# 基于缓存管理的 GPU 通用计算优化的设计与实现

# 摘　要

GPU 为了对抗存储墙的问题采用 2 级缓存结构，缓存设置的初衷是基于访存请求的空间局部性和时间局部性来减少访存延时，然而由于 GPU 私有 L1 缓存结构，导致缓存之间存在重复数据，进而导致缓存利用率低；而且发现 cache line 的大小会影响到最终实际访存数据的大小；并且高复用距离的访存请求可能会污染 L1 缓存，导致 L1 缓存高缺失率。本文的主要贡献有三点，首先分析共享 L1 缓存结构，它消除私有 L1 缓存有重复数据的问题；其次分析 sector 缓存和非 sector 缓存的特点以及分别适合于哪些特点的程序；最后提出三种选择性缓存旁路策略，将高复用距离的访存请求旁路 L1 缓存。经过研究和测试，共享 L1 缓存结构可以平均减少 27.2%缓存缺失率；三种选择性缓存旁路策略分别对于平均复用距离较高的程序可以分别提升 17%、9.1%和 20.4%的 IPC，而对于平均复用距离较低的程序可以分别平均保持 99.78%、100%和 100.1%的 IPC。

**关键词：GPU　L1 缓存　缓存旁路　复用距离　缓存组织方式**

# ABSTRACT

Graphics Processing Units (GPUs) use two-level caches to confront the memory wall. The original intention of cache is to reduce the latency of access based on the spatial locality and time locality of access requests. However, due to the GPU private L1 cache structure, there is repeated data between caches, which lead to low cache utilization. Furthermore, the size of cache line can make a difference to the size of data that actually access. Besides, access requests with high reuse distance may contaminate L1 cache, resulting in high loss rate of L1 cache. The main contributions of this thesis are as follows: firstly, the shared L1 cache structure is proposed to eliminate the duplicate data of the original private L1 cache; secondly, the characteristics of sector cache and non-sector cache and the programs with which characteristics are suitable are analyzed ; finaly, three kinds of selective L1 cache bypass strategy are proposed, which is bypassing L1 cache for the access requests with high reuse distance. After research and test, sharing L1 cache structure can reduce cache miss rate by 27.2% on average. Three kinds of selective bypass strategy can improve IPC by 17%,9.1% and 20.4% respectively for applications whose average reuse distance is high, while applications whose average reuse distance is low can maintain IPC by 99.78%,100% and 100.1% on average respectively.

**Keywords: GPU L1 data cache bypass cache reuse distance cache organization**

# 目 录

# 第一章 绪论

## 1.1 研究背景

GPU 全称 graphics processing unit，即图形处理单元。尽管早期 GPU 是用于图像渲染，而在当今人工智能火热的时代，GPU 经常被用来做模型的加速训练，它的功能却不局限于图像渲染和模型训练。最近几十年，GPU 成为通用程序的执行平台，称为 GPGPU（通用图形处理器）。通过类似 OpenCL 和 CUDA 的编程语言，编程者可以在各种领域使用大规模并行架构，例如分子科学、高性能计算、线性代数等。

在摩尔定律越来越难以维持的当今时代，计算机体系结构面临撞上功耗墙和存储墙[1]的问题。功耗墙是指由于 CPU 功率过大导致的发热现象短时间内很难降温冷却，因此 CPU 的单核性能达到了极限，只能向多核和并行方向发展，而 GPU 是早期用于图形图像并行处理而发明出来的。存储墙是指处理器计算性能的提升速度远远超过主存（主要是 DRAM）的访问速度，主存的访问延时已经成为整个系统的性能瓶颈。

## 1.2 研究现状

GPU 为了对抗存储墙的问题，采用传统的二级缓存结构[2][3]，其中 L1 缓存是每个 GPU 核心（NVIDIA 称其为 SM:streaming multiprocessors，即流式多处理器）私有的，而 L2 缓存是通过片上互联网络对所有 GPU 核心共享且分区存储的。L1 和 L2 缓存的出现是通过增加片上带宽来应对存储墙的手段。而片上带宽的增加对于访存密集型的程序可以转换成性能的提升。

由于缓存需要实现空间局部性的特点，所以一般缓存的数据量会大于实际需要的数据量，例如缓存块的基本单元大小为 128B，即使当前程序只需要访问 1B 的数据，依然需要最少 128B（即缓存块基本单元大小）的数据访问，所以不规律的访存序列会导致大量多余的数据访问，进而加重程序的访存负担，尽管之后这些数据可能会被访问，但是很可能由于缓存本身大小不足而造成数据块替换，进

而造成缓存命中率降低。由此可见，访存序列是否规律和缓存块基本单元大小紧密相关，合适的缓存块基本单元大小对于访存集中型程序的性能至关重要，因此本文分析了 GPU 中 sector 缓存和非 sector 缓存对程序性能的影响。

常见的缓存策略主要有 LFU（最不经常使用）、LRU（最近最少使用）和 FIFO（先入先出）[4]，可以看到这些缓存策略均为被动式预测策略。“被动”意思是只有当该数据块缺失时，它才会放到缓存中，与此相对的还有缓存预取策略，即预测还未被访问到的数据将来可能会被访问，进而提前将这些数据块放入到缓存中。“预测”意思是只能根据之前的缓存命中或缺失信息来推测未来哪些数据块会被访问，进而执行相应的替换策略，例如 LRU 认为最近访问的数据块将来会被再次访问的概率最大，所以 LRU 就会替换那些最近最少使用的数据块。事实上，不同程序的访存特点并不是完全一致的，只使用静态的预测策略很容易出现在某些情况下出现较高的性能损失。因此，本文通过使用复用距离来分析 LRU 策略对于在 GPU 上执行的程序的影响（若无其他说明本文实验中使用的缓存替换算法均为 LRU），并提出了相应的旁路策略。

## 1.3 论文主要工作

**本文主要有如下贡献：**

- 在对某些访存集中型程序的访存序列研究中发现，每个 GPU 核心的访存序列的复用距离分布极其相似，且不同 GPU 核心会大量访问相同的缓存块，这就造成了在每个 GPU 核心的私有 L1 缓存有相当一部分缓存是相同的[5][6][7]，这极大地降低了 L1 缓存的利用率，为此本文分析了共享 L1 缓存结构，共享 L1 缓存对于所有 GPU 核心是共享的，也就是说任何一个 GPU 核心都可以访问共享 L1 缓存，这完全消除了原来私有 L1 缓存中缓存重复问题。
- 通过研究分析 sector 缓存和非 sector 缓存的组织特点，得出影响这两个缓存组织的重要因素是缓存基本单元的大小，并根据访存特点将程序划分为访存集中型和访存分散型，其中访存分散的程序更适合较小的 cache line 大小，其中适合 sector 缓存的程序最高得到了 188.6%的 IPC 提升，而不适合 sector 缓存的程序最高得到了 14.5%的 IPC 下降。
- 通过研究不同程序访存序列的复用距离分布，发现高缓存缺失率的程序访存序

列复用距离远高于低缓存缺失率程序，因此高缓存缺失率主要和高复用距离的访存有关，为了避免 L1 缓存被高复用距离的访存污染，本文提出了三种选择性旁路策略（selective bypass，简称 SBP），分别是 SBP-split、SBP-stage 和 SBP-LRU，SBP 根据存储块的特定指标来判断访存请求是否值得旁路。

- 经过研究和测试，在保证 L1 缓存总大小不变情况下，共享 L1 缓存结构相比于私有 L1 缓存结构可以平均减少 27.2%缓存缺失率。SBP-split、SBP-stage 和 SBP-LRU 对于平均复用距离较高的程序可以分别平均提升 17%、9.1%和 20.4%的 IPC，而对于平均复用距离较低的程序可以分别平均保持 99.78%、100%和 100.1%的 IPC。

## 1.4 论文结构

本文的其余章节组织如下：

在第二章对本文用到的相关理论和知识进行总结和介绍。

在第三章首先提出本文实验用到的相关程序分析指标并建立实验，之后对实验提取到的访存序列进行分析，最后分析影响程序性能的缓存架构相关因素并提出选择性缓存旁路策略。

在第四章对本文的工作进行了总结，并对未来的工作进行展望。

# 第二章 相关理论概述

## 2.1 NVIDIA GPU 及其架构

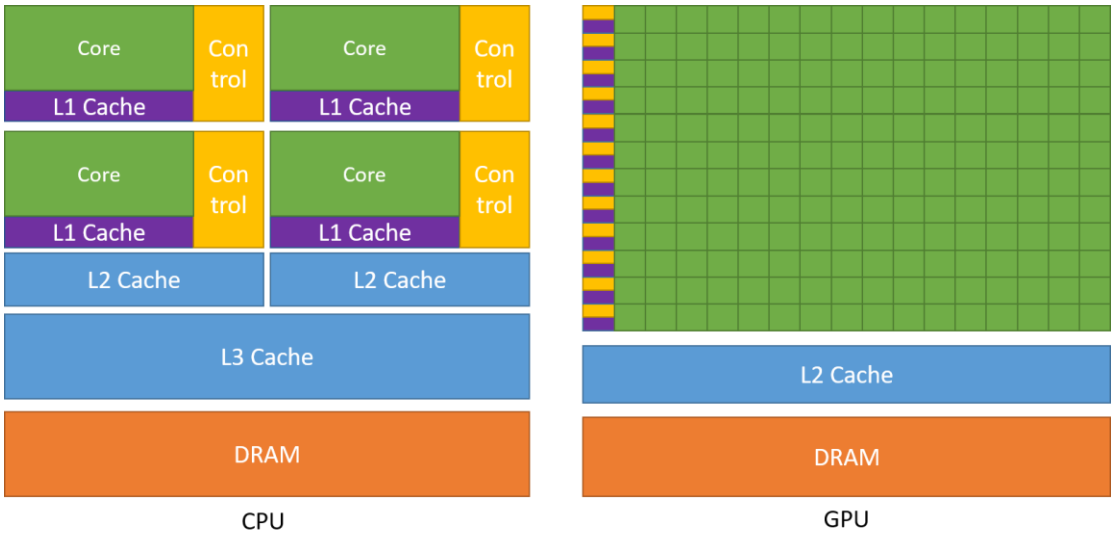


图 2.1 CPU 和 GPU 架构对比[8]

从图 2.1 可以看出 CPU 中只有少部分的晶体管来完成实际的运算工作，而大部分晶体管用来实现控制电路和缓存，所以其比较适合串行处理；而 GPU 则与之相反，它的控制逻辑相对简单，且大部分晶体管用来做运算，所以其多用于并行计算。GPU 的执行单元（NVIDIA 称其为 CUDA 核心，也叫 SP:streaming processors，即流处理器）是 CPU 核心的精简版，它去除了缓存单元、乱序执行控制器、分支预测器、存储预取器等。GPU 的并行性体现在它的多个执行单元并行地执行不同的程序片元。每一个 SM（流式多处理器）包含多个 CUDA 核心和片上存储。每一个 SM 中的 CUDA 核心执行相同的指令流，但是数据不相同，即 SIMD。Warp 作为基本的调度单元，一般包含 32 个线程，Warp 上每一个线程也是执行相同的指令流并且同步地执行，即 SIMT。Warp 需要分配到指定的 SM 上执行。由于访存延迟过大，GPU 采用一种机制来隐藏延迟。这种机制称为延迟隐藏：多个 Warp 分配到一个 SM 上去执行，每一个时刻有固定数量的 Warp 在 SM 上执行，当正在运行的 Warp 需要访存而停滞时，调度下一个 Warp 在 SM 上执行，当由于访存而停滞的 Warp 获取到需要的数据后，它可以重新被调度执行。通过这样的调度机制，GPU 可以获得较大的吞吐量提升，但是这并不意味着 GPU 没有受到访存延迟而导

致的性能下降，当分配给一个 SM 上的所有 Warp 均因访存而停滞时，就会导致 GPU 的严重性能下降。

GPU 存储层次有寄存器、共享存储（shared memory）、1 级缓存（L1 cache）、2 级缓存（L2 cache）、全局存储（global memory）和主机端存储（host memory）。其中设备端一般指 GPU，主机端一般指 CPU，设备端和主机端的交互通过 PCIE 总线。共享存储是除寄存器外可编程的最快访问资源，是线程块内所有线程共享的。当需要的数据不在寄存器和共享存储时，会首先访问 L1 cache，当 L1 cache 丢失时，会访问 L2 cache，当 L2 cache 丢失时，会访问全局存储，这就构成了 GPU 的多级存储层次。通常编程人员会在程序起始时，将所需要的数据从主机端存储放入共享存储或全局存储或纹理缓存。因为 L1 缓存是缓存体系的第一级，L1 缓存缺失造成的性能损失远远大于 L2 缓存，所以本文主要以 L1 缓存为例，L2 缓存的分析方法和 L1 缓存一致。L1 缓存一般包括数据缓存、指令缓存和纹理缓存，如果没有特殊说明，本文中的 L1 缓存特指数据缓存。

## 2.2 CUDA 编程模型

CUDA 是 NVIDIA 在 2006 年 11 月提出的一个通用意义并行计算平台和编程模型[8]。它可以利用 NVIDIA 提供的并行计算引擎通过一种比 CPU 更快速的方式来解决很多复杂的计算编程问题。CUDA 的软件开发环境允许开发者使用 C++作为高级编程语言。在 CUDA 编程模型中，GPU 被当做一个协处理器：在 CPU 上运行的程序可以启动一个大规模并行计算 kernel，而 kernel 的执行在 GPU 上。kernel 有多个线程网格组成，每个线程被赋予特定的标识符来划分工作任务。在一个线程网格内，线程被组成线程块（也叫 cooperative thread arrays，即 CTAs）。在一个线程块内有可以快速访问的存储，称为共享存储。如果需要的话，线程块内的线程可以执行同步操作。

CUDA 主要提供了三个抽象概念：线程层级，共享的存储，屏障同步，这些抽象概念提供了嵌套在粗粒度的数据并行性和任务并行性中的细粒度数据并行化和线程并行化。它指导编程者将问题划分为可以在线程块独立运行的粗粒度的子问题，每一个粗粒度的子问题也可以分为可以被线程块内合作的线程解决的更细粒度的部分。

在 CUDA 编程模型中主要有三个函数类型限定词：__device__、__global__和__host__。__device__声明一个函数是在设备端（GPU）执行的，且仅可从设备端调用。__global__声明一个函数作为一个存在的 kernel，这样的函数是在设备端（GPU）执行的，且仅可从主机端（CPU）调用。__host__声明的函数是在主机端执行的，且仅可从主机调用。

在 CUDA 编程模型中主要有三个变量类型限定词：__device__、__constant__和__shared__。__device__声明驻留在设备上的一个变量，这个变量驻留在全局内存空间、具有应用的生存期并且从线程网格内所有线程和从主机通过运行时库是可访问的。__constant__与__device__一起使用，声明驻留在设备上的一个变量，这个变量驻留在常量内存空间，具有应用的生存期并且从线程网格内所有线程和从主机通过运行时库是可访问的。__shared__与__device__一起使用，声明驻留在设备上的一个变量，这个变量驻留在线程块的共享内存空间中、具有块的生存器并且只有块之内的所有线程是可访问的。

