

IPCC

# 第六讲-并行优化实战

时间：2021年6月

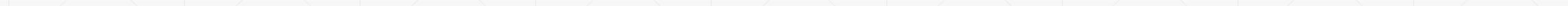
主讲人：张力越

# 前几讲回顾：

- 1、并行计算的概念和现代计算机体系结构  
    多核CPU、异构加速器、互联网络、IO、缓存；加速比、Flops等
- 2、性能分析工具  
    Vtune, Gprof, paramon paratune性能指标收集工具，readelf、objdump文件分析工具
- 3、并行开发编程技术和工具库  
    OpenMP、MPI；aocc、gcc、icc：编译优化开关、内建函数、内联汇编；  
    MKL、AOCL、BLAS
- 4、处理器优化技术  
    局部访存缓存优化、循环展开、向量化、数据对齐、函数内联、数据预取

# 本讲概要：

- 1、第一届IPCC初赛题优化报告
- 2、NUFFT应用优化案例分享
- 3、并行优化经验之谈



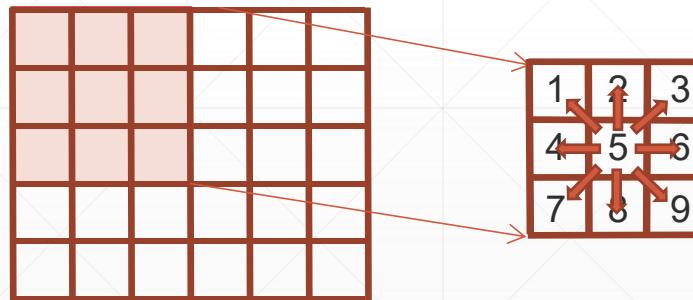
# 上届IPCC初赛题 stencil优化报告



## 1.1 赛题简介

- 9-point stencil 算法是矩阵计算中最为基础的一种算法，可以从输入的 png 图像中读取像素点信息然后对非边界网格进行加权模糊计算，从而得出结果像素点并生成输出 png 图像。该算法可以从图像计算延伸到各种矩阵数值计算领域中。

将图像读入后存储为一个二维矩阵，每个方块代表一个像素点



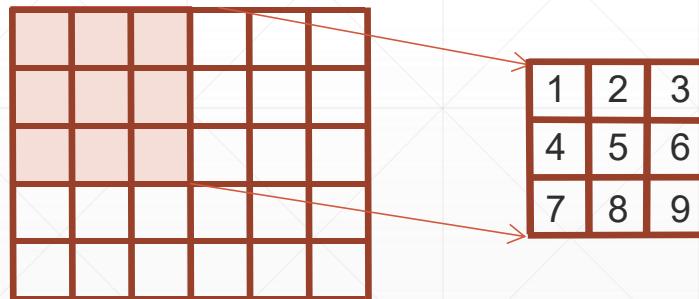
对于每一个像素点而言，stencil操作则是用该像素点的值分别减去与之相临的8个像素点的值，然后把它们的差累加起来得到该点计算后的值

$$\begin{aligned} \text{stencil\_value}[5] &= (a[5]-a[1])+(a[5]-a[2])+(a[5]-a[3])+ \\ &\quad (a[5]-a[4])+(a[5]-a[6])+ \\ &\quad (a[5]-a[7])+(a[5]-a[8])+(a[5]-a[9]) \\ &= -a[1] - a[2] - a[3] \\ &\quad - a[4] + 8*a[5] - a[6] \\ &\quad - a[7] - a[8] - a[9] \end{aligned}$$

## 1.1 赛题简介

- 9-point stencil 算法是矩阵计算中最为基础的一种算法，可以从输入的 png 图像中读取像素点信息然后对非边界网格进行加权模糊计算，从而得出结果像素点并生成输出 png 图像。该算法可以从图像计算延伸到各种矩阵数值计算领域中。

将图像读入后存储为一个二维矩阵，每个方块代表一个像素点



$$\begin{aligned} \text{stencil\_value}[5] &= (a[5]-a[1])+(a[5]-a[2])+(a[5]-a[3])+ \\ &\quad (a[5]-a[4])+\dots+(a[5]-a[6])+ \\ &\quad (a[5]-a[7])+(a[5]-a[8])+(a[5]-a[9]) \\ &= -a[1] - a[2] - a[3] \\ &\quad - a[4] + 8*a[5] - a[6] \\ &\quad - a[7] - a[8] - a[9] \end{aligned}$$

