间结果；
12 3. 依此类推，log(n)个步骤后得到一个值即为最终结果。
13
14 // 实现方式1：递归函数，优点是简单，缺点是递归函数调用开销较大
15 function recursion(n)
16 {
17     if (n == 1)
18         return;
19     else
20     {
21         for (i = 0; i < n / 2; i++)
22             a[i] += a[n - i - 1];
23         n = n / 2;
24         recursion(n);
25     }
26 }
27
28 // 实现方式2：二重循环
29 for (m = n; m > 1; m /= 2) // log(n)个步骤
30     for (i = 0; i < m / 2; i++)
31         a[i] = a[i * 2] + a[i * 2 + 1] // 相邻元素相加连续存储到数组最前面
32 // a[0]为最终结果
```

5.3 程序编译、运行、结果分析及更多思考

一个可以探索的问题，如果是进行浮点运算，计算次序的改变可能会导致结果变化（计算机表示浮点数精度有限导致），而本问题的指令级并行算法与串行算法中元素累加顺序是不同的，可对此进行探索。

6 使用 VTune 剖析程序性能

注意：VTune 主要针对 Intel CPU 设计，使用 AMD 或其他 CPU 的设备很难发挥 VTune 的完整功能，非 Intel CPU 机器建议优先尝试其他的性能测试工具。

6.1 VTune 简要介绍

VTune 是 Intel 推出的一款可视化的性能剖析 (profiling) 工具, 支持的平台包括: Windows, Linux 和 macOS。本实验展示的是在 Windows 系统上的应用, 在 Linux 上的安装和使用可参考[官方文件](#) (如安装 oneAPI 开发套件, 其中也包含了 VTune)。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。

6.2 VTune 的安装及使用

6.2.1 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。

6.2.2 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, 同时尽量关掉电脑的各种病毒防护和防火墙, 否则 VTune 可能无法正常工作。

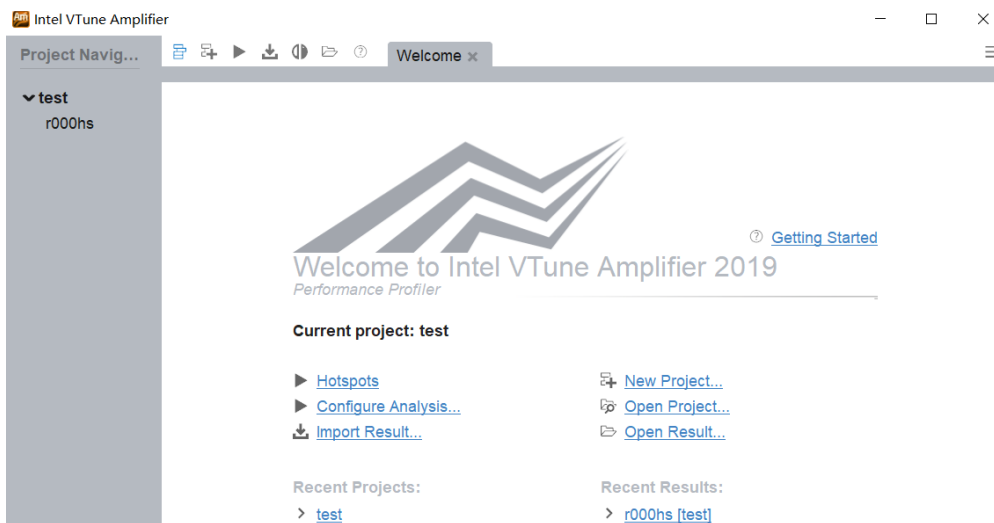


图 6.13: VTune 主页面

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。

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 Platform 数据。如图 6.16所示, Summary 主要分析的数据有: 执行时间, 高热热点部分, 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。