## 2.3 GPU 仿真器 GPGPU-Sim

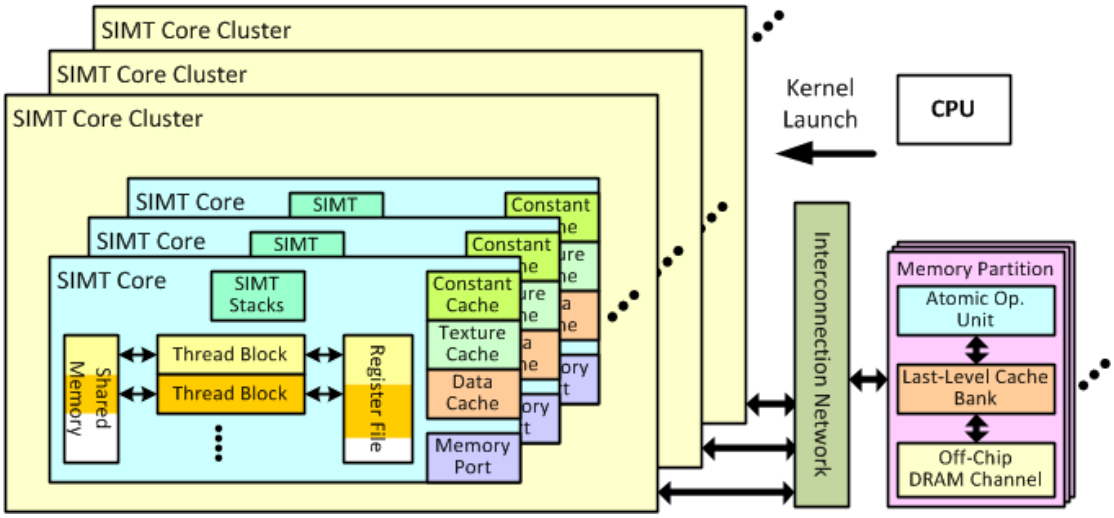


图 2.2 GPGPU-Sim 架构图

GPGPU-Sim 是一个周期级别的专注于通用 GPU 计算的 GPU 性能仿真器[9][10]。它通过重新链接编译成可执行文件的 CUDA 程序，来进行 GPU 仿真。重新链接后的可执行文件并不会在真实的 GPU 上运行，而是在 CPU 上运行 GPGPU-Sim 的仿真。

图 2.2 展示了 GPGPU-Sim 的微架构,它主要由 SIMT 核心簇、互联网络和存储分区三部分组成。其中互联网络连接 SIMT 核心簇和存储分区，SIMT 核心与流式多处理器（SM）相似，存储分区模拟 L2 缓存和显存。

## 2.4 缓存组织方式

目前共有四种缓存组织方式：组相联、全相联、直接映射和 sector。组相联是指内存区域中的任何一个地址的数据都只能存储在缓存中的对应一组部分存储块中。全相联是指内存区域的任何一个地址的数据均可以存储在缓存中的任何一块中。直接映射是指内存区域的任何一个地址的数据都只能存储在缓存中的唯一一个存储块。sector 是指缓存由一组 sector 组成，每个 sector 对应一个地址标记，sector 又进一步分成 subsector，每个 subsector 都有自己 valid 和 dirty 标记。sector 缓存的主要优势在于可以用更少的标记位，并且减少数据传输[11]。图 2.3 为 sector 缓存组织方式示例。

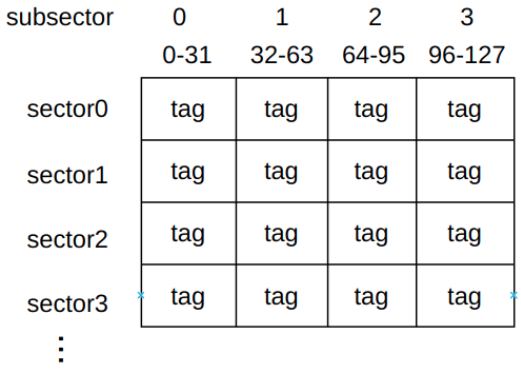


图 2.3 sector 缓存组织方式（128B cache line）

## 2.5 缓存缺失和命中后的处理

缓存缺失分为读写缺失，同样缓存命中也分为读写命中。一般读命中只会更新相关缓存块的参数，例如最后一次访问时间，而读缺失需要从下一级存储申请调入相关缓存块，并在缓存满的情况下进行缓存替换。写命中和写缺失之后的处理相对来说复杂些，因为涉及到缓存和下一级存储一致性的问题。写命中后一般有 3 种处理策略：1.write-back，在写命中时只更新缓存中的数据，当缓存块被逐出缓存时，将更新写入下一级存储。2.write-through，在写命中时同时更新缓存和下一级存储中的数据。3.write-evict，在写命中时逐出相关缓存并更新下一级存储

中的数据。写缺失后一般有 2 种处理策略：1.no write-allocate，不将相关数据读入缓存，而是直接在下一级存储更新数据。2.write-allocate，将相关数据读入缓存，并采取写命中时策略。本文中 L1 缓存的写策略见表 2.1。

表 2.1 L1 缓存写命中和缺失策略

| | Local Memory | Global Memory |
|---|---|---|
| Write Hit | Write-back | Write-evict |
| Write Miss | Write no-allocate | Write no-allocate |

## 2.6 复用距离（reuse distance）

表 2.2 访存序列复用距离示例

| 访问 | a[0] | a[3] | a[0] | a[1] | a[2] | a[3] |
|---|---|---|---|---|---|---|
| 地址 | 0 | 3 | 0 | 1 | 2 | 3 |
| 复用距离 | ∞ | ∞ | 1 | ∞ | ∞ | 3 |

给定一个存储访问序列，对于每一次访问，复用距离被定义为这次访问和上一次对于相同地址访问之间相隔的不重复的地址访问次数[12][13][14]。当之前没有访问过某个地址时，距离被设置为无穷。表 2.2 为“030123”访存序列复用距离示例。

在使用 LRU 的替换策略时，通过分析复用距离，可以直接得出一个全相联缓存的缺失率。假设可以缓存 n 个缓存块，一次访问的复用距离为 d，当 d 大于 n 时，这次访问是缓存缺失，当 d 小于等于 n 时这次访问是缓存命中。

## 2.7 本章小结

本章首先说明了 NVIDIA GPU 相关知识和 NVIDIA 提出的 CUDA 编程模型，其次说明了本文实验用到的 GPU 模拟器 GPGPU-Sim，之后对缓存架构和相关的读写缺失命中处理进行了阐述，最后提出了本文对访存序列分析用到的复用距离。

# 第三章 设计、分析和实验

## 3.1 程序分析指标

本文主要用到的程序分析指标如下：

- 时钟周期数（cycle）
- 每时钟周期执行的指令个数（IPC）
- 平均访问次数（average number of access）
- 缓存缺失率（miss rate）
- 带宽利用率（bandwidth availability ratio）

其中 IPC 一般用于衡量 GPU 运行程序时的性能；平均访问次数是指每时钟周期缓存平均访问次数；缓存缺失率为缓存缺失次数除以总的缓存访问次数；带宽指每时钟周期可以传输的数据量，而带宽利用率是指当前带宽除以最大带宽。

## 3.2 共享 L1 缓存架构

**私有 L1 缓存数据重复：**在发现一个程序不同 SIMT 核心对 L1 缓存访问序列在复用距离高度相似后，进而就想到不同 SIMT 核心是否会访问到相同缓存块，之后便计算了被不同 SIMT 核心访问到的缓存块数量占访问的所有缓存块数量的比例，以 BFS 程序为例，见图 3.1（横轴为访问同一缓存块 SIMT 核心数，纵轴为缓存块所占比例，例如图 3.15 中 0.461 指同时有两个 SIMT 核心访问到的缓存块所占比例（其他测试程序的结果见附录）。

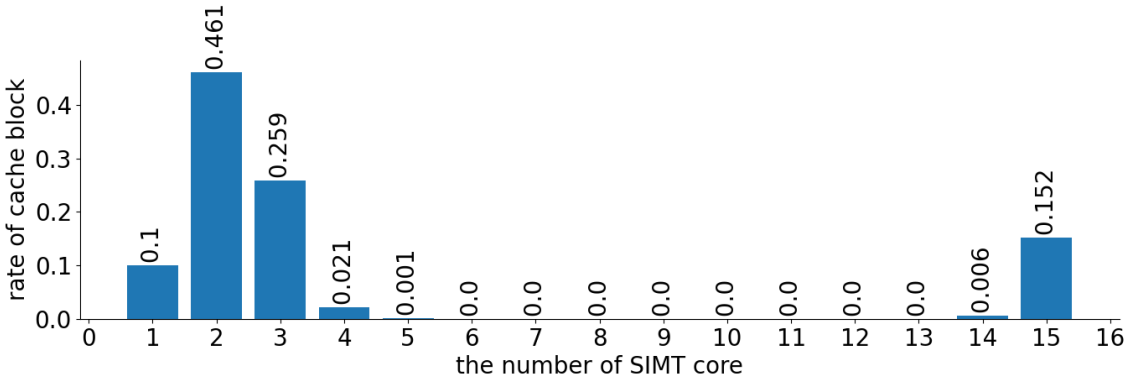


图 3.1 不同 SIMT 核心访问到的缓存块数量占总缓存块数量的比例

**共享 L1 缓存架构：**从图 3.1 可以得出确实存在不同 SIMT 核心访问相同缓存块的情况，所以为了减少 L1 缓存重复数据，分析了共享 L1 缓存架构。

| 共享 L1 缓存架构 |
| --- |
| 将私有 L1 缓存变为共享 L1 缓存，且只存在一块共享 L1 缓存，每一个 SIMT 核心都可以通过互联网络访问该共享 L1 缓存，且保证其他因素不变（包括 L1 缓存总大小等）。 |

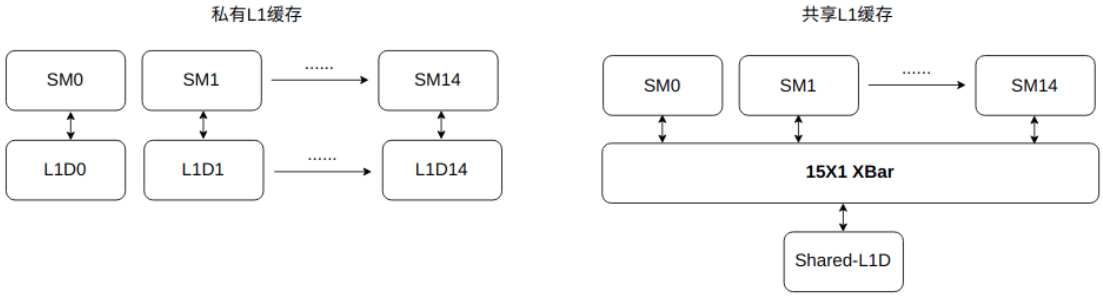


图 3.2 私有 L1 缓存和共享 L1 缓存架构对比

根据图 3.2 中的共享 L1 缓存架构可以完全解决掉 L1 缓存重复的问题，进而提高 L1 缓存的利用率。为了测试上述设想，以访存序列作为输入进行建模，模拟程序访问 L1 缓存，缓存块替换策略为 LRU。基准架构为 15 个私有 L1 缓存，可以缓存 128 个大小为 128 字节的缓存块，每个 L1 缓存只缓存对应的 SIMT 核心的访存；共享 L1 缓存架构为只有一个公有 L1 缓存，可以缓存 1920 个大小为 128 字节的缓存块，所有 SIMT 核心的访存均可以缓存到公有 L1 缓存，归一化结果如下。（归一化指共享 L1 缓存架构的模拟指标和基准架构下的模拟指标之比）

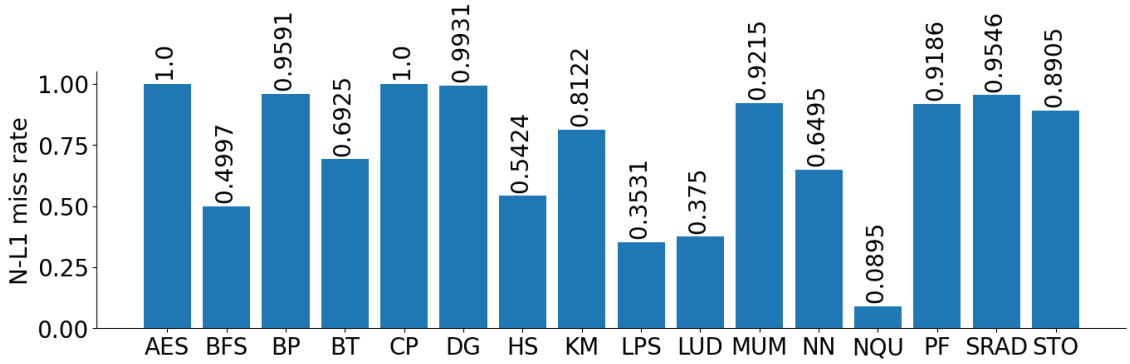


图 3.3 共享 L1 缓存架构得到的归一化 L1 缓存缺失率

**共享 L1 缓存架构性能分析：**从图 3.3 可以看出共享 L1 缓存架构在总缓存大小不变的情况下极大地降低缓存缺失率。尽管共享 L1 缓存架构有着如此优秀的表现，但是它在很大程度上降低 L1 缓存的访问带宽（大约降低到原来的 1/60，如果使用 32 字节链路[6]），且在硬件实现时需要添加片上互联结构来连接 SIMT 核心和 L1 缓存，这无疑需要额外的开销，但是从图 3.4 发现在基准测试中大部分程序 L1 缓存的带宽利用率都很低，说明可以通过牺牲部分 L1 缓存带宽来换取 L1 缓存缺失率的降低，或提高互联网络的带宽来弥补访问带宽的下降。

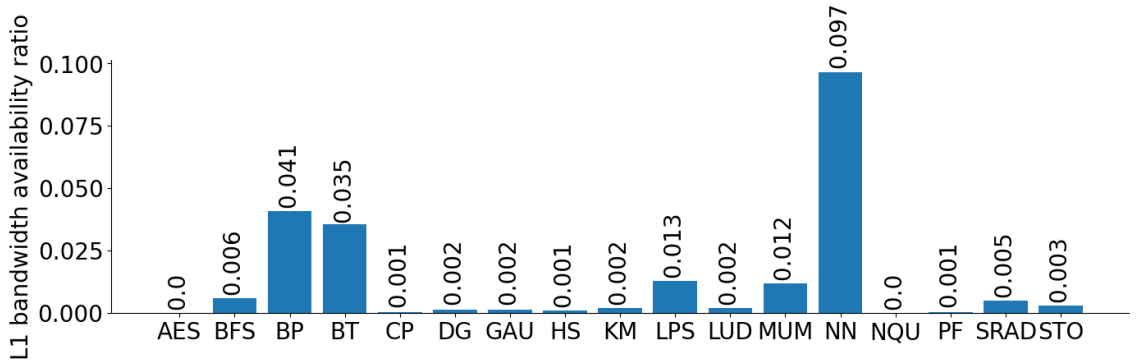


图 3.4 基准配置下测试程序 L1 缓存带宽利用率

## 3.3 Sector 缓存和非 Sector 缓存的对比与分析

在 GPGPU-Sim 的配置中 Fermi 架构的 L1 缓存采用的是非 sector 组织方式，而 Turning 架构的 L1 缓存采用的是 sector 组织方式，为了控制变量，将基于 GTX480 配置文件进行修改对比在非 sectorL1 缓存和 sector L1 缓存下测试程序的表现，表 3.1 为 GPGPU-Sim 配置对比（其他配置和基准配置相同）。

表 3.2 缓存组织方式配置对比

| L1 缓存 | GPGPU-Sim 配置 |
|---|---|
| 非 sector 缓存 | N:1:128:128,L:L:m:N:L,S:64:8,8 |
| sector 缓存 | S:1:128:128,L:L:m:N:L,A:64:8,8 |

表 3.3 sector 缓存归一化指标

| | AES | BFS | BP | BT | CP | DG | GAU | HS | KM | LPS | LUD | MUM | NN | NQU | PF | SRAD | STO |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1.01 | **1.70** | 0.96 | 0.99 | 1.00 | 1.00 | 0.99 | 1.00 | **2.87** | 0.96 | 1.01 | 1.00 | **0.86** | 1.00 | 0.93 | 0.97 | 1.00 |
| 2 | 4.00 | 1.33 | 2.27 | 1.70 | 2.00 | 3.88 | 1.69 | 1.92 | 1.00 | 1.78 | 2.00 | 1.01 | 1.15 | 1.02 | 2.43 | 1.83 | 1.93 |
| 3 | 1.00 | 1.03 | 1.00 | 1.63 | 1.00 | 1.00 | 1.05 | 1.02 | 1.00 | 1.28 | 1.08 | 2.12 | 3.33 | 1.00 | 1.46 | 1.11 | 0.62 |
| 4 | 1.01 | 0.47 | 0.88 | 0.76 | 0.69 | 1.00 | 0.53 | 0.56 | 0.27 | 0.67 | 0.65 | 1.00 | 0.98 | 1.00 | 0.91 | 0.64 | 1.00 |
| 5 | 1.00 | 1.09 | 1.13 | 1.10 | 1.44 | 0.98 | 1.75 | 1.68 | 3.65 | 1.44 | 1.53 | 1.00 | 0.95 | 0.99 | 1.09 | 1.56 | 1.00 |