如此计算每个非边缘像素点的值直到全部完成

## 1.2 样例程序代码分析

1. 源代码需包括以下文件：

✓ 根目录

- 1) IPCC.png: 程序输入图像文件
- 2) check.png: 结果验证图像文件
- 3) check.txt: 结果验证数据文件

✓ src

- 1) image.cc: 图像处理接口函数实现
- 2) image.h: 图像处理接口函数声明头文件
- 3) main.cc: 主程序入口
- 4) stencil.cc: 模糊计算函数实现
- 5) stencil.h: 模糊计算函数声明

✓ include

- 1) png.h、pngconf.h、pnglibconf.h: png 图像文件处理头文件

✓ lib

- 1) libpng16.a、libz.a: png 图像文件处理静态库文件
-

## 1.2 样例程序代码分析

- 核心计算部分(stencil.cc):
- 非常简单的一个二维遍历
- 计算部分是8次减法，一次乘加
- 注意要计算后的结果有取值范围

```
template<typename P>
void ApplyStencil(ImageClass<P> & img_in, ImageClass<P> & img_out) {
    const int width = img_in.width;
    const int height = img_in.height;

    P * in = img_in.pixel;
    P * out = img_out.pixel;

    for (int j = 1; j < width-1; j++) {
        for (int i = 1; i < height-1; i++) {

            int im1jm1 = (i-1)*width + j-1;
            int im1j   = (i-1)*width + j;
            int im1jp1 = (i-1)*width + j+1;
            int ijm1   = (i )*width + j-1;
            int ij    = (i )*width + j;
            int ijp1   = (i )*width + j+1;
            int ip1jm1 = (i+1)*width + j-1;
            int ip1j   = (i+1)*width + j;
            int ip1jp1 = (i+1)*width + j+1;
            P val =
                -in[im1jm1] - in[im1j] - in[im1jp1]
                -in[ijm1] + 8*in[ij] - in[ijp1]
                -in[ip1jm1] - in[ip1j] - in[ip1jp1];
            val = (val < 0 ? 0 : val);
            val = (val > 255 ? 255 : val);

            out[i*width + j] = val;
        }
    }
}
```

## 1.2 样例程序代码分析

- 计时部分 (main.cc) :

运行10次相同的stencil，每次都计时  
最后以10次总时间为准

此处跑了原始样例程序，单次时间1100ms左右，  
总时间10989.4 ms  
以此为基准衡量后续加速比

```
34     nTrials = 10;
35     double t, dt;
36
37     for (int iTrial = 1; iTrial <= nTrials; iTrial++) {
38         const double t0 = wtime();
39         ApplyStencil(img_in, img_out);
40         const double t1 = wtime();
41
42         const double ts    = t1-t0; // time in seconds
43         const double tms   = ts*1.0e3; // time in milliseconds
44
45         t += tms;
46         dt += tms*tms;
47
48         printf("%5d %15.3f \n",
49                iTrial, tms);
50         fflush(stdout);
51     }
52
53     printf("-----\n");
54     printf("%s %8.1f %s\n",
55            "Total :", t, "ms");
56     printf("-----\n");
```

## 1.2 样例程序代码分析

- 根据赛题规则：
- 不可在 nTrials 循环做并行划分。
- 计时不包括图片读写的前后处理时间，耗时以程序最终输出的 Total 时间为准

尽量减少单次stencil耗时

- 可以改变数据结构和数据类型
- 可以使用其他读取图像算法库代替include 中头文件、lib 中静态库读取、输出结果图像文件

可以适当修改输入输出部分数据结构和类型

---

## 1.3 优化思路

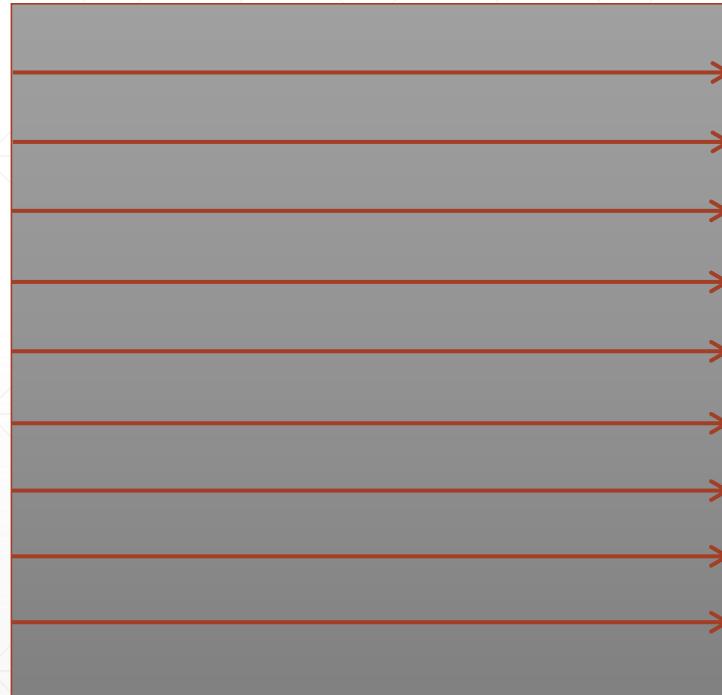
- 1、编译优化
  - 2、OpenMP并行执行stencil
  - 3、循环展开&向量化
-