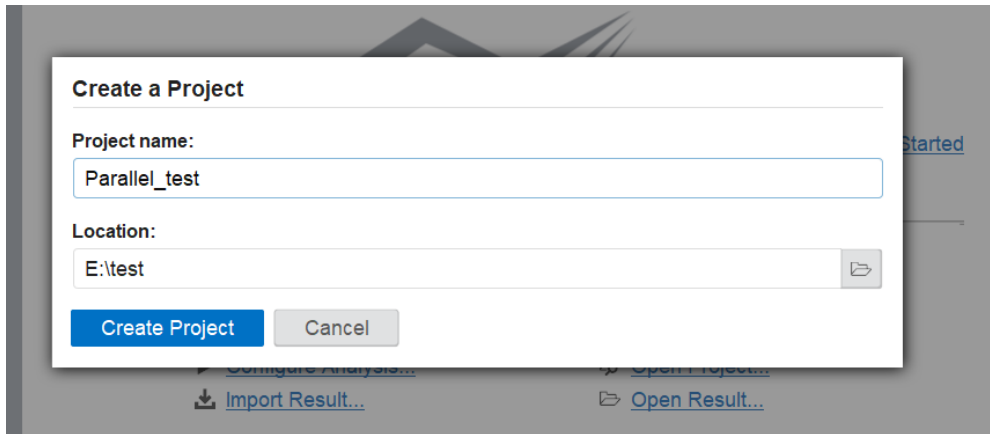


图 6.14: 创建项目

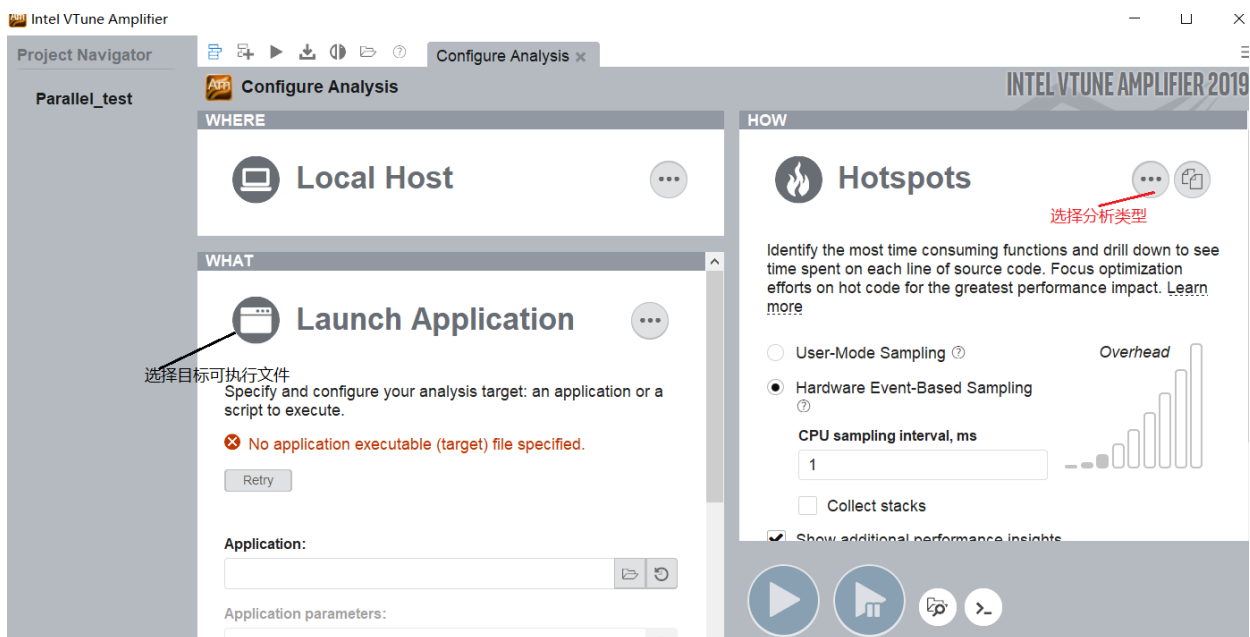


图 6.15: 配置分析项目

6.3 体系结构相关的程序性能剖析

6.3.1 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。

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 6.17，在 Hotspots 窗口中选择“Hardware Event-Based Sampling”。然后编辑希望采样的事件，如图 6.18所示，在 Events configured for CPU 中勾选 MEM_LOAD_RETIURED.L1_HIT、MEM_LOAD_RETIURED.L1

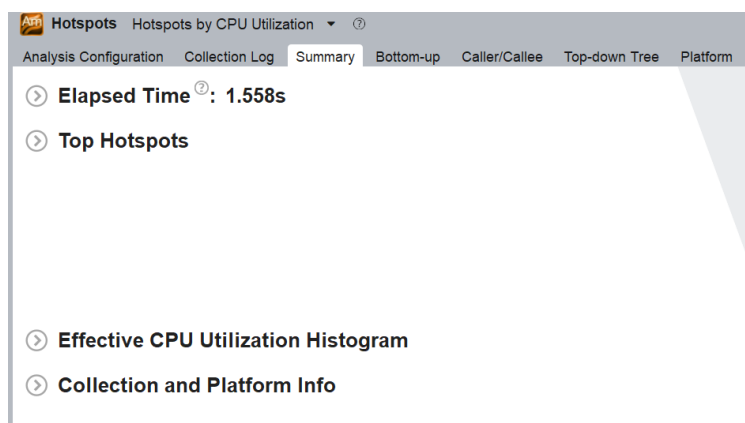


图 6.16: Hotspot 简单分析结果

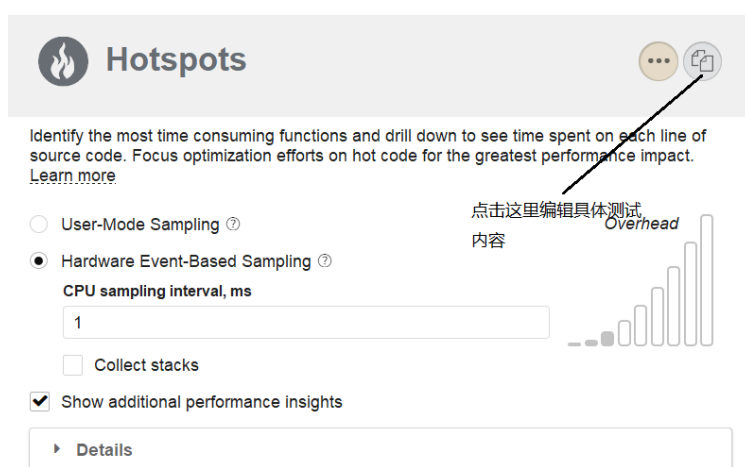


图 6.17: 选择硬件事件采集

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 6.19给出了测试结果。

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 6.20所示。

6.3.2 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。

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

Event Name	Sample ...	Description
<input checked="" type="checkbox"/> MEM_LOAD_RETIRED.L1...	2000003	Retired load instructions with L1 ca...
<input checked="" type="checkbox"/> MEM_LOAD_RETIRED.L1...	100003	Retired load instructions missed L1 ...
<input checked="" type="checkbox"/> MEM_LOAD_RETIRED.L2...	100003	Retired load instructions with L2 ca...
<input checked="" type="checkbox"/> MEM_LOAD_RETIRED.L2...	50021	Retired load instructions missed L2 ...
<input checked="" type="checkbox"/> MEM_LOAD_RETIRED.L3...	50021	Retired load instructions with L3 ca...
<input checked="" type="checkbox"/> MEM_LOAD_RETIRED.L3...	100007	Retired load instructions missed L3 ...