注：1. IPC 2. L1 访问次数 3. L1 缺失率 4. L2 访问次数 5. L2 缺失率

表 3.3 为 sector 缓存组织形式相对于非 sector 缓存组织形式的归一化结果，可以看到 BFS 和 KM 得到了很大的性能增益，而 NN 则相反获得了较大的性能减益。sector 缓存和非 sector 缓存组织形式另外一个主要不同是访存合并的最大单元即缓存的基本单元，sector 缓存为 32B 而非 sector 缓存为 128B，访存合并的最大单元越小，实际访存的数据量越小，这一点可以从图 3.22 L2 缓存访存数据普遍下降得到证明；缓存的基本单元容量越小，往往缓存的缺失率会越大，因为削弱了缓存的空间局部性，这一点可以从图 3.21 L1 缓存缺失率普遍上升得到证明；这对于访存分散的程序很有利（访存分散指访存的数据在地址空间中的距离较大），而对于访存集中的程序不利，图 3.5 的类型 1 和类型 2 分别为访存集中和访存分散的表现。BFS 在计算每一个节点的邻接节点和距离时需要访存，而一个节点的邻接节点往往在存储中不是相邻的，造成了 BFS 访存的分散性，而 NN 有多层神经网络，在进行前向传播和反向传播时需要用到每一层神经网络的参数，而一层神经网络中的参数往往在存储中是相邻的，所以 NN 访存是集中的，这也解释了 BFS 和 NN 程序 IPC 表现差异的原因。而 BFS 访存分散导致访存具有较弱的空间局部性（BFS 的 L1 缓存缺失率基本不变），又因为 BFS 程序的瓶颈恰恰是访存，所以 BFS 在访存得到改善后 IPC 变化很大。

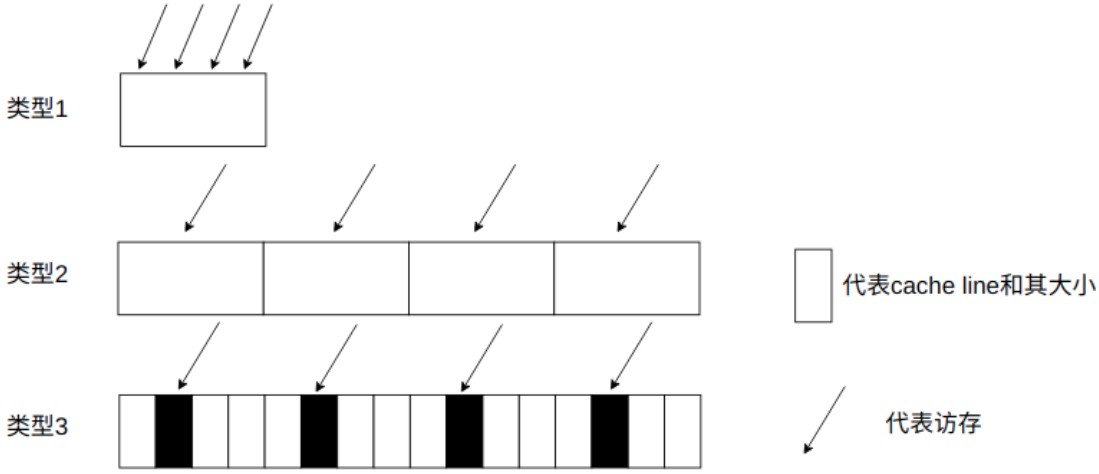


图 3.5 访存特点和 cache line 大小对比

如果透过现象看本质的话，造成 sector 缓存和非 sector 缓存性能表现不同的主要原因是 cache line 的大小，图 3.5 类型 2 和类型 3 则表现了不同大小的 cache line 对于实际访存的数据量的影响。而不同程序的最优 cache line 大小（最优 cache line 大小是指在这个 cache line 大小时程序可以获得最佳的性能）肯定是有所不同的，而 GPU 因其本身的体系结构特点采用了固定的 cache line 大小，这无疑会对某些特点（设定的 cache line 与最优 cache line 大小相差过大）程序造成较大的性能损失。[17]为了解决缓存利用率低的问题，将 sector 缓存进行了优化，在一个 cache line 中可以存放空间不连续的数据；[18]提出了动态变化 cache line 大小的策略，但是这对于硬件部分实现比较困难。

## 3.4 选择性缓存旁路策略

**访存序列复用距离分析**：仅仅是从缓存访问次数和缓存缺失率两个指标来分析缓存性能往往太过宏观，因此通过修改 GPGPU-Sim 源代码，在进行仿真程序的同时将 L1 缓存访问的缓存块的地址序列输出到文件中，这样就获得了访问 L1 缓存地址序列。复用距离是衡量访存序列的一个指标，用复用距离对得到的访问 L1 缓存地址序列进行分析，由于 L1 缓存是每一个 SIMT 核心（NVIDIA 称其为 SM，GPGPU-Sim 称其为 SIMT 核心）所私有的，在基准测试时配置的有 15 个 SIMT 核心，所以会得到 15 个访存序列，因为每个 SIMT 核心分配到的 warp 执行的指令是有重复的，所以这 15 个访存序列的复用距离分布大致是相似的。

另外一个发现是不同程序的复用距离分布可以表现出这个程序运行时缓存缺失率的大小，这也和复用距离本身的定义有关，复用距离低的请求相比复用距离高的请求更可能达到缓存命中，以 BFS 和 NN 程序为例进行说明（其他测试程序的结果见附录）。

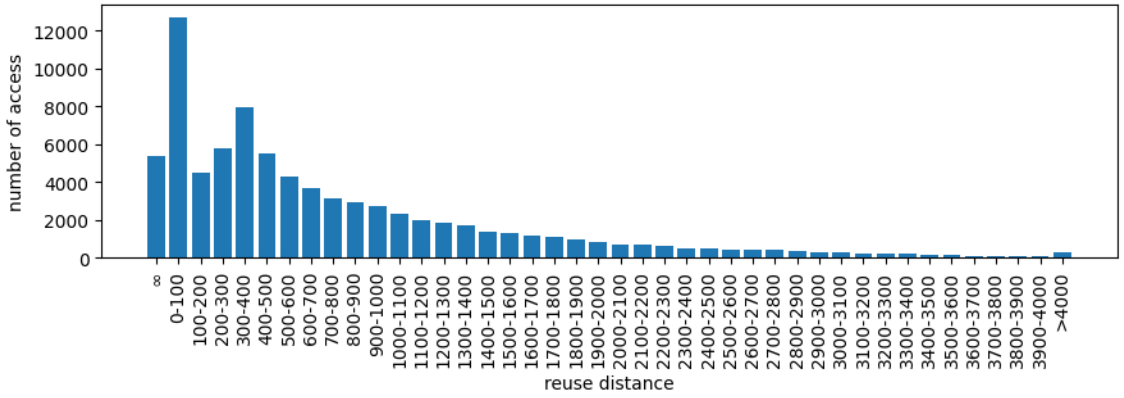


图 3.6 BFS 程序某一 SIMT 核心访存序列的复用距离分布

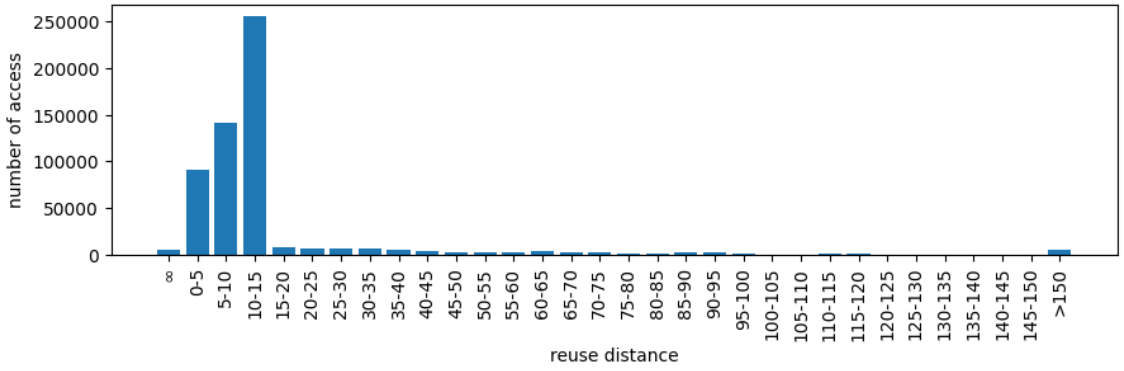


图 3.7 NN 程序某一 SIMT 核心访存序列的复用距离分布

根据图 3.6-7，可以发现 BFS 大部分请求的复用距离在 0-2000，而 NN 大部分请求的复用距离在 0-15，复用距离分布的显著差异体现出 BFS 高缓存缺失率和 NN 高缓存命中率的原因。

**缓存旁路分析:**现有的 GPU 体系 L1 缓存会缓存每一个访存请求中请求的数据，但是每个请求的复用距离都是不同的，在缓存中高复用距离的请求数据可能会逐出低复用距离的请求数据，这样就会导致缓存的块很难被命中，被逐出的缓存块会造成缺失，进而导致过高的缓存缺失率，有时缓存的去除反而会提升程序的性能，即缓存成为整个程序的瓶颈，这其实违背设置缓存的初衷，于是就提出缓存旁路。缓存旁路其实在 CPU 和 GPU 领域并不是陌生的概念[19][20]，但是哪些访存请求应该旁路缓存，如何让可以得到缓存旁路增益的程序保持增益效果、让

受到缓存旁路减益的程序恢复到没有缓存旁路时的性能是缓存旁路的主要问题。

图 3.9 是缓存旁路的示意图，其中经过旁路路径的请求会通过填充路径 1 返回，经过 L1D 路径的请求如果缓存缺失会通过填充路径 2 返回，其中是否旁路单元通过旁路仲裁器来实现，相联存储器的作用在硬件实现部分介绍。

首先使用 GPGPU-Sim 测试对于所有访存请求旁路 L1 缓存（bypassAll）的程序性能，归一化结果见图 3.8（控制单一变量为旁路策略，L1 缓存配置和 3.4 中 sector 缓存配置相同，其他配置和基准配置相同）。

可以从图 3.8 发现，大约一半程序因为缓存旁路受到性能减益，其中 BP 和 NN 受到的影响较大，KM 得到较大的性能增益，其他程序则受到了轻微的性能增益，由此可见 L1 缓存对于不同特点程序来说有着不同的影响。

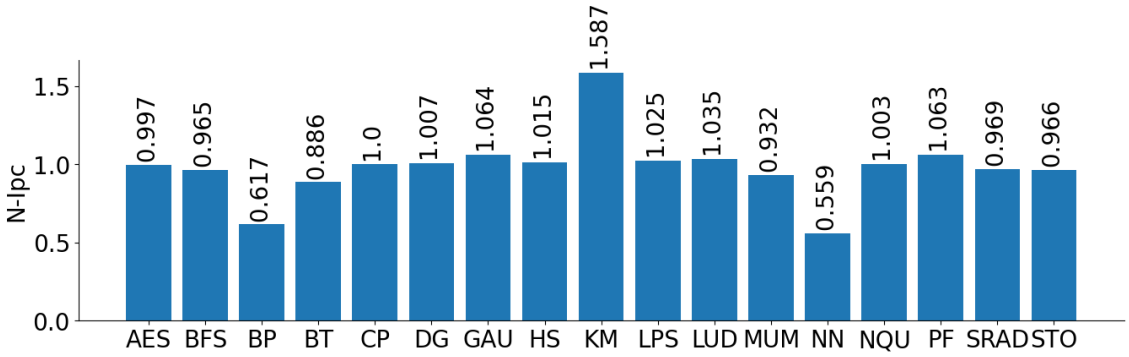


图 3.8 完全旁路 L1 缓存后测试程序的归一化 IPC

**SBP-split**：这时缓存旁路的问题又一次提出：哪些访存请求应该旁路缓存、让可以得到缓存旁路增益的程序保持增益效果、让受到缓存旁路减益的程序恢复到没有缓存旁路时的性能。于是想到给复用距离低的请求更大的缓存机会，而给复用距离高的请求更小的缓存机会。但是在程序执行过程中精确地计算出每个访存请求的复用距离代价很大，所以本文提出一种创新的指标X来衡量访存请求是否值得旁路，并提出了 SBP-split（选择性旁路策略），SBP-split 用到了参数 H。

| 指标 X 维护策略 |
| --- |
| 每一个存储块（存储块的大小和缓存块的大小一致）都对应一个指标X。<br>初始时$X = 0$；<br>当存储块在 L1 缓存命中时，$X = X + 1$；<br>当存储块在 L1 缓存缺失时，$X = X - 1$ 。 |

| SBP-split（selective bypass split） |
| --- |
| 如果X ≥ H，不对这些访存请求旁路；<br>如果X ＜ H，对这些访存请求旁路； |

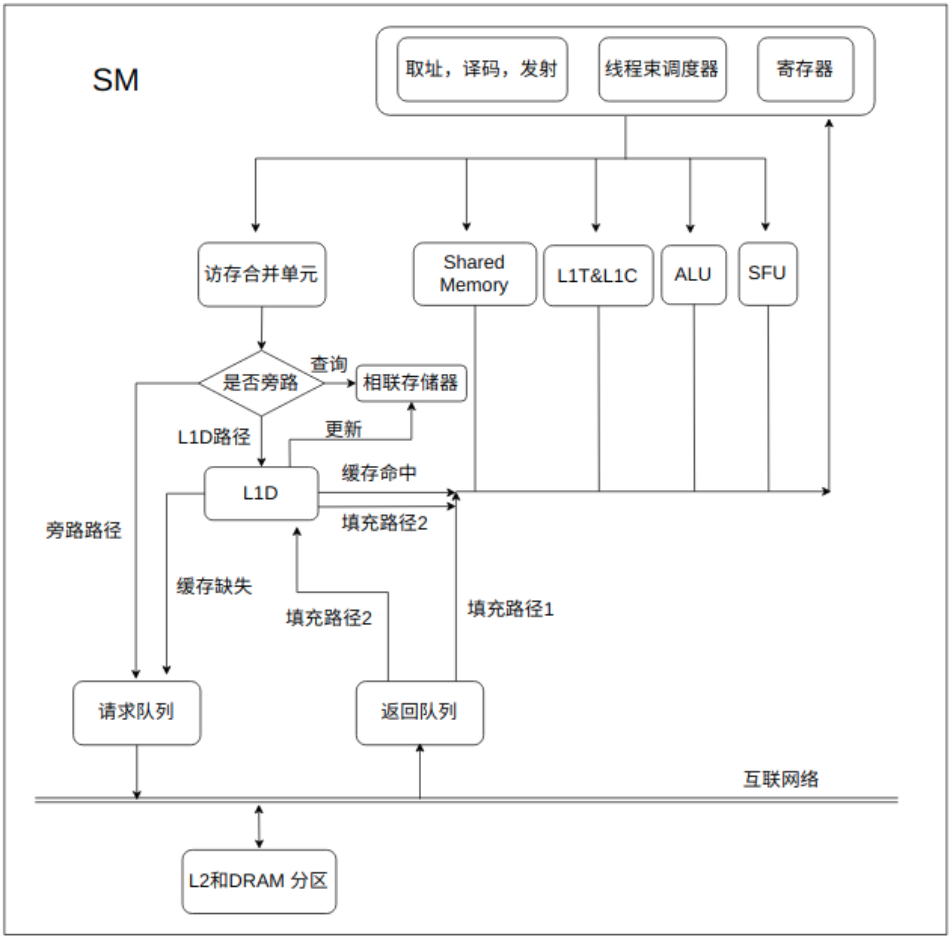


图 3.9 缓存旁路示意图

指标 X 等于存储块在 L1 缓存命中次数减去缺失次数，它表征着该存储块历史的表现，如果X ≥ H 则表明该存储块所对应的访存请求有着较小的复用距离，如果X ＜ H则表明该存储块对应的访存请求有着较大的复用距离，所以便有了 SBP-split 的执行策略。由于存储块在未被访问到时指标为 0，在第一次缓存后指标为-1，所以当 X 指标为-1 时一定不会旁路，即H ≤ −1。