<input type="checkbox"/> MEM_LOAD_RETIRED.F...	100007	Retired load instructions which data...
<input type="checkbox"/> MEM_LOAD_RETIRED.F...	100003	Retired load instructions which data...
<input type="checkbox"/> MEM_LOAD_RETIRED.L1...	2000003	Retired load instructions with L1 ca...
<input type="checkbox"/> MEM_LOAD_RETIRED.L1...	100003	Retired load instructions missed L1 ...
<input type="checkbox"/> MEM_LOAD_RETIRED.L2...	100003	Retired load instructions with L2 ca...
<input type="checkbox"/> MEM_LOAD_RETIRED.L2...	50021	Retired load instructions missed L2 ...

图 6.18: 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 6.19: 列主次序访问算法的分析结果

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。

如图 6.21 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 6.22 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

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 6.20: 行主次序访问算法的分析结果

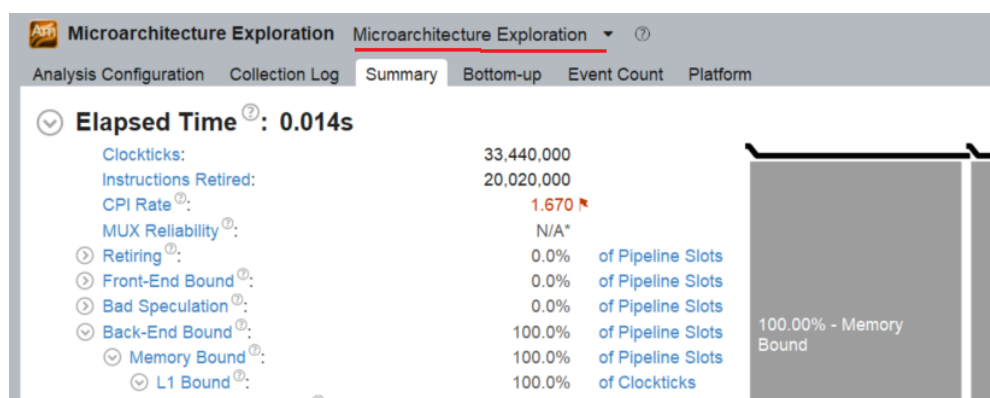


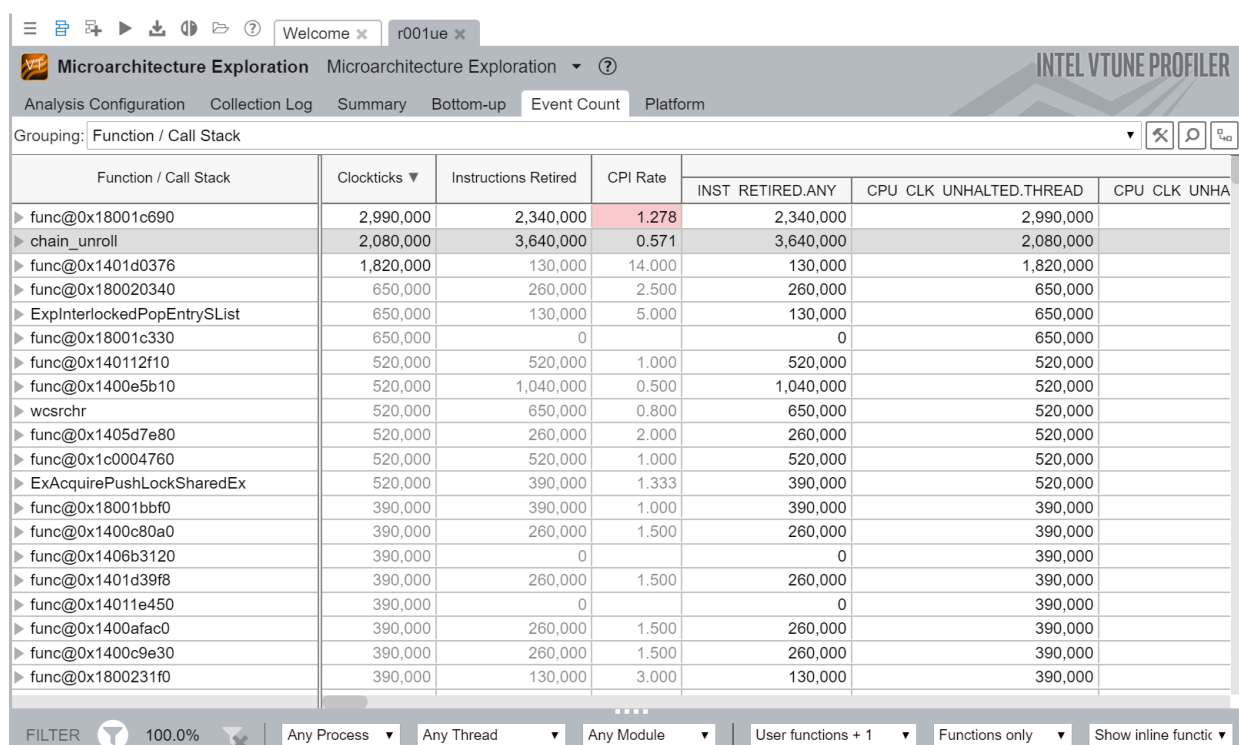
图 6.21: 超标量分析结果

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。

对比两路链式算法分析结果，如图 6.23 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 6.23 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：

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。



Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate	INSTRUMENTED		
				INST RETIRED.ANY	CPU CLK UNHALTED.THREAD	CPU CLK UNHA
func@0x18001c690	2,990,000	2,340,000	1.278	2,340,000	2,990,000	
chain_unroll	2,080,000	3,640,000	0.571	3,640,000	2,080,000	
func@0x1401d0376	1,820,000	130,000	14.000	130,000	1,820,000	
func@0x180020340	650,000	260,000	2.500	260,000	650,000	
ExpInterlockedPopEntrySList	650,000	130,000	5.000	130,000	650,000	
func@0x18001c330	650,000	0		0	650,000	
func@0x140112f10	520,000	520,000	1.000	520,000	520,000	
func@0x1400e5b10	520,000	1,040,000	0.500	1,040,000	520,000	
wcsrchr	520,000	650,000	0.800	650,000	520,000	
func@0x1405d7e80	520,000	260,000	2.000	260,000	520,000	
func@0x1c0004760	520,000	520,000	1.000	520,000	520,000	
ExAcquirePushLockSharedEx	520,000	390,000	1.333	390,000	520,000	
func@0x18001bbf0	390,000	390,000	1.000	390,000	390,000	
func@0x1400c80a0	390,000	260,000	1.500	260,000	390,000	
func@0x1406b3120	390,000	0		0	390,000	
func@0x1401d39f8	390,000	260,000	1.500	260,000	390,000	
func@0x14011e450	390,000	0		0	390,000	
func@0x1400afac0	390,000	260,000	1.500	260,000	390,000	
func@0x1400c9e30	390,000	260,000	1.500	260,000	390,000	
func@0x1800231f0	390,000	130,000	3.000	130,000	390,000	

图 6.22: 链式算法详细结果

7 使用 Perf 剖析程序性能

7.1 Perf 简要介绍

Perf(Performance Events for Linux) 是一个用于基于 Linux 2.6+ 的系统的分析器工具，它抽象出 Linux 性能测量中的 CPU 硬件差异，并提供了一个简单的命令行界面。该工具基于 Linux 内核提供的 `perf_events` 接口实现。

下文中，当命令行以 \$ 开头时，表示以非 root 用户执行，以 # 开头时，表示需要 root 权限。[关于 Ubuntu 中的 root 用户](#)。

7.2 Perf 的安装及使用

7.2.1 Perf 安装

以 Ubuntu 20.04 为例，在 linux 系统中可以使用以下命令安装 perf 工具，如果下载速度过慢，可以考虑更换国内软件源，可参见[镜像源帮助页](#)。

```
sudo apt install linux-tools-$(uname -r) linux-tools-generic
```

注意，若 linux 系统安装在虚拟机中，需要打开虚拟机的“虚拟化 CPU 性能计数器”或“虚拟化 PMU”功能，否则 perf 将不支持硬件事件的采样。目前发现，VirtualBox、WSL2 暂不支持类似功能。

7.2.2 Perf 使用

使用 perf 对程序性能进行剖析，主要借助对事件的收集测量。事件分软件事件和硬件事件两类，软件事件主要包括上下文切换等 linux 内核提供的事件，硬件事件主要包括周期数、指令数、缓存未命

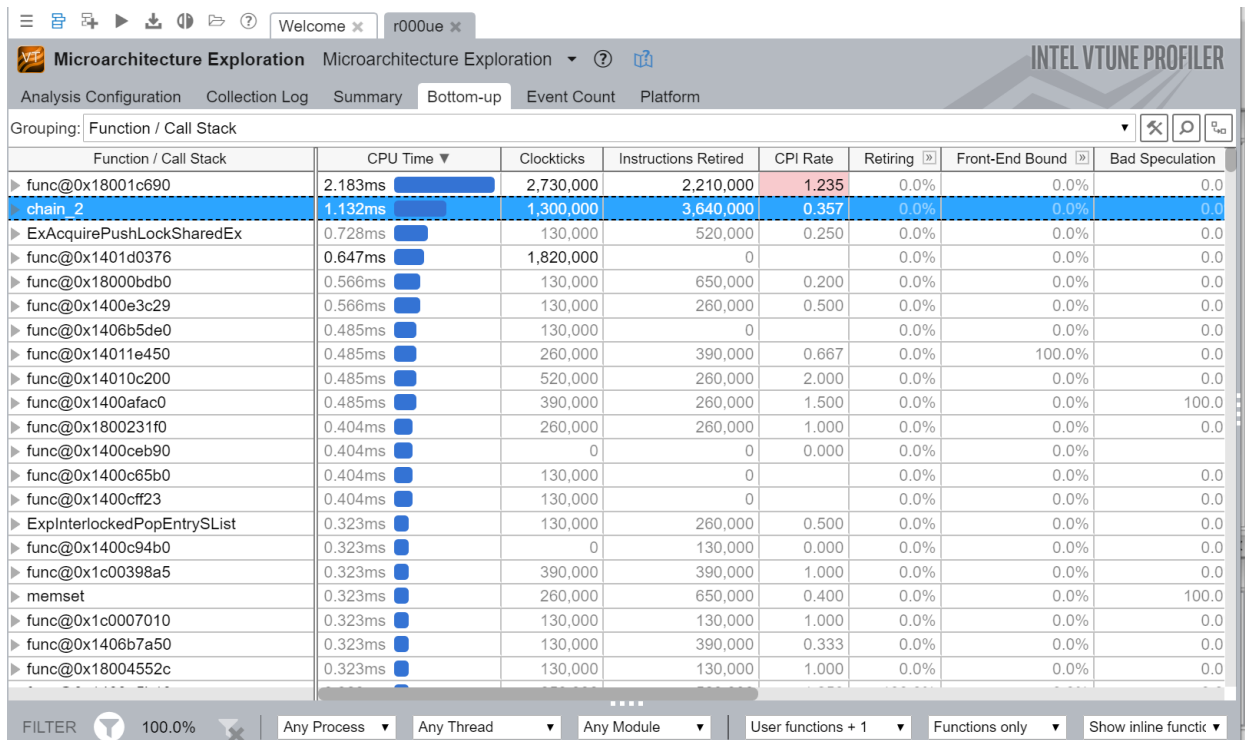


图 6.23: 两路链式算法详细结果

中等体系结构相关事件，需要依赖硬件单元 PMU（Performance Monitoring Unit）。

可使用list命令查看在你的系统上有哪些事件，下面举几个例子。

```
perf list -h          #列出帮助