使用 GPGPU-Sim 采取 SBP-split 策略，参数 H 取-4，对测试程序进行模拟后，归一化结果见表 3.4。（控制单一变量为旁路策略，L1 缓存配置和 3.4 中 sector 缓存配置相同，其他配置和基准配置相同）

**SBP-stage：**可以从表 3.4 看到，SBP-split 对于平均复用距离较高的 BFS 有着不错的性能增益，而对于其他平均复用距离相对较低的程序也基本保持了原来的性能，但是对于平均复用距离很低的 NN 却有着轻微的性能损失，所以本文提出了 SBP-stage，SBP-stage 用到了参数H（注意$H < 0$）。

| SBP-stage（selective bypass stage） |
| --- |
| 如果$X \geq 0$ ，不对这些访存请求旁路； |
| 如果$0 > X \geq H$，那么有 $\frac{H-X-1}{H}$ 概率不被旁路，$\frac{X+1}{H}$ 概率被旁路； |
| 如果$X < H$，对这些请求旁路。 |

SBP-stage 相比 SBP-split 的主要区别是它提供了一个阶梯概率缓冲区$0 > X \geq H$（参考图 3.10），在这个缓冲区内的访存请求有一定概率被旁路。它保证了被认为复用距离较大的请求也会有一定概率经过 L1 缓存，如果之后该存储块表现较好（有多次命中），X 指标可以转变为大于等于 0，即认为该存储块值得缓存到 L1 缓存中，SBP-stage 相比 SBP-split 有着更强的鲁棒性。

之后使用 GPGPU-Sim 采取 SBP-stage 策略，参数 H 取-10，对程序进行了测试，归一化结果见表 3.4（控制单一变量为旁路策略，L1 缓存配置和 3.3 节中 sector 缓存配置相同，其他配置和基准配置相同）。

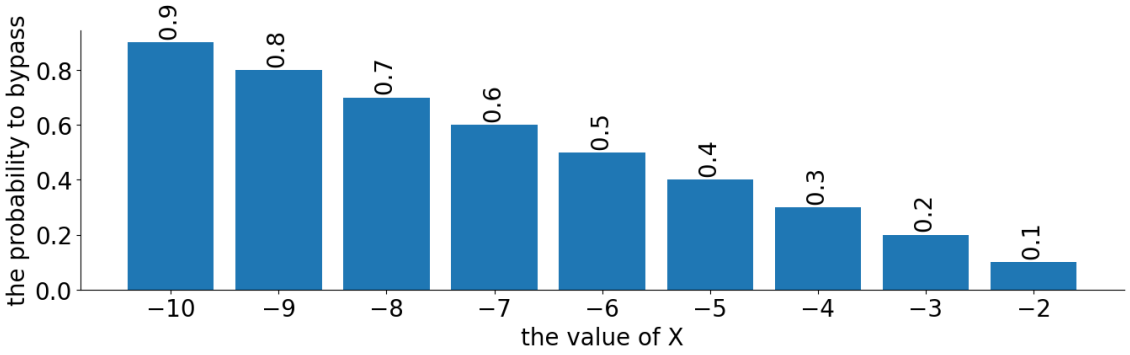


图 3.10 策略 SBP-stage 的阶梯概率缓冲区(H 取-10)

从表 3.4 可以看到，平均复用距离较高的程序 BFS 尽管增益效果相比 SBP-split 少，但是依然实现了性能提升，除了 BFS 的其他程序基本保持了原来的性能。这说明 SBP-stage 对于不同特性的程序（平均复用距离分布不同的程序）都有着很优秀的表现，体现了 SBP-stage 的通用性。

**SBP-LRU：**尽管 SBP-split 和 SBP-stage 已经实现了不错的效果，但是它们都需要提前给定参数 H 的值，但是每个程序的最优参数值都是不一样的（最优参数

值指在该参数下程序可以获得最佳性能），所以通过参考 LRU 的替换策略得出需要使用可以相互比较的指标来指导访存是否旁路，因此本文创新性地提出指标 Y 来决定访存是否旁路，并提出了 SBP-LRU。

| 指标 Y 维护策略 |
| --- |
| 每一个存储块（存储块的大小和缓存块的大小一致）都对应一个指标 Y。<br>初始时$Y = -1$；<br>当存储块需要被访问到时，Y 更新为当前时间（单位为时钟周期） |

| SBP-LRU（selective bypass LRU） |
| --- |
| 在需要决策时，先查询获得当前缓存中存储块指标 Y 的最小值，也就是获得最早的上一次访问时间$Y^*$<br>然后查询获得当前需要访存的存储块指标$Y'$<br>1.当$Y'$为-1 时，不对其进行旁路，否则执行步骤 2<br>2.当$Y' \geq Y^*$时，不对其进行旁路；当$Y' < Y^*$时，对其进行旁路。 |

SBP-LRU 是参考 LRU 进行的预测，LRU 认为当前时间距离上一次访问时间越长则该存储块将来一段时间的命中机会就越小，而 SBP-LRU 通过比较上一次访问时间的大小来判断这次访存是否值得旁路，参考值便是缓存中所有存储块最早的上一次访问时间，上一次访问时间离当前时间越近（指标 Y 越大），那么 SBP-LRU 便预测下一次访问时间距离当前这次访问的时间越近，那么该存储块就越可能在缓存中命中，因此它就不应该旁路。

之后使用 GPGPU-Sim 采取 SBP-LRU 策略，对程序进行了测试，归一化结果见表 3.4。（控制单一变量为旁路策略，L1 缓存配置和 3.3 节中 sector 缓存配置相同，其他配置和基准配置相同）。

从 SBP-LRU 的结果可以看出，它对 L1 缺失率很高的 KM 程序表现出了较大的性能提升，同时也保持了 BFS 的性能增益和其他不敏感程序的性能基本不变。

**硬件实现**：为了在硬件层面上，记录并查找存储块的指标 X 或 Y，每个私有 L1 缓存增设一个相联存储器，索引字段为存储块的标号（为存储块的首地址），存储的值为指标 X 或 Y，在相联存储器查找不到的存储块时认为其指标 X 为 0 或 Y 为-1，旁路仲裁器在决定是否旁路前查询相联存储器访存请求对应的存储块的指

标 X 或 Y，并根据旁路策略决定最终是否旁路，L1 缓存在命中或缺失后会向相联存储器发出请求来更新存储块的指标 X 或 Y。

**思考和分析**：选择性旁路策略究竟是依靠什么来获得性能提升呢？1. 高复用距离的访存请求旁路 L1 缓存，可以避免多余的访存请求经过 L1 缓存延时：从表 3.4 可以看出 BFS 有 54%的访存请求旁路 L1 缓存 2. 高复用距离的访存请求对应的存储块不会存储到 L1 缓存中，为低复用距离的访存请求对应的存储块提供更高的命中率：参考表 3.4，BFS 的 L1 缓存缺失率降低到了原来的 65.7%；综上所述，选择性旁路策略可以提高 L1 缓存命中次数，参考表 3.4，BFS 的 L1 缓存命中次数增加了 58.6%。

表 3.4 三种选择性旁路策略归一化指标

| | AES | **BFS** | BP | BT | CP | DG | GAU | HS | **KM** | LPS | LUD | MUM | NN | NQU | PF | SRAD | STO |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1t | 1.00 | 1.17 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.99 | 0.98 | 1.00 | 1.00 | 1.00 | 1.00 |
| 1e | 1.00 | 1.09 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.25 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.01 | 1.00 |
| 1u | 1.00 | 1.10 | 1.00 | 0.99 | 1.00 | 1.00 | 1.00 | 1.00 | 1.31 | 1.00 | 1.01 | 1.02 | 0.98 | 1.00 | 1.00 | 1.02 | 1.00 |
| 2t | 1.00 | 0.46 | 1.00 | 0.88 | 1.00 | 0.94 | 0.96 | 1.00 | 1.00 | 1.00 | 1.00 | 0.57 | 0.97 | 1.00 | 1.00 | 0.99 | 0.90 |
| 2e | 1.00 | 0.50 | 0.92 | 0.89 | 0.93 | 0.79 | 0.88 | 0.99 | 0.49 | 0.99 | 0.94 | 0.68 | 1.00 | 0.99 | 0.99 | 0.98 | 0.91 |
| 2u | 1.00 | 0.39 | 0.99 | 0.89 | 1.00 | 0.85 | 1.00 | 0.99 | 0.50 | 1.00 | 0.74 | 0.77 | 0.99 | 1.00 | 0.99 | 0.88 | 1.00 |
| 3t | 1.00 | 0.66 | 1.00 | 0.91 | 1.00 | 1.02 | 1.01 | 1.00 | 1.00 | 1.00 | 1.01 | 0.56 | 1.01 | 1.00 | 1.00 | 1.00 | 0.68 |
| 3e | 1.00 | 0.91 | 1.06 | 0.94 | 1.08 | 1.19 | 1.08 | 1.01 | 0.91 | 1.01 | 1.04 | 0.66 | 1.00 | 1.00 | 1.00 | 1.00 | 0.79 |
| 3u | 1.00 | 0.79 | 1.01 | 0.97 | 1.00 | 1.02 | 1.00 | 1.00 | 0.76 | 1.00 | 0.98 | 0.54 | 1.01 | 1.00 | 1.00 | 0.91 | 1.00 |
| 4t | 1.00 | 1.59 | 1.00 | 0.93 | 1.00 | 0.91 | 0.93 | 1.00 | 1.00 | 1.00 | 0.98 | 0.76 | 0.97 | 1.00 | 1.00 | 1.00 | 1.00 |
| 4e | 1.00 | 0.81 | 0.89 | 0.92 | 0.85 | 0.54 | 0.75 | 0.82 | 1.13 | 0.89 | 0.86 | 0.85 | 1.00 | 1.00 | 0.97 | 0.98 | 0.97 |
| 4u | 1.00 | 0.97 | 0.98 | 0.91 | 1.00 | 0.83 | 1.00 | 0.98 | 2.25 | 1.00 | 0.78 | 1.04 | 0.99 | 1.00 | 0.99 | 6.85 | 1.00 |

注：1.IPC 2. L1 访问次数 3. L1 缺失率 4. L1 命中次数；t 代表 split 策略；e 代表 stage 策略；u 代表 LRU 策略

## 3.5 测试程序和实验建立

为了研究不同程序的特点，在 GPGPU-Sim 上对 CUDA 程序进行基准测试。

测试程序参考[9][15][16]，最终选择测试的程序如下：

AES 加密（AES）：AES 由 Manavski 开发，以 CUDA 程序的形式实现了高级加密标准来进行加密和解密文件。其中常量被存储在常量存储中，秘钥被存储在纹理存储中，输入的数据在共享存储中处理。在测试中使用 128 位对 256KB 的图片进行加密操作。

图算法广度优先搜索（BFS）：BFS 由 Harish 和 Narayanan 开发，在图上执行广度优先算法。由于图上的每一个节点被映射到一个线程，程序的并行性会随着输入图的大小变化。

库伦电势（CP）：CP 是由伊利诺伊大学香槟分校 IMPACT 研究小组开发的 Parboil Benchmark 套件的一部分。CP 在分子动力学领域是有用的。循环被手动展开以减少循环开销，点电荷数据被存储在常量内存中以利用缓存。CP 经过了大量优化(与 CPU 版本相比，它可以实现 647 倍加速)。本测试在网格大小为 256X256 的网格上模拟 200 个原子。

gpuDG（DG）：DG 是一个不连续的 Galerkin 时域求解器，用于电磁领域从三维物体计算雷达散射和分析波导，粒子加速器和电磁兼容性。数据从纹理内存加载到共享内存中。本测试使用多项式阶数 N=6 的 3D 版本，并将时间步长减少到 2，以减少模拟时间。

三维拉普拉斯求解器(LPS)：拉普拉斯是一个高度并行的金融应用。除了使用共享内存外，应用程序开发人员还注意确保合并全局内存访问。本测试在 100x100x100 网格上运行一次迭代。

MUMmerGPU (MUM)：MUMmerGPU 是一个并行的两两局部序列比对程序，它将由标准 DNA 核苷酸(A、C、T、G)组成的查询字符串匹配到一个参考字符串，用于基因分型、基因组重测序和宏基因组学等目的。参考字符串作为后缀树存储在纹理内存中，并使用纹理缓存来进行 2 维局部优化。然而，树的大小意味着高的缓存 miss 率，导致 MUM 受到内存带宽的限制。由于每个线程执行自己的查询，搜索算法的性质也使性能容易受到分支差异的影响。本测试以炭疽芽孢杆菌 Ames 基因组的前 140,000 个字符作为参考字符串，以全基因组为种子随机生成 50,000 个 25 个字符的查询。

神经网络(NN)：NN 使用一种传统的神经网络来识别手写数字。预先确定的神经元权重随输入数字一起载入全局存储器。本测试模拟了从数据库中识别 28 个手写数字。

N 皇后问题求解(NQU) ：NQU 解决一个经典的谜题，即在 NxN 的棋盘上放置 N 个皇后，使得没有皇后可以捕获另一个。它使用一个简单的回溯算法来尝试确定所有可能的解决方案。本测试模拟 N = 10。

StoreGPU (STO)： STO 是一个库，用于加速为中间件设计基于哈希的原语。本测试选择在大小为 192KB 的输入文件上使用 MD5 算法的滑动窗口实现。

反向传播（BP）：反向传播是机器学习的训练神经网路的算法，BP 实现了神经网络的前向传播和反向传播。

Speckle Reducing Anisotropic Diffusion (SRAD)：SRAD 是一个基于偏微分方程的扩散算法并且用于在不牺牲图像重要特征的情况下移去图像中的斑点。SRAD 广泛应用于超声和雷达图像处理。

HotSpot（HS）：HS 是一个热模拟工具，用于根据架构图和模拟功率测量估计处理器温度。HS 包括二维瞬态热模拟内核，该内核迭代求解一系列块温度微分方程。

B+tree（BT）：BT 有许多维护数据库和进程查询的内部命令，它实现了建立 B+树、对节点的访问、输出叶子节点、进行范围查询等一系列操作。

Gaussian（GAU）：GAU 实现了线性代数的高斯消元法。GAU 逐行计算结果，求解线性系统中的所有变量。算法必须在迭代之间同步，但每次迭代中计算的值可以并行计算。

LUD（LUD）：LU 分解是一种计算线性方程组解的算法。LUD 实现了对矩阵的 LU 分解，即将一个矩阵分解为一个下三角矩阵和一个上三角矩阵的乘积。

PathFinder（PF）：PF 使用动态规划在二维网格上找到一条从下行到上行具有最小累积权重的路径，其中路径的每一步都是直线前进或对角前进。它逐行迭代，每个节点在前一行中选择一个累积权重最小的相邻节点，并将自己的权重加到和中。

Kmeans（KM）：Kmeans 是一种广泛应用于数据挖掘和其他领域的聚类算法。其中包括两个 kernel 函数，一个用于数组从行主序变换为列主序，另外一个用于聚类算法，为了体现缓存管理的重要性，只测试访存密集的前一个 kernel 函数。

表 3.5 GPGPU-Sim 基准配置

| | |
|---|---|
| SIMT 核心 (SM) | 15 个核心；SIMD 宽度=32，1.4GHz，5 阶段流水线 |
| 每个 SM 最多有 | 1536 个线程；32768 个寄存器；48 个线程束（warp） |
| L1 缓存 / SM | 128 字节大小的缓存块；32 组 4 路相连 sector；1 个时钟周期的命中延迟 |
| 共享存储 / SM | 48KB；32 个存储体；3 个时钟周期延迟；每个时钟周期可以访问 1 次 |
| L2 缓存（总共） | 768KB；每个存储体 128KB；6 个存储体；128 字节大小的缓存块；16 路相连 |
| DRAM | 6 个访存通道；带宽：每个时钟周期 48 字节，1.4GHz |
| DRAM 调度队列 | 大小=16；调度策略=FR-FCFS |
| warp 调度策略 | Greedy then oldest (GTO) |