perf list hw          #列出硬件事件
perf list cache        #列出L1 cache相关事件
perf list pmu          #列出pmu相关事件
perf list l2_cache     #列出L2 cache相关事件
perf list pmu | grep cache #过滤包含"cache"的行
```

perf 提供了一系列命令来跟踪程序和分析性能，其中重点介绍stat, record, report。

Perf 有两种模式：

1. Counting 模式下（例如使用perf stat命令）会数出某事件在一段时间内一共发生了多少次
2. Sampling 模式下（例如使用perf record命令），在计数一定数量的时间后产生采样中断

perf stat对程序运行的事件数进行统计，给出最终结果，默认会对周期数，指令数等几项进行统计。可以使用-e <event1>,<event2>...选项指定对其他事件的统计，使用-r n选项进行 n 次重复统计，输入perf stat -h查看帮助。

perf record通过采样对程序运行信息进行收集，结果默认保存在当前目录的perf.data文件中。使用选项-e <event>指定收集的事件，使用-g记录运行过程中函数调用关系，使用-F <freq>指定采样的频率，使用选项-a则对所有 CPU 进行记录，使用选项-c n指定每 n 次事件发送进行一次采样，更多选项可输入perf record -h获得帮助。

perf report对收集得到的perf.data进行分析，默认进入交互式页面，初始界面按 h 获得帮助。而使用选项--stdio可一次性全部输出。

下一节就具体例子对重点功能做介绍说明，而更详细的使用说明可以参考下列文档。

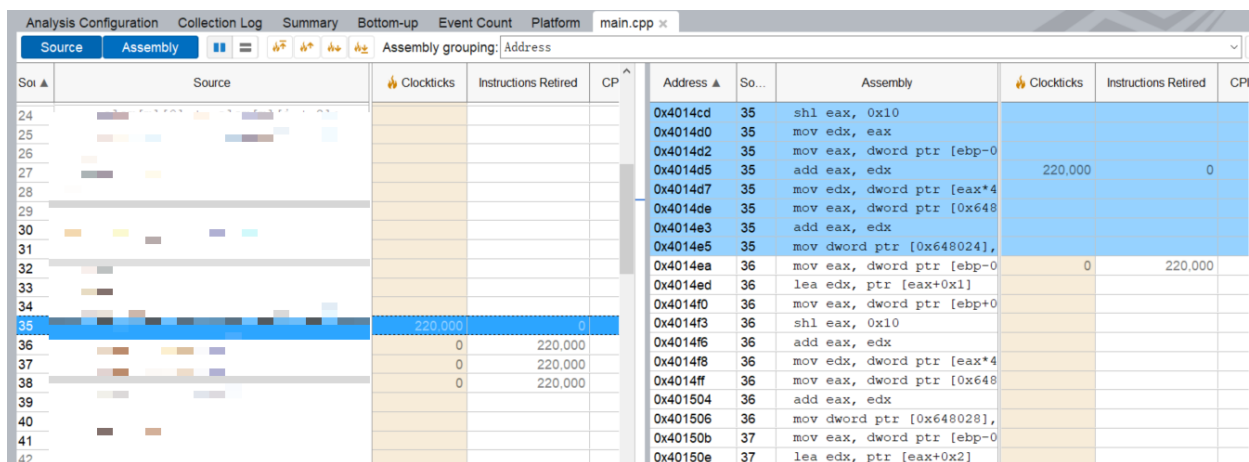


图 6.24: 两路链式算法执行执行分析

1. linux perf 官方 WIKI:<https://perf.wiki.kernel.org/index.php/Tutorial>
2. 官方 WIKI 的一篇译文: <https://segmentfault.com/a/1190000021465563>
3. 一篇详细的 perf 用例 (推荐): <https://www.brendangregg.com/perf.html>

此外, 有几个方法可对 perf 进行可视化分析, 有兴趣可进一步了解。

1. 借助脚本生成火焰图对 perf.data 进行可视化分析。
2. 据[intel 官方文档](#)说明, vtune 工具支持对 perf.data 进行导入分析。
3. 使用一个可以将perf.data可视化的工具[hotspot](#)。

7.3 体系结构相关的程序性能剖析

7.3.1 cache 命中率

仍然以前文所述的数组列求和为例, 在下文的示例程序中, 函数col_major按列优先计算, 而函数row_major按行优先计算。

使用某个编译器对代码进行编译, 以 gcc 为例, 为了在分析过程中保留更详细的符号和源代码信息, 编译时使用调试信息选项-g。确保生成的可执行文件可以正常执行后, 使用 perf 针对 cache 相关事件进行采样。依次输入指令如下

```
1 $ gcc column_sum.c -g -o column_sum
2 # perf record -e L1-dcache-load-misses,L1-dcache-loads,L1-dcache-stores -ag ./row_column
3 # perf report
```

如图 7.25所示, 显示了所选择三个事件的采样数, 并可依次按下 Enter 查看对应的直方图。

如图 7.26所示, 显示了在所有采样的 L1-dcache-load-misses 事件中, 各个函数的占比, 在当前条件下col_major比row_major的 L1 数据缓存载入不命中率要高很多。继续按下 Enter, 并选择 Annotate, 可查看该函数具有源代码注释的反汇编代码, 与各指令上的事件采样比率, 由于硬件平台不同, 可能产生完全不一样的反汇编代码, 图 7.27展示了x86_64平台上的反汇编结果。此外, 如果安装了可视化的 hotspot 工具, 运行# hotspot perf.data, 可以更加清晰地展示火焰图、函数调用关系等。如图

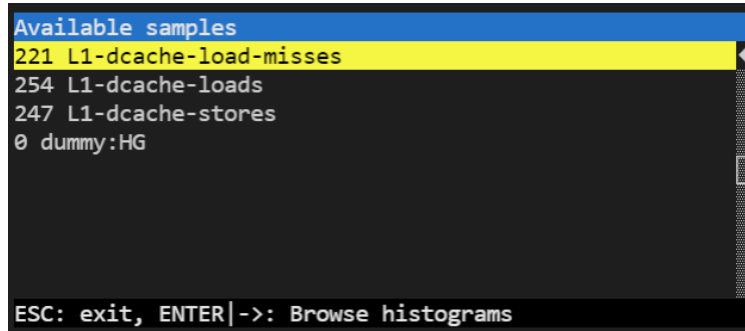


图 7.25: perf report 结果 1

7.28所示，为 L1-dcache-load-misses 事件的火焰图，图中颜色深浅无明显含义，关键看各个函数的宽度，“平原”为该程序的瓶颈处。

影响实验结果的因素有很多，在理解 cache 工作原理的基础上，首先要清楚所用机器的各项参数，在 Linux 上使用命令 `$ getconf -a | grep CACHE` 可以得到各级 cache 的大小 (SIZE 单位 Byte)，相联度 (ASSOC)，缓存行大小 (LINESIZE 单位 Byte)。改变计算规模，或者在不同的机器上，实验结果可能有很大差异。

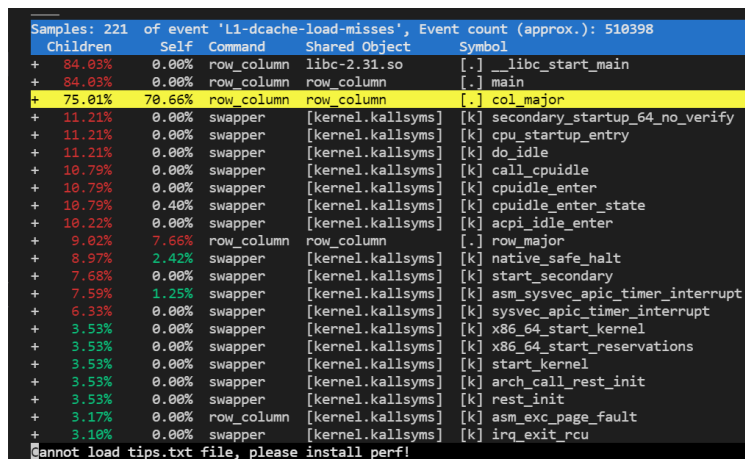


图 7.26: perf report 结果 2

7.3.2 超标量

与 VTune 一节同样，比较两路链式累加算法和普通的链式算法的性能，两路链式累加算法 IPC 应该明显优于普通的链式算法。下例程序对 4096 元素求和 500 次，perf 重复统计 instruction 和 cycles 共 100 次。

Listing 1: 平凡算法

```
$ perf stat -e instructions,cycles -r 100 ./liner
```

```
Performance counter stats for './liner' (100 runs):
```

```
25,310,778 instructions          # 1.00 insn per cycle   ( +- 0.00% )
25,327,056 cycles                ( +- 0.05% )
```


Percent		↓ jmp 9f
		column_sum[i] += b[j][i];
3.12	46:	mov -0x8(%rbp),%eax
		cltq
		lea 0x0(,%rax,4),%rdx
		lea column_sum,%rax
5.27		mov (%rdx,%rax,1),%edx
		mov -0x8(%rbp),%eax
		cltq
		mov -0x4(%rbp),%ecx
3.94		movslq %ecx,%rcx
		shl \$0xb,%rcx
		add %rcx,%rax
		lea 0x0(,%rax,4),%rcx
1.96		lea b,%rax
		mov (%rcx,%rax,1),%eax
67.53		lea (%rdx,%rax,1),%ecx
6.94		mov -0x8(%rbp),%eax
3.97		cltq
		lea 0x0(,%rax,4),%rdx
		lea column_sum,%rax
		mov %ecx,(%rdx,%rax,1)
		for (j = 0; j < N; j++)
6.27		addl \$0x1,-0x4(%rbp)
	9f:	cmpl \$0x7ff,-0x4(%rbp)
		Press 'h' for help on key bi Press any key...