GPGPU-Sim 配置基于 NVIDIA Fermi GPU，具体参考表 3.5，测试结果见图 3.11-16，称该结果为基准指标。

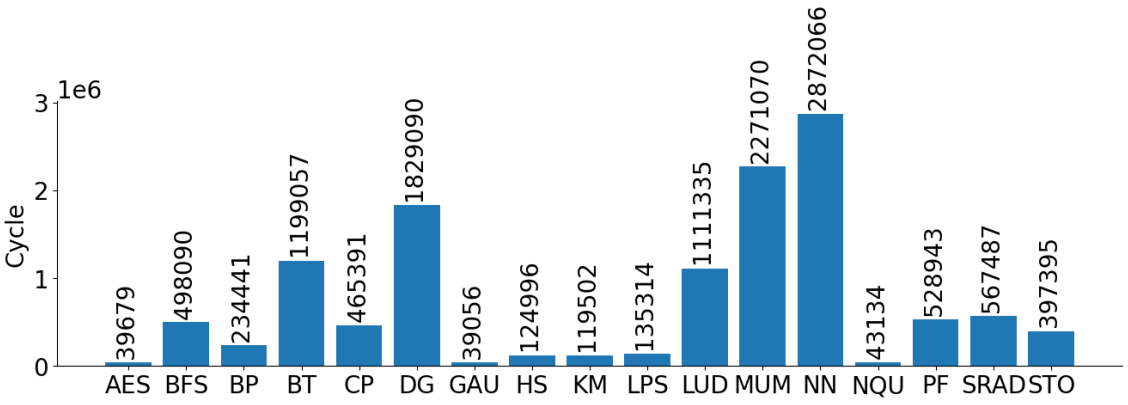


图 3.11 基准配置下测试程序的运行周期数

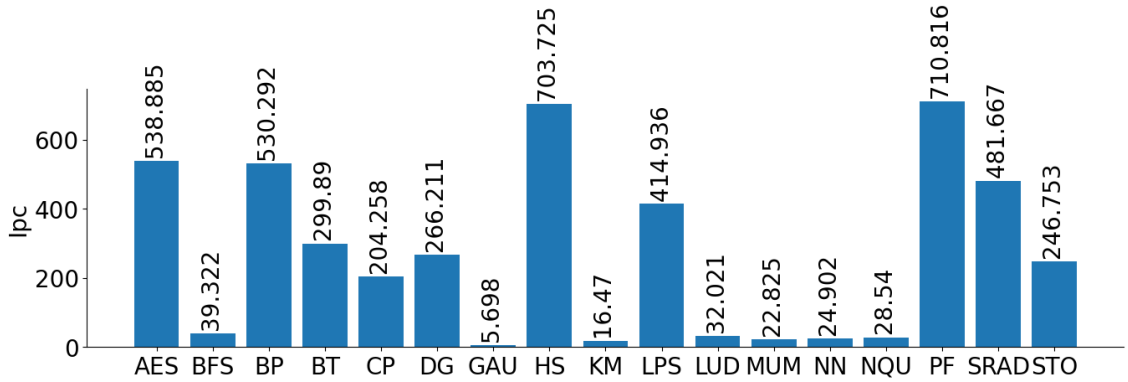


图 3.12 基准配置下测试程序的 IPC

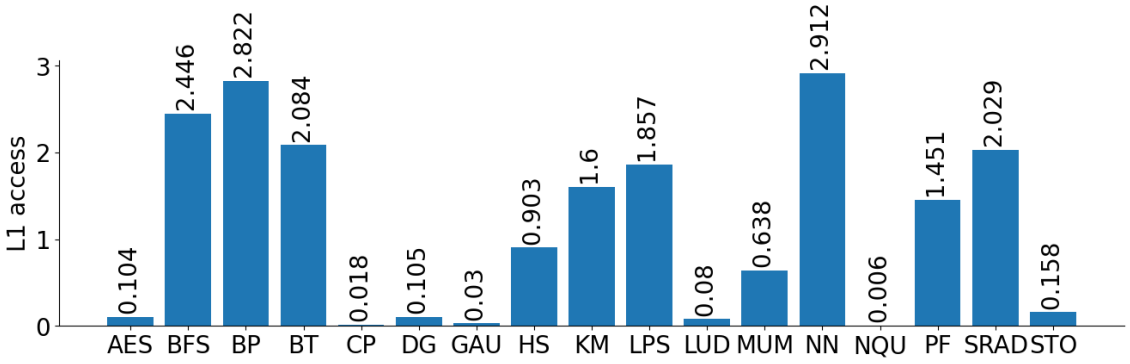


图 3.13 基准配置下测试程序的 L1 缓存平均访问次数

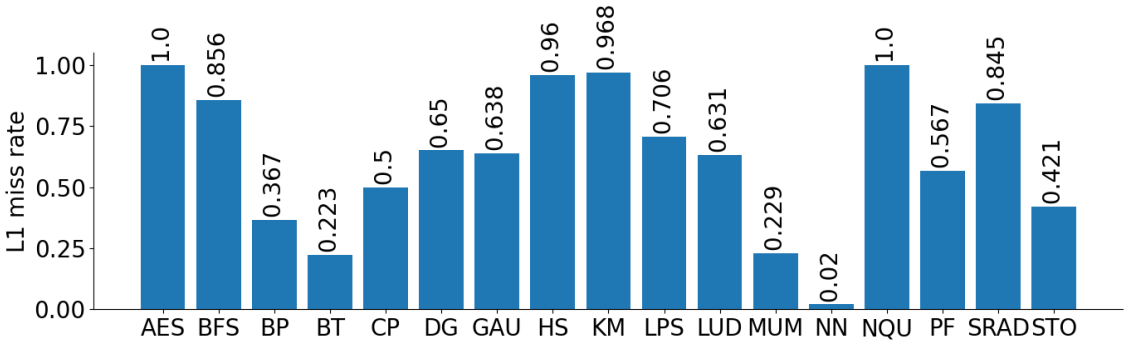


图 3.14 基准配置下测试程序的 L1 缓存缺失率

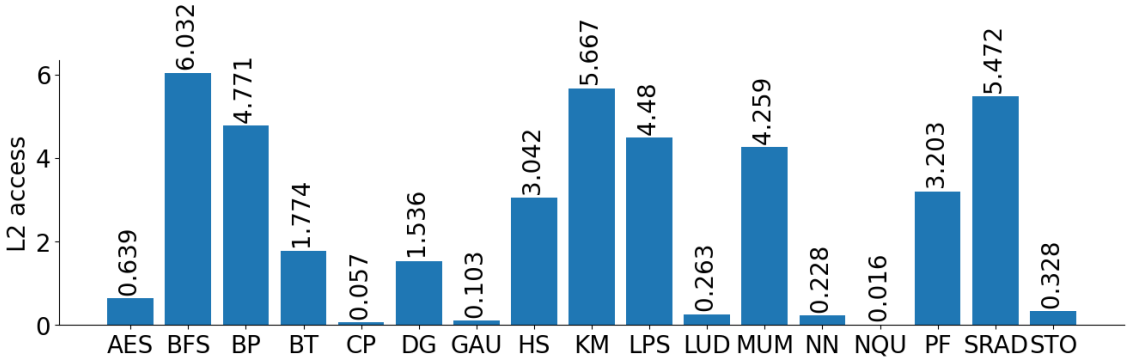


图 3.15 基准配置下测试程序的 L2 缓存平均访问次数

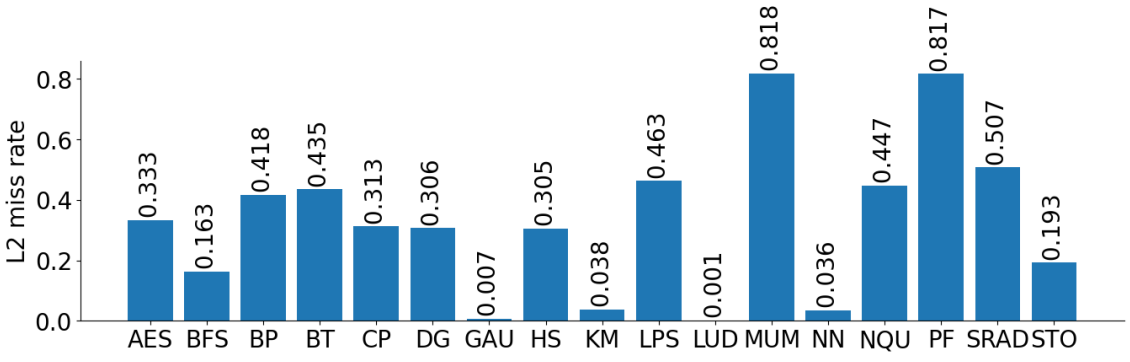


图 3.16 基准配置下测试程序的 L2 缓存缺失率

从测试结果可以看出每个程序对于访存的需求都有着很大的差别，其中 BFS、BP、BT、NN、SRAD 程序对于 L1 缓存平均访问次数较多（大于 2），HS、MUM、LPS、PF 对于 L1 缓存平均访问次数较少（小于 2 大于 0.2），AES、CP、DG、GAU、LUD、NQU、STO 对于 L1 缓存平均访问次数很少（小于 0.5），因此称 BFS、BP、BT、NN、SRAD 为访存密集型程序，称 AES、CP、DG、GAU、LUD、NQU、STO 为计算密集型程序，称 HS、MUM、LPS、PF 为一般程序。访存密集型程序性能比计算密集型程序受缓存缺失率的影响更大。

为了研究缓存大小对 IPC 的影响，在修改了 L1 缓存的配置后重新测试，改动是将组相联的 L1 缓存从 32 组 4 路变为 32 组 320 路，增加到原来的 80 倍 L1 缓存容量。

配置改变后测试的结果见表 3.5，所有指标为与基准指标相比归一化之后的结果（即更改配置后的指标与基准指标之比）

表 3.5 更改配置后归一化指标

| | AES | BFS | BP | BT | CP | DG | GAU | HS | KM | LPS | LUD | MUM | NN | NQU | PF | SRAD | STO |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1.00 | 0.34 | 1.00 | 0.93 | 1.00 | 1.00 | 1.00 | 1.00 | 0.20 | 1.00 | 1.00 | 0.94 | 0.95 | 1.00 | 1.00 | 0.93 | 1.00 |
| 2 | 1.00 | 2.91 | 1.00 | 1.08 | 1.00 | 1.00 | 1.00 | 1.00 | 4.97 | 1.00 | 1.00 | 1.07 | 1.05 | 1.00 | 1.00 | 1.07 | 1.00 |
| 3 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 4 | 1.00 | 0.45 | 0.99 | 0.69 | 1.00 | 0.96 | 1.00 | 0.96 | 0.14 | 1.00 | 0.93 | 0.50 | 0.56 | 1.00 | 1.00 | 0.90 | 1.00 |
| 5 | 1.00 | 0.24 | 1.00 | 0.68 | 1.00 | 0.99 | 1.00 | 0.96 | 0.06 | 1.00 | 0.96 | 0.94 | 0.55 | 1.00 | 1.00 | 0.96 | 1.00 |
| 6 | 1.00 | 2.14 | 1.00 | 1.43 | 1.00 | 1.01 | 1.00 | 1.04 | 13.69 | 1.00 | 1.05 | 0.99 | 1.81 | 1.00 | 1.00 | 1.04 | 1.00 |

注：1. cycle 2. IPC 3. L1 访问次数 4. L1 缺失率 5. L2 访问次数 6. L2 缺失率

从表 3.5 可以看出，KM 和 BFS 在缓存容量增加后 IPC 分别提升了大约 4 倍和 2 倍,，BT、MUM、NN 和 SRAD 则小幅提升，其他测试程序 IPC 基本没变。从图 3.10 可以看出，KM、BT、BFS、MUM 和 NN 的 L1 缓存缺失率大幅减少，HS、LUD、MUM 和 SRAD 的 L1 缓存缺失率小幅下降，其他测试程序的 L1 缓存缺失率基本没变。可以看到尽管 BT、BFS、MUM 和 NN 的 L1 缓存缺失率均大幅减少，但是 BT、MUM 和 NN 程序的 IPC 却提升并不大，这首先与 BT、MUM 和

NN 原本的 L1 缓存缺失率较低有关，尤其是 NN 的缓存缺失率原本就已经很低了，此时访存已经不再成为其瓶颈，所以缓存缺失率的降低并没有带来程序 IPC 的显著提升。根据表 3.5 可以得出，L1 缓存缺失率下降的程序，L2 缓存的访问次数均呈现不同程度的下降，这再次证明了在两级缓存体系中，第一级缓存缺失的代价远远大于第二级缓存缺失的代价，这也是很好理解的。

尽管缓存容量的增加可能带来访存集中型程序 IPC 的提升，但是由于 GPU 本身架构的特点，片上空间极其宝贵，不太可能牺牲计算资源来换取缓存资源提升。

## 3.7 本章小结

本章首先提出用到的程序分析指标并基于 GPGPU-Sim 仿真器建立实验，之后通过程序模拟分析了共享 L1 缓存架构对于缓存缺失率的影响，以及缓存块基本单元大小对于访存集中程序的性能影响，然后基于仿真器获得程序的访存序列，通过访存序列计算得到程序访存的复用距离分布，并提出了三种选择性旁路策略，以及针对其在测试程序上的表现进行了实验分析，最后介绍了测试程序和实验建立的过程。

# 第四章 总结与展望

## 4.1 工作总结

本文首先重新思考 L1 缓存体系，并分析共享 L1 缓存架构，它解决私有 L1 缓存数据重复的问题，提高 L1 缓存的利用率，进而降低 L1 缓存的缺失率，提升访存密集型程序的性能；其次分析 sector 缓存和非 sector 缓存的组织特点，得出影响这两个缓存组织的重要因素是 cache line 大小，并根据访存特点将程序划分为访存集中型和访存分散型，其中访存分散的程序更适合较小的 cache line 大小；最后提出三种选择性旁路策略，SBP-split 和 SBP-stage 使用指标 X 来衡量访存请求是否应该旁路 L1 缓存，其中 SBP-stage 相比 SBP-split 有更好的通用性和鲁棒性，并提出 SBP-LRU，它不需要提前给定参数来决策，而是通过比较指标 Y 来进行决策，这样避免不同程序的最优参数不同而难以确定参数值的问题，上述三种旁路策略均使得平均复用距离较大的程序性能得到提升，并且基本保持平均复用距离较小的程序性能，充分体现 SBP 的泛用性。

## 4.2 国内外相关工作对比与分析

私有 L1 缓存存在数据重复的问题也在[5][6][7]得到了体现，然而[5][6][7]对于该问题的解决方法却不尽相同。[5]提出了可以通过获取其他核心的私有 L1 缓存数据来减少访存时间；[7]提出了每个核心的私有 L1 缓存可以通过片上互联网络进行相互数据传输，每个私有 L1 缓存负责一部分地址的数据存储，私有 L1 缓存之间负责的地址范围不重复；[6]提出了可以将几个私有 L1 缓存联合成为一个 DC-L1 缓存，DC-L1 缓存通过互联网络和核心进行数据传输。其中[6]提出的共享 DC-L1 缓存架构和本文中分析的共享 L1 缓存架构很像，其中的区别是有无分体存储，[6]将本文的一个共享 L1 缓存体分为多个小缓存体，每个缓存体独立地负责不重复且全地址覆盖的部分地址缓存，这样的好处是可以增大 L1 缓存带宽，缺点是提高了互联网络的功耗和面积。

GPU 缓存旁路的方法在[19][20]中也有提出。其中[19]提出的缓存旁路策略为：标

志存储中有一系列的数据块地址和其被访问次数，当某个数据块的访问次数大于阈值后将该数据块放到 L1 缓存中存储，值得一提的是标志存储中只有固定数量的条目，也就是说对于某个数据块不在标志存储中，会使用替换算法替换旧的条目，这样就造成了标志存储中只有部分先验信息，因为会丢失被替换出的数据块的先验信息，而且该策略没有使用到数据块缺失的历史信息，且 L1 缓存需要冷启动（即经过一段时间访存后才会有数据缓存）。而[20]提出的缓存旁路策略为所有访存请求依然会经过 L1 缓存，只是会根据 PC 指针来预测这些访存请求的存储块是否值得存储到 L1 缓存中，这样的旁路策略依然要付出 L1 缓存的延时。

[21]提出了一种高性能缓存替换算法，深刻地分析了 LRU 替换算法和复用距离的关系，同样也强调了复用距离对于缓存替换算法的指导意义，但是和[19]相似，[21]的缓存替换策略存储的先验信息仅为在缓存中的存储块的先验信息，而丢失了没有在缓存中存储块先验信息，同样并没有利用存储块在缓存缺失的信息，只是利用了命中信息，这样会导致先验知识的不充分，进而会使得预测偏差较大。

## 4.3 工作展望

未来首先会思考如何针对不同访存特点的程序进行缓存块基本大小的划分，并在 GPGPU-Sim 实现相关算法并进行实验验证。

其次，会将本文提出的选择性旁路算法应用到更多的测试程序和不同的输入数据上，以测试和验证该算法的鲁棒性。

最后，会思考是否可以在编译程序时分析得出需要访存的数据，之后在程序运行前预取数据或者是动态地预取数据到 GPU 缓存部分，这样可以实现计算和访存的并行执行，进而提高程序性能。

# 参考文献

[1] W. A. Wulf and S. A. McKee, “Hitting the Memory Wall: Implications of the Obvious,” ACM SIGARCH Computer Architecture News, 1995.