图 7.27: perf report 结果 3

0.0058768 +- 0.0000205 seconds time elapsed (+- 0.35%)

Listing 2: 2 路展开累加

\$ perf stat -e instructions,cycles -r 100 ./2way

Performance counter stats for './2way' (100 runs):

22,235,765	instructions	#	1.71	insn per cycle	(+- 0.01%)
12,984,925	cycles				(+- 0.05%)

0.0030010 +- 0.0000124 seconds time elapsed (+- 0.41%)

对比普通的链式算法和两路链式累加算法,指令数 25,310,778 与 22,235,765 相近,周期数 25,327,056 与 12,984,925 翻倍,IPC (CPI 倒数) 1.00 和 1.71 有较大差距。

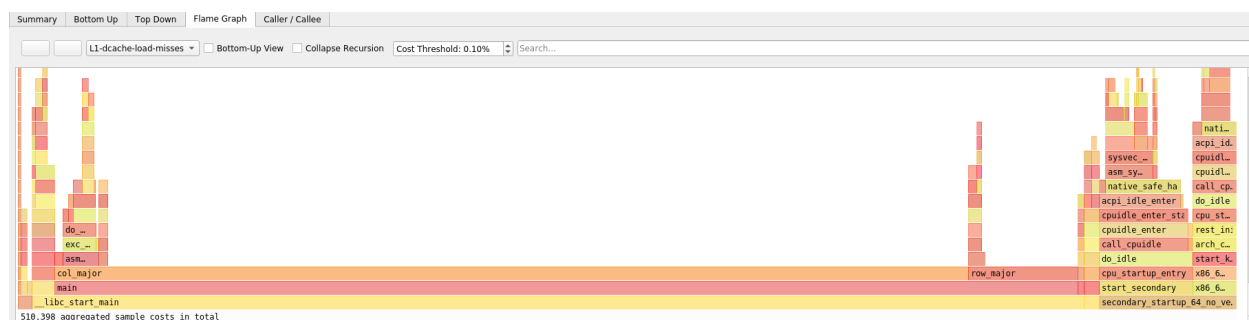


图 7.28: hotspot Flame Graph

这里也可使用 `perf record` 对函数详细剖析，分别收集 instruction 和 cycle 数计算 CPI。

8 使用 Godbolt 分析汇编程序

Godbolt 是一个很好用的在线编译浏览器。如图 8.29 所示，我们打开 godbolt 主页，在左侧窗口中输入 C/C++ 源程序（这里输入的是 n 个数求和的链式算法的程序），在右侧窗口左上方选择编译器（这里选择的是我们安装 oneAPI 就会自带的 Intel icx 编译器），在右侧窗口就会显示编译出的汇编代码。我们通过研究汇编代码，能更深层次地分析、理解程序为什么会产生相应的性能。通过对比不同算法的汇编代码、对比不同优化选项下生成的汇编代码，也能理解、学习更底层的程序优化方法。例如，如图 8.30 和图 8.31 所示，在右侧窗口右上方的文本框内输入 -O3 编译选项，右侧窗口就会显示 -O3 优化力度下编译出的汇编代码，与无优化的版本（图 8.29）相比，可以看出很多部分、特别是核心的循环计算部分有明显不同——编译器对源程序进行了自动向量化，利用 SSE 指令对核心的累加运算实现了 SIMD 并行，可显著加速目标程序。我们还可以输入更复杂的编译选项，例如输入 `-march=skylake-avx512` 以生成 skylake 架构 CPU 上 avx512 的汇编代码。

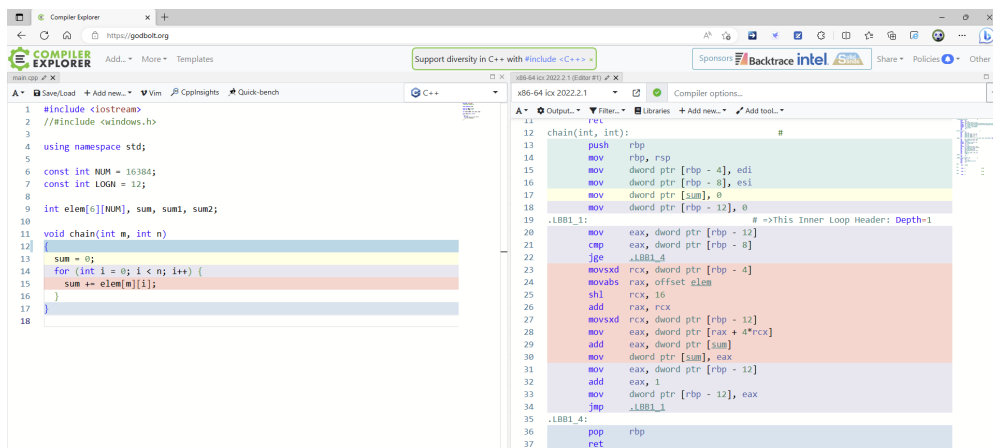


图 8.29: 使用 godbolt 分析求和程序的汇编代码。

Godbolt 支持数十种主流编程语言、数百种主流编译器，涵盖了 x86、arm、powerpc 等主流平台架构。而且，godbolt 提供了一些功能，帮助使用者研究、分析编译得到的汇编程序。例如，godbolt 会以相同颜色对应显示源程序代码片段及其翻译出的汇编代码，方便使用者对应分析翻译结果。再如，如果你对某个汇编指令并不熟悉，可以将悬停在对应指令上，godbolt 会弹出一个窗口显示该指令的具体解释，如图 8.32 所示。如果你觉得解释还不够详细，可以在指令上点击鼠标右键，在弹出的上下文菜单中选择“View assembly documentation”，godbolt 会在一个弹出窗口中显示指令更详细的解释，还可能包含指向指令详细文档的超链接，如图 8.33 所示。

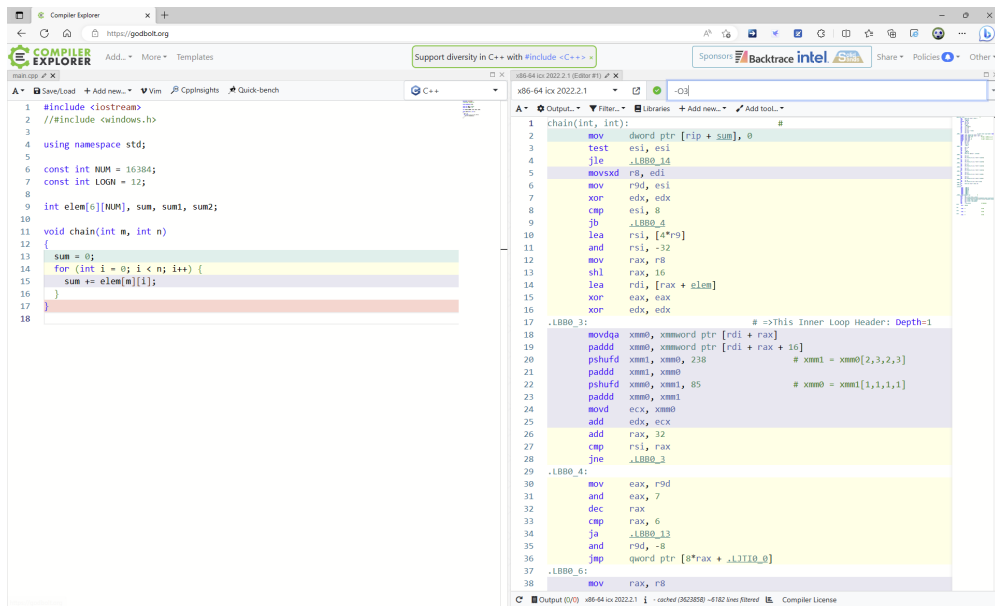
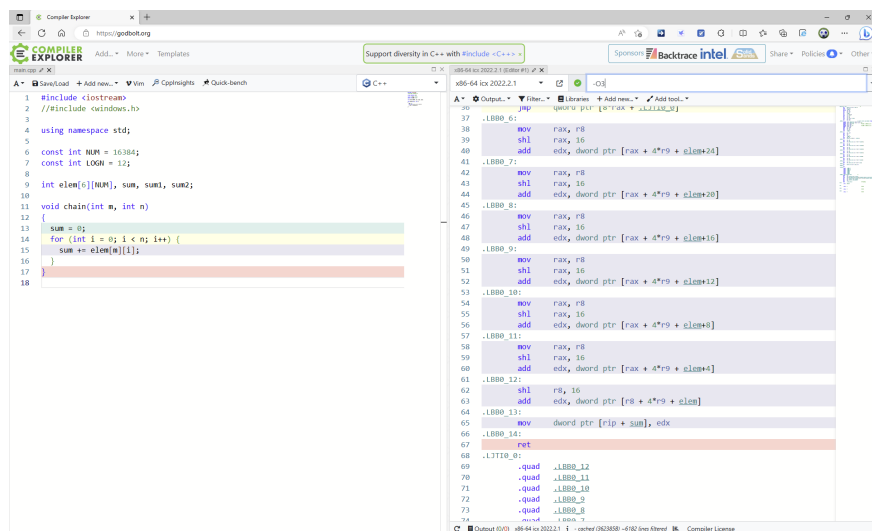


图 8.30: 使用 godbolt 分析求和程序优化编译得到的汇编代码。



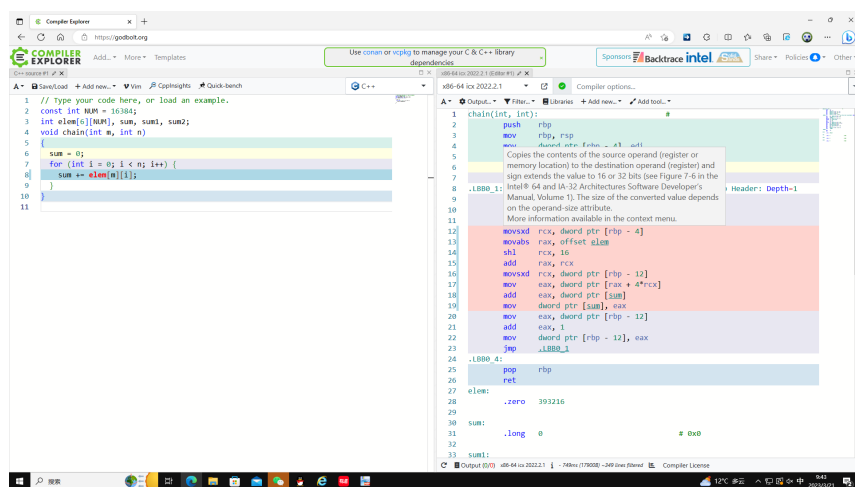


图 8.32: Godbolt 给出汇编指令的解释。

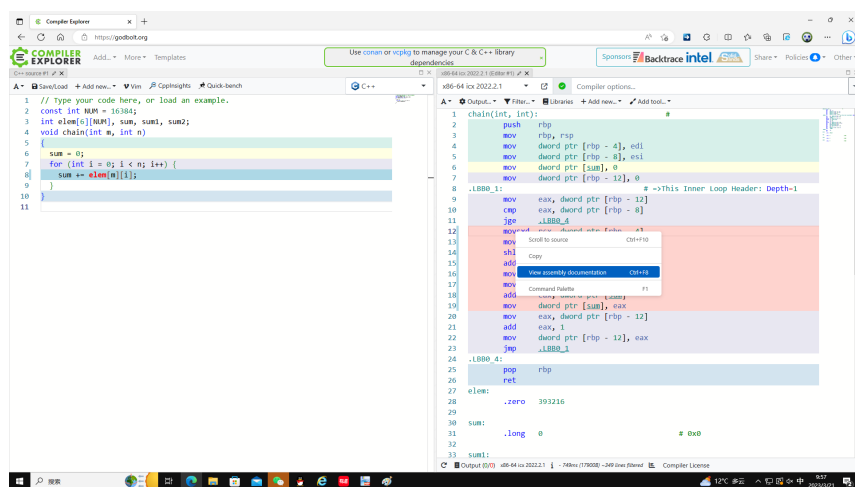


图 8.33: Godbolt 上下文菜单。