[2] NVIDIA Fermi Compute Architecture Whitepaper https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[3] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2010.

[4] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," in IBM Systems Journal,1966.

[5] M. A. Ibrahim, H. Liu, O. Kayiran, and A. Jog, “Analyzing and Leveraging Remote-core Bandwidth for Enhanced Performance in GPUs,” in Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT), 2019.

[6] Ibrahim, Mohamed Assem , et al. "Analyzing and Leveraging Decoupled L1 Caches in GPUs." 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2021.

[7] Ibrahim, Mohamed Assem , et al. "Analyzing and Leveraging Shared L1 Caches in GPUs." International Conference on Parallel Architectures and Compilation Techniques (PACT), 2020.

[8] NVIDIA CUDA C programming guide https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[9] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, “Analyzing CUDA Workloads Using a Detailed GPU Simulator,” in Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), 2009.

[10] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, Timothy G Rogers.“Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling.”In proceedings of the 47th IEEE/ACM International Symposium on Computer Architecture (ISCA),May 29 - June 3, 2020.

[11] Rothman, J. B. , and A. J. Smith . "Sector cache design and performance." Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings. 8th International

Symposium on IEEE(MASCOTS), 2000.

[12] C. Nugteren, G. -J. van den Braak, H. Corporaal and H. Bal, "A detailed GPU cache model based on reuse distance theory," 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), 2014.

[13] Nugteren, C. , et al. "A detailed GPU cache model based on reuse distance theory." IEEE International Symposium on High Performance Computer Architecture (HPCA), 2014.

[14] K. Beyls and E. D’ Hollander. "Reuse Distance as a Metric for Cache Behavior." Conference on Parallel and Distributed Computing and Systems(IASTED), 2001.

[15] S. Che et al., "Rodinia: A benchmark suite for heterogeneous computing," 2009 IEEE International Symposium on Workload Characterization (IISWC), 2009.

[16] M. Burtscher, R. Nasre and K. Pingali, "A quantitative study of irregular programs on GPUs," 2012 IEEE International Symposium on Workload Characterization (IISWC), 2012.

[17] Li, B. , et al. "Elastic-Cache: GPU Cache Architecture for Efficient Fine- and Coarse-Grained Cache-Line Management." IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2017.

[18] Veidenbaum, Alexander V. , et al. "Adapting Cache Line Size to Application Behavior." International Conference on Supercomputing DBLP, 1999:145-154.

[19] Li, Chao , et al. "Locality-Driven Dynamic GPU Cache Bypassing." Acm on International Conference on Supercomputing (ICS), 2015.

[20] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jiménez. "Adaptive GPU cache bypassing." In Proceedings of the 8th Workshop on General Purpose Processing using GPUs (GPGPU-8),2015.

[21] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)." In Proc. Of the Int’l Symp. on Computer Architecture (ISCA), 2010.

# 附录

## 附录 A 测试程序访存序列的复用距离分布

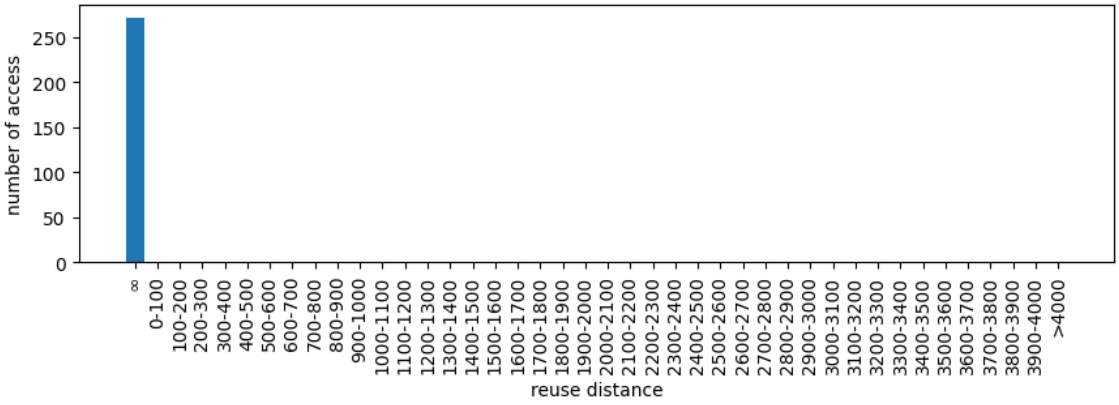


图 A1 AES

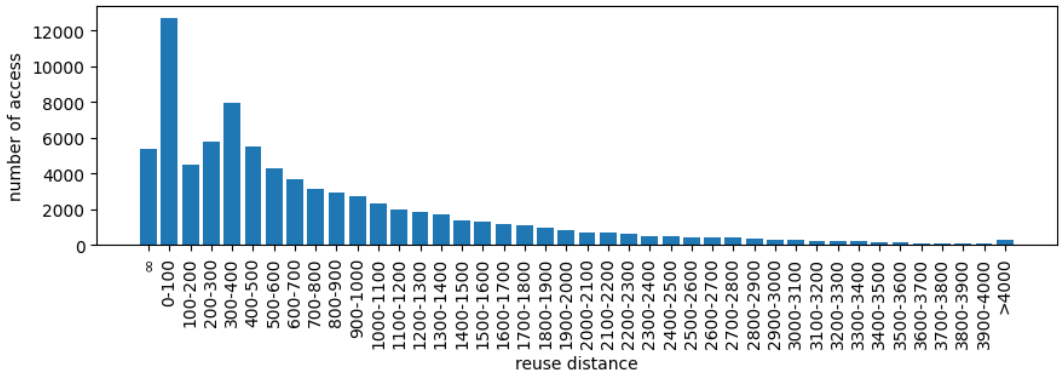


图 A2 BFS

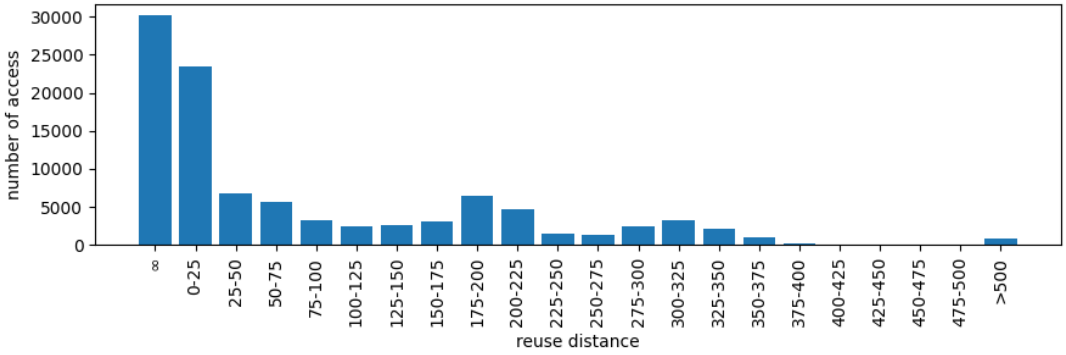


图 A3 BP

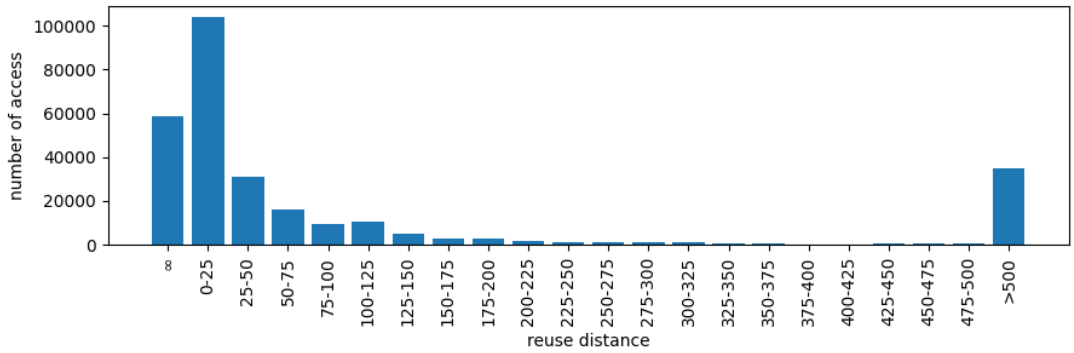


图 A4 BT

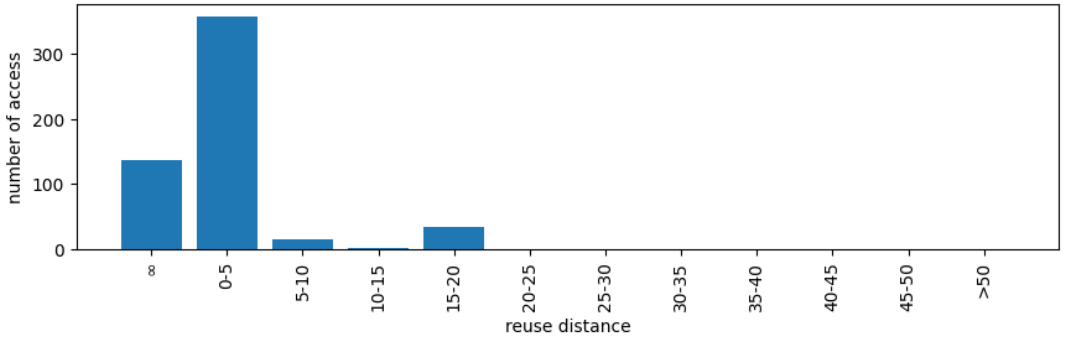


图 A5 CP

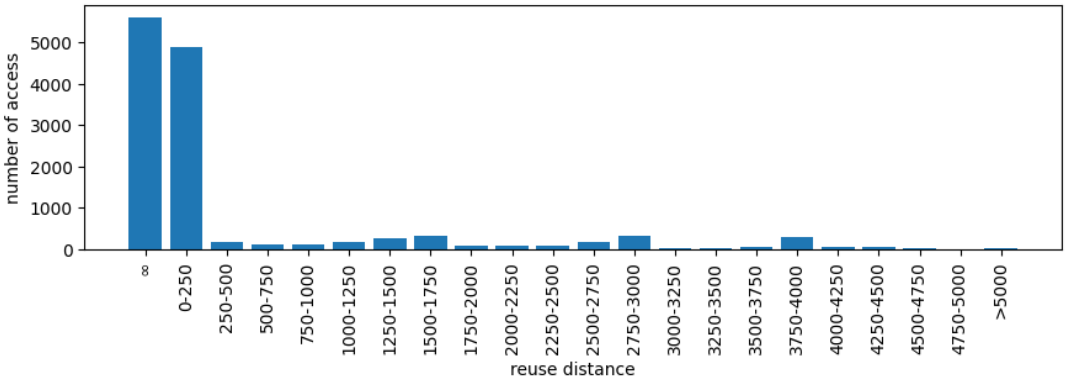


图 A6 DG

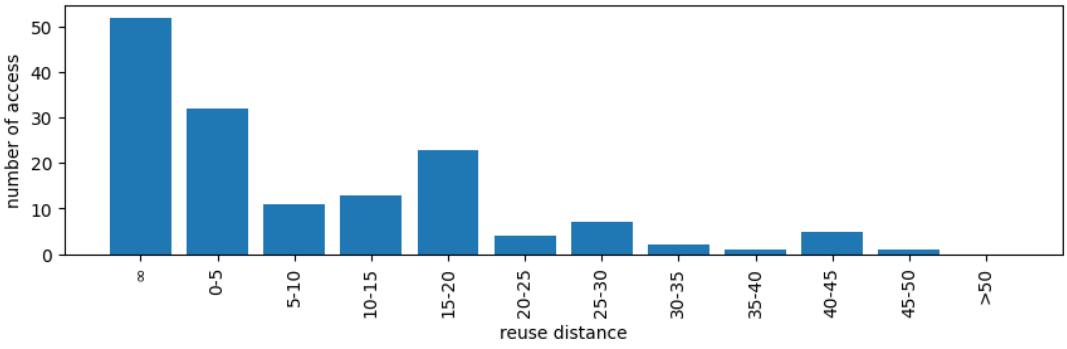


图 A7 GAU

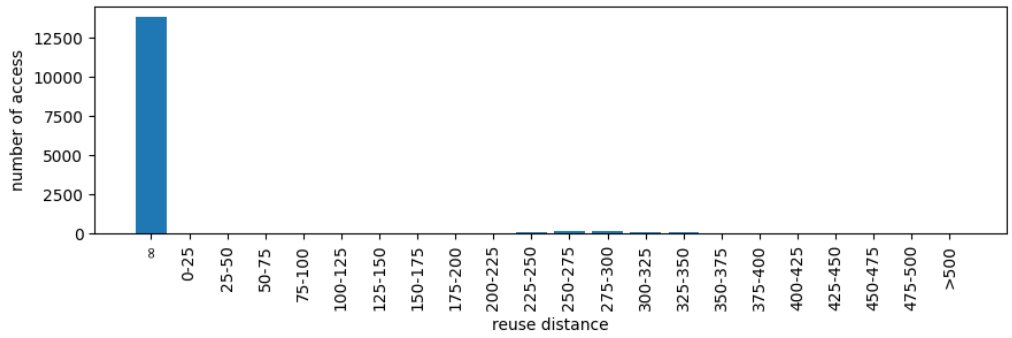


图 A8 HS

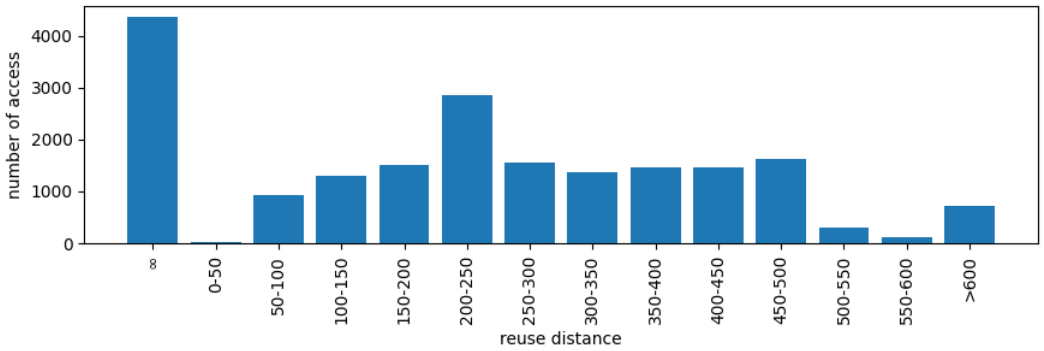


图 A9 KM

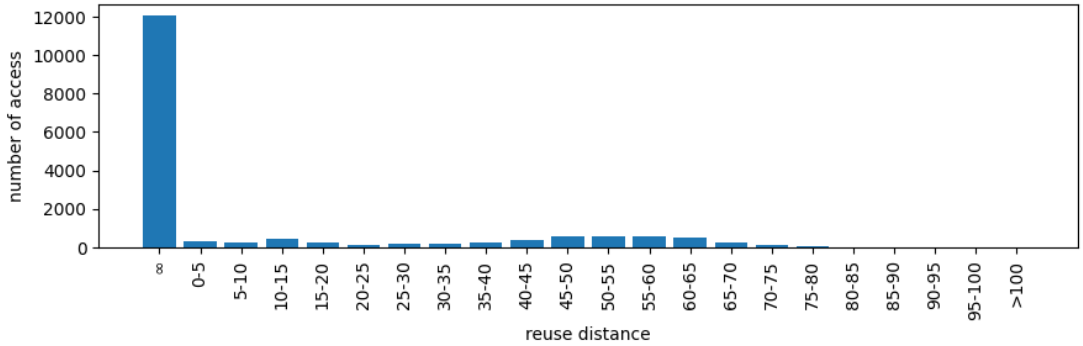


图 A10 LPS

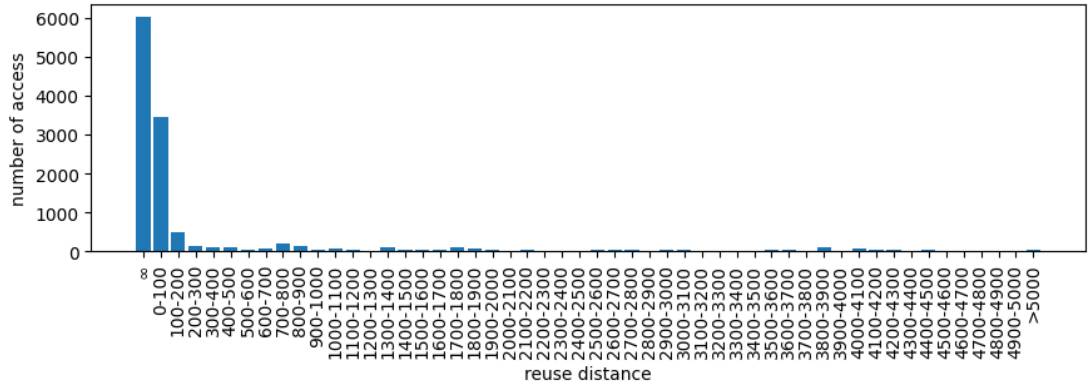


图 A11 LUD

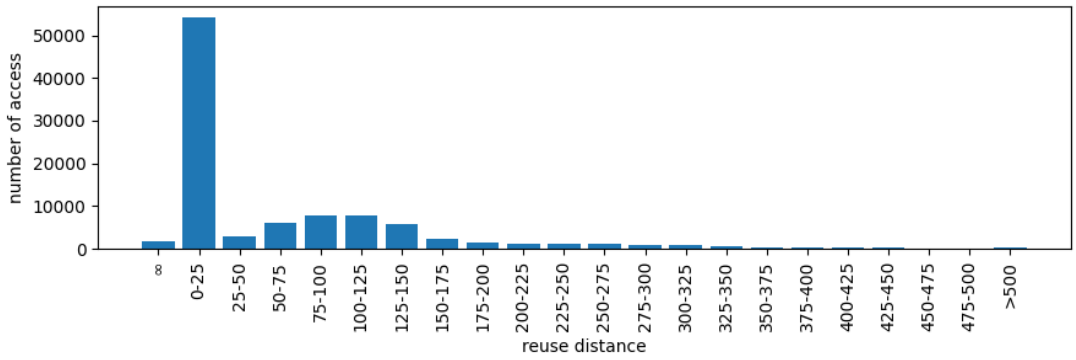


图 A12 MUM

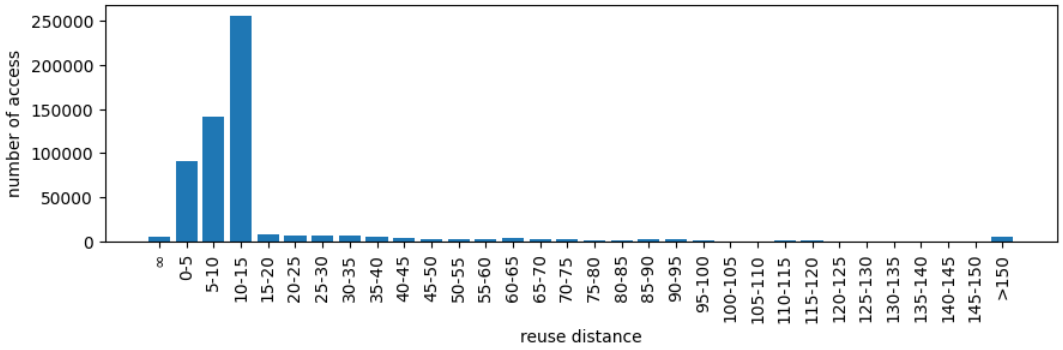


图 A13 NN

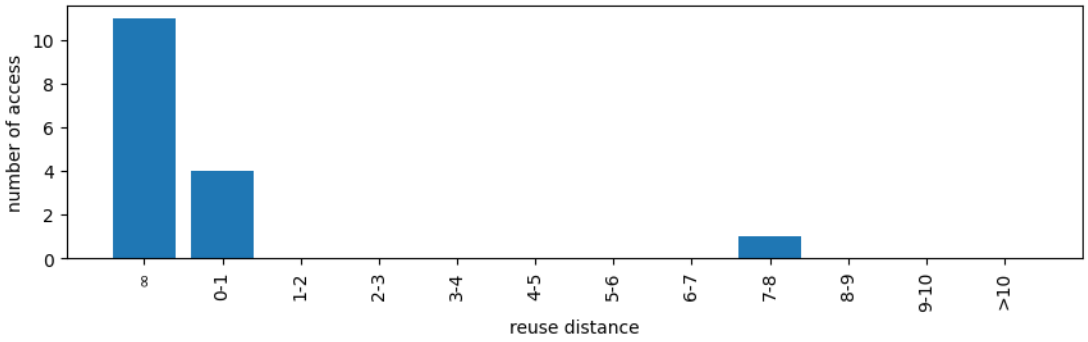


图 A14 NQU

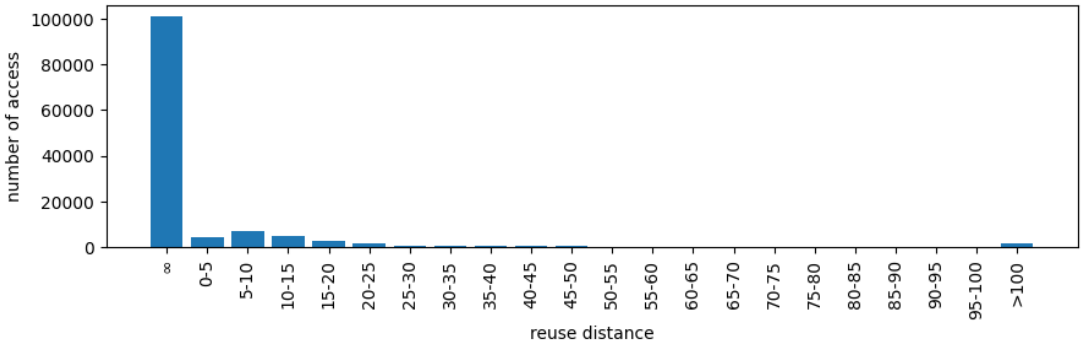


图 A15 PF

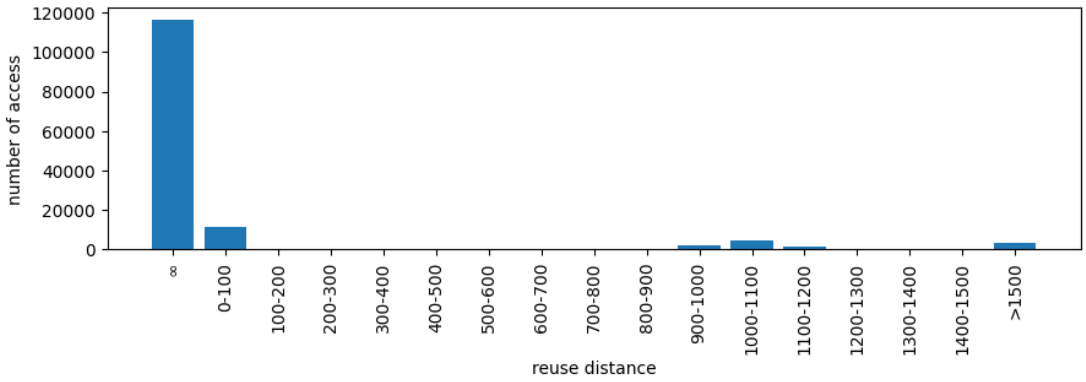


图 A16 SRAD

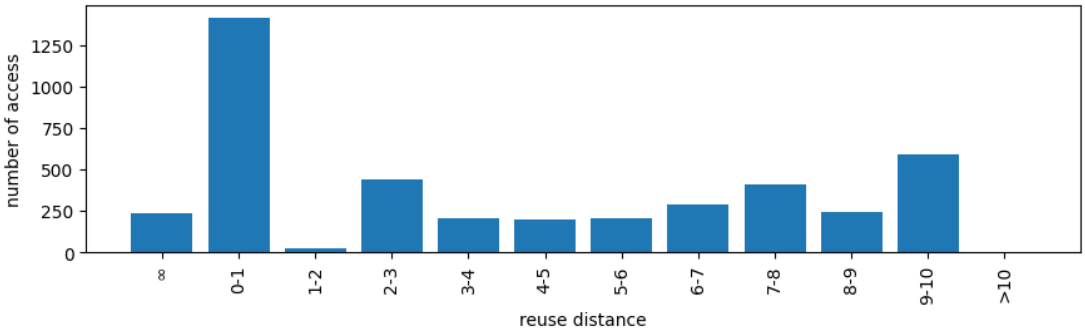


图 A17 STO

# 附录 B 测试程序不同 SIMT 核心访问到的缓存块数量占比

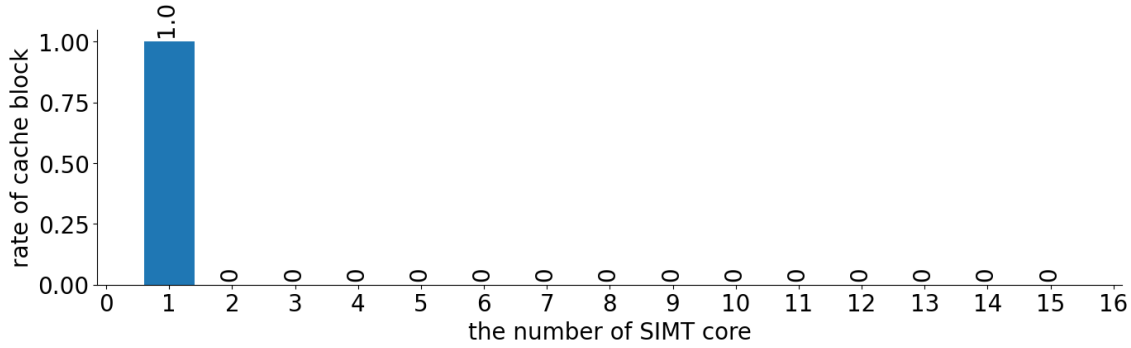


图 B1 AES

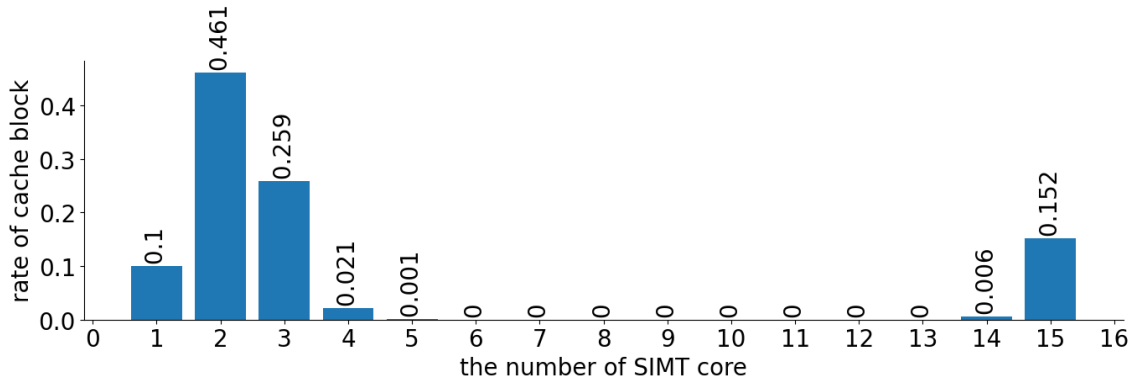


图 B2 BFS

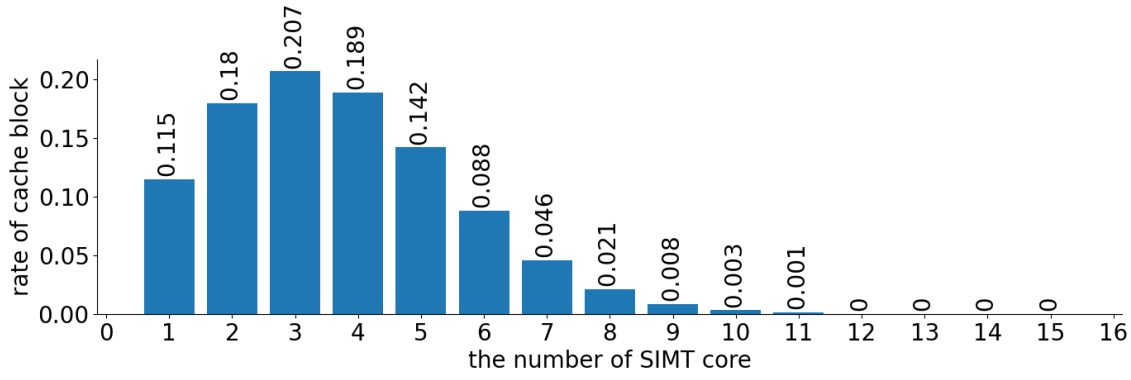


图 B3 BT

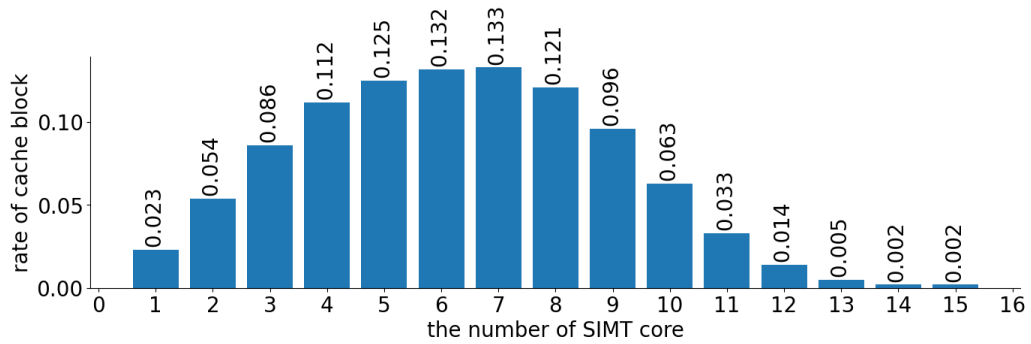


图 B4 BP

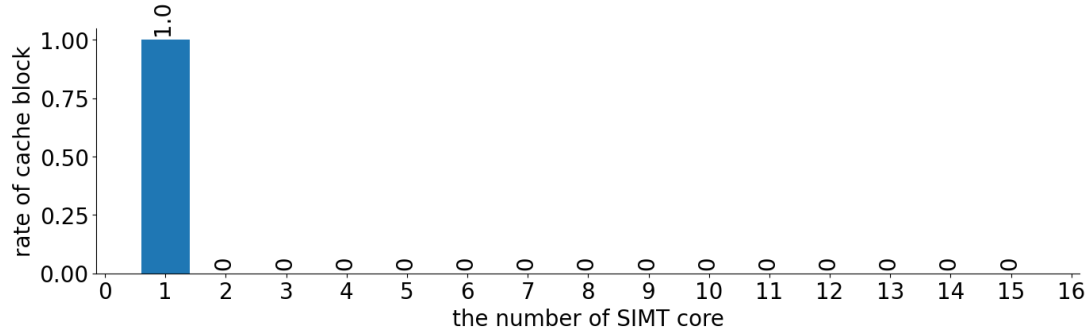


图 B5 CP

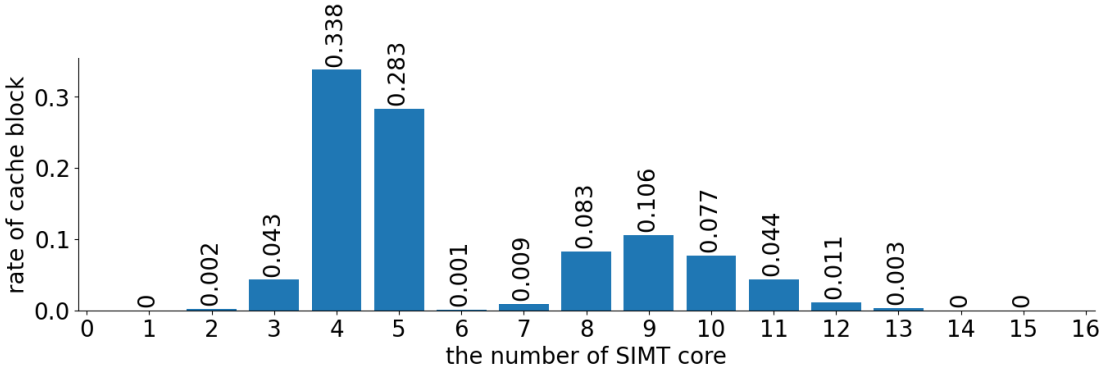


图 B6 DG

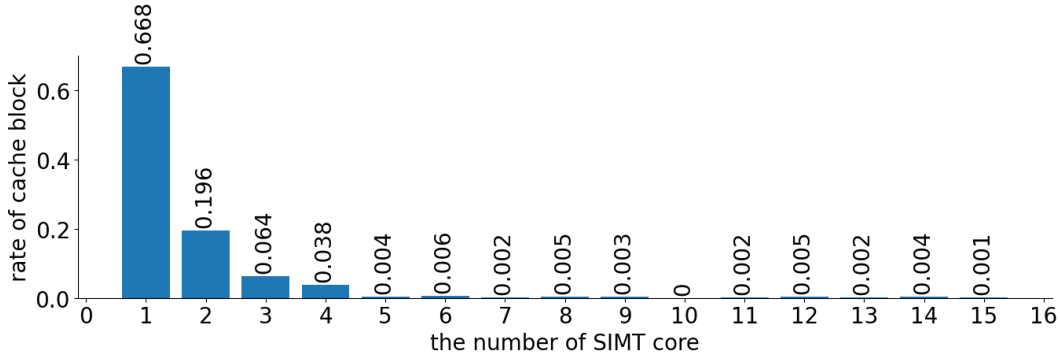


图 B7 GAU

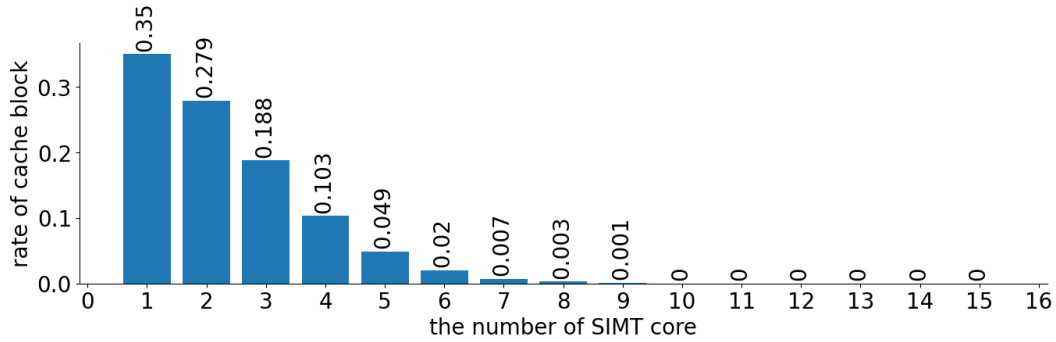


图 B8 HS

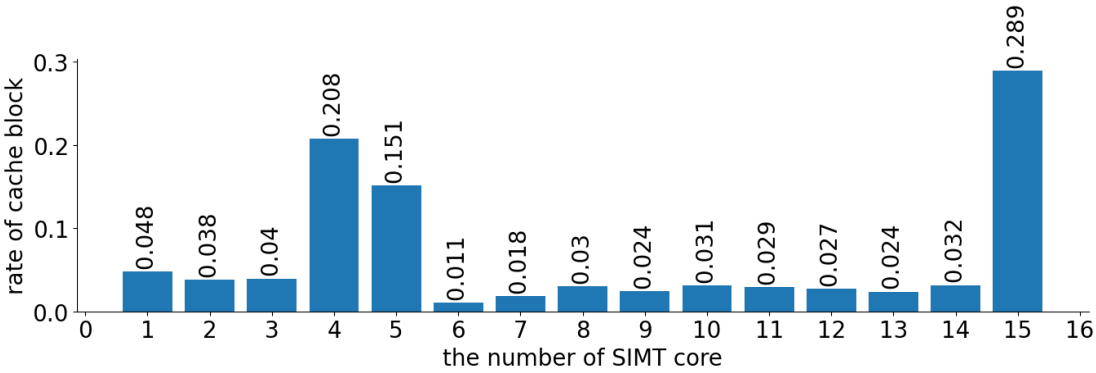


图 B9 KM

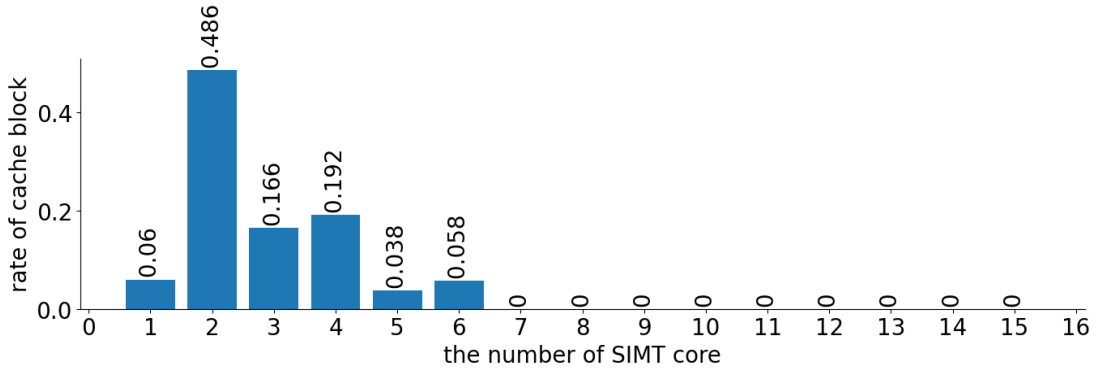


图 B10 LPS

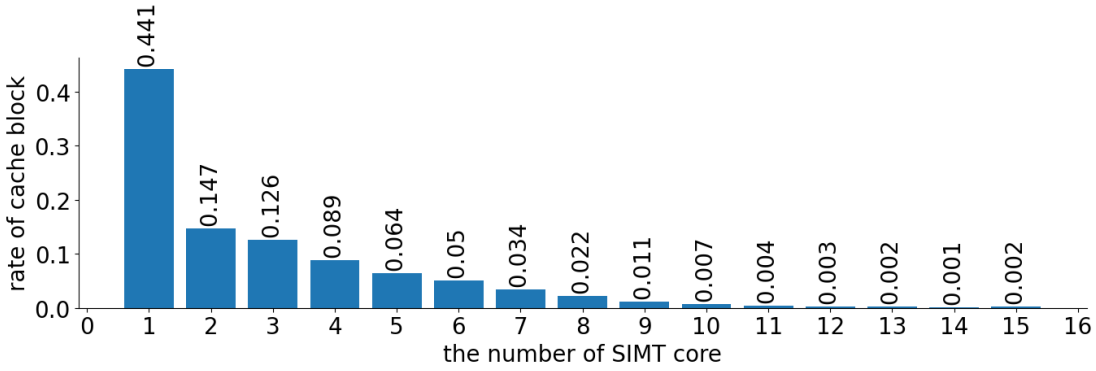


图 B11 LUD

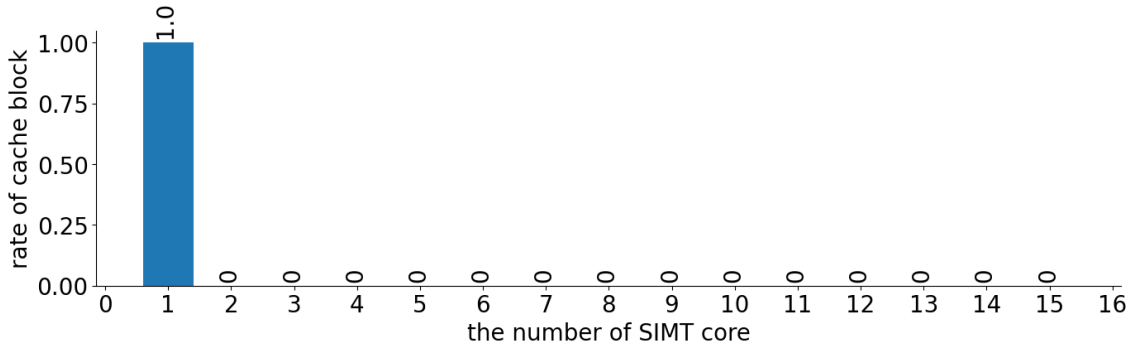


图 B12 MUM

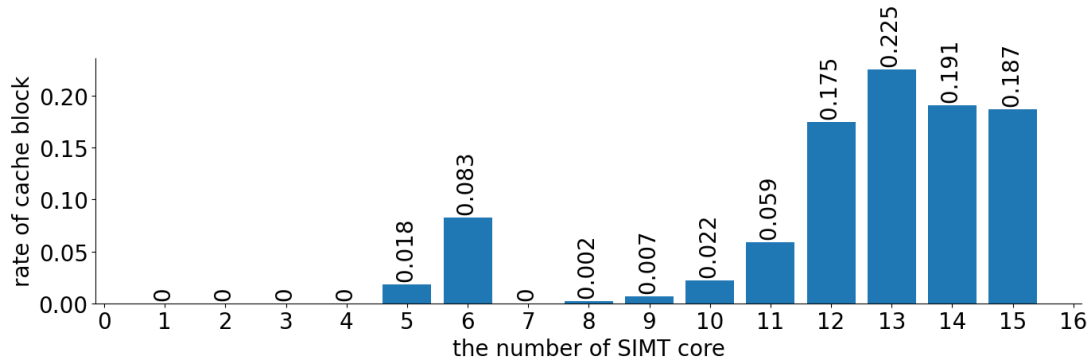


图 B13 NN

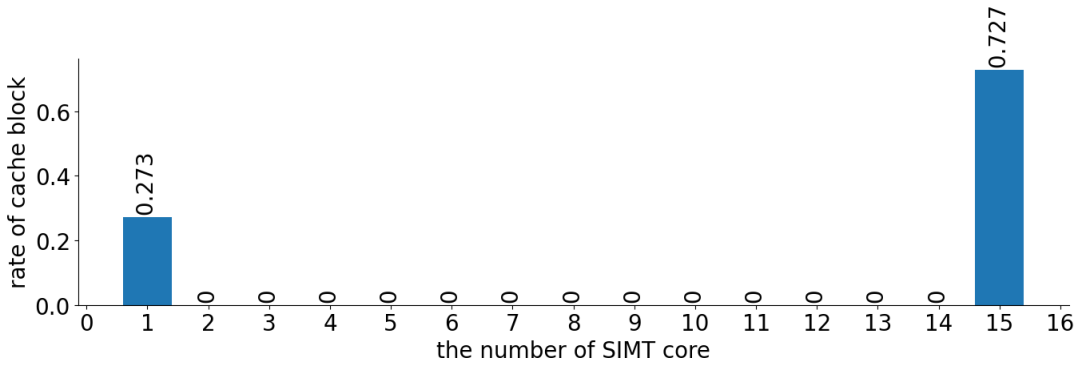


图 B14 NQU

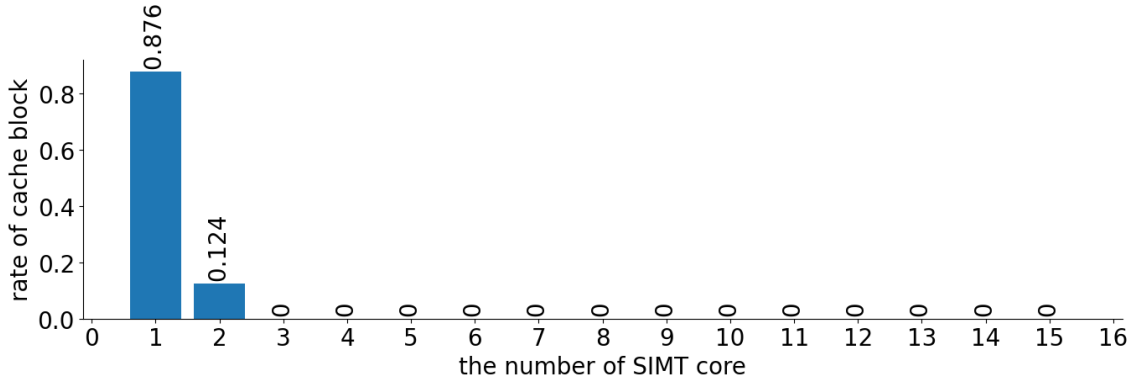


图 B15 STO

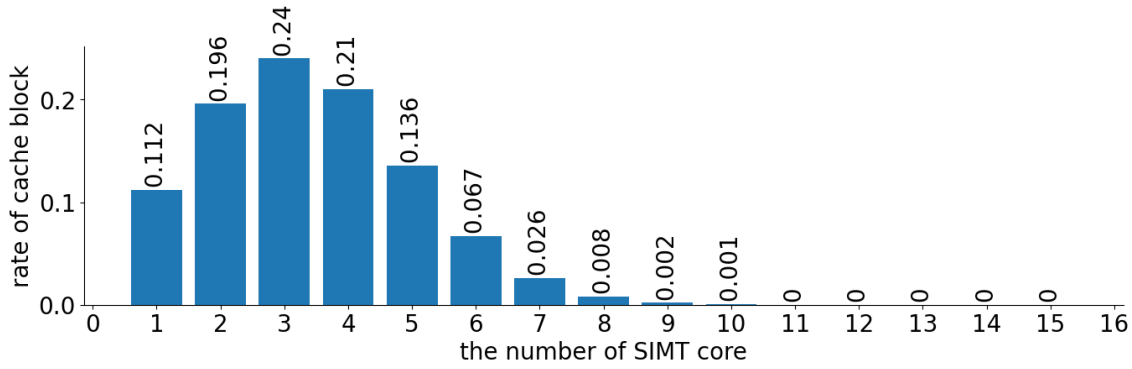


图 B16 PF

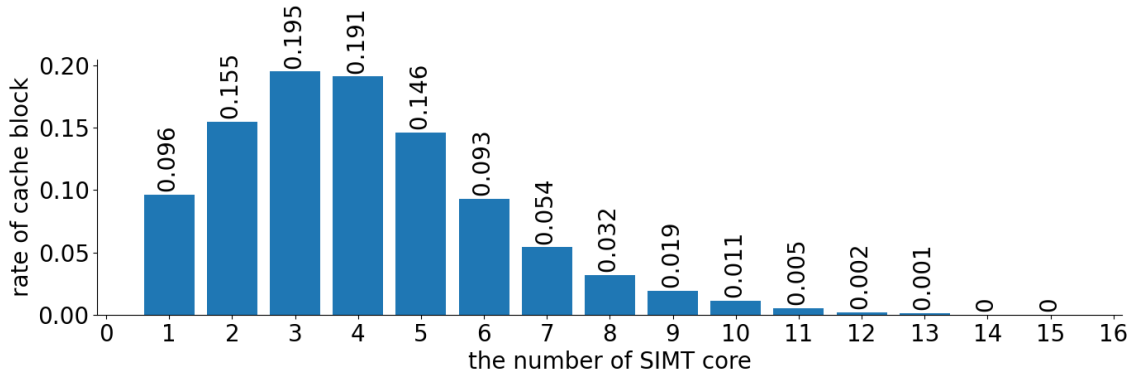


图 B17 SRAD