## 1.4 优化实现——编译优化

- 使用intel icpc编译器，并加入优化参数如下：  
`-std=c++11 -no-gcc -O3 -ipo -qopenmp -fp-model fast=2 -xHost`
- 开启O3优化，ipo过程间优化，fp-model fast2加快浮点计算速度，
- xHost使得编译器根据平台cpu生成使用avx等高级指令的程序。
  
- 使用intel编译器，比gcc aocc开启同等级优化快了一倍左右：
- 2628.9 ms vs 4637.8 ms
- 相比基准10989.4 ms已经实现了4倍左右的加速

# 1.4 优化实现——openmp并行

- 使用OpenMP for制导，对数据按照行划分（单节点64核使用64线程）



```
#pragma omp parallel for simd
for (int i = 1; i < height-1; i++) {
    for (int j = 1; j < width-1; j++) {

        int im1jm1 =(i-1)*width + j-1;
        int im1j   =(i-1)*width + j;
        int im1jp1 =(i-1)*width + j+1;
        int ijm1   =(i  )*width + j-1;
        int ij    =(i  )*width + j;
        int ijp1   =(i  )*width + j+1;
        int ip1jm1 =(i+1)*width + j-1;
        int ip1j   =(i+1)*width + j;
        int ip1jp1 =(i+1)*width + j+1;
        p val =
            -in[im1jm1] - in[im1j] - in[im1jp1]
            -in[ijm1]   + 8*in[ij] - in[ijp1]
            -in[ip1jm1] - in[ip1j] - in[ip1jp1];

        val = (val < 0 ? 0 : val);
        val = (val > 255 ? 255 : val);

        out[i*width + j] = val;
    }
}
```

注意这里内外层交换了，  
先行后列有利于访存局部性

## 1.4 优化实现——循环展开&向量化

- 内层循环以8个为一组，凑齐avx256宽度（8个float）
- 使用intrinsics接口手动实现向量化
- 使用avx256向量的时候要注意边界，尾部少于8个数据的时候则需要单个计算



# 1.4 优化实现——循环展开&向量化

```
#pragma omp parallel for
for (int i = 1; i < height - 1; i++)
{
    int j = 1;
    for (; j < width - 1 - 8; j += 8) {
        __m256 v00,v01,v02,v10,v11,v12,v20,v21,v22;
        __m256 val, eight, zero, b255;
        eight = _mm256_set1_ps((float)8);
        zero = _mm256_set1_ps((float)0);
        b255 = _mm256_set1_ps((float)255);
        v00 = _mm256_loadu_ps(in + (j - width - 1));
        v01 = _mm256_loadu_ps(in + (j - width));
        v02 = _mm256_loadu_ps(in + (j - width + 1));
        v10 = _mm256_loadu_ps(in + (j - 1));
        v11 = _mm256_loadu_ps(in + j);
        v12 = _mm256_loadu_ps(in + (j + 1));
        v20 = _mm256_loadu_ps(in + (j + width - 1));
        v21 = _mm256_loadu_ps(in + (j + width));
        v22 = _mm256_loadu_ps(in + (j + width + 1));

        val = _mm256_fmsub_ps(eight, v11, _mm256_add_ps(_mm256_add_ps(v10,v12),
        _mm256_add_ps(_mm256_add_ps(v00,_mm256_add_ps(v01,v02)),_mm256_add_ps(v20,_mm256_add_ps(v21,v22)))));

        val = _mm256_max_ps(val, zero);
        val = _mm256_min_ps(val, b255);
        _mm256_storeu_ps(out + j, val);
        for ( ; j < width - 1; j++) {
            int im1jm1 =(i-1)*width + j-1;
            int im1j   =(i-1)*width + j;
            int im1jp1 =(i-1)*width + j+1;
            int ijm1   =(i  )*width + j-1;
            int ij     =(i  )*width + j;
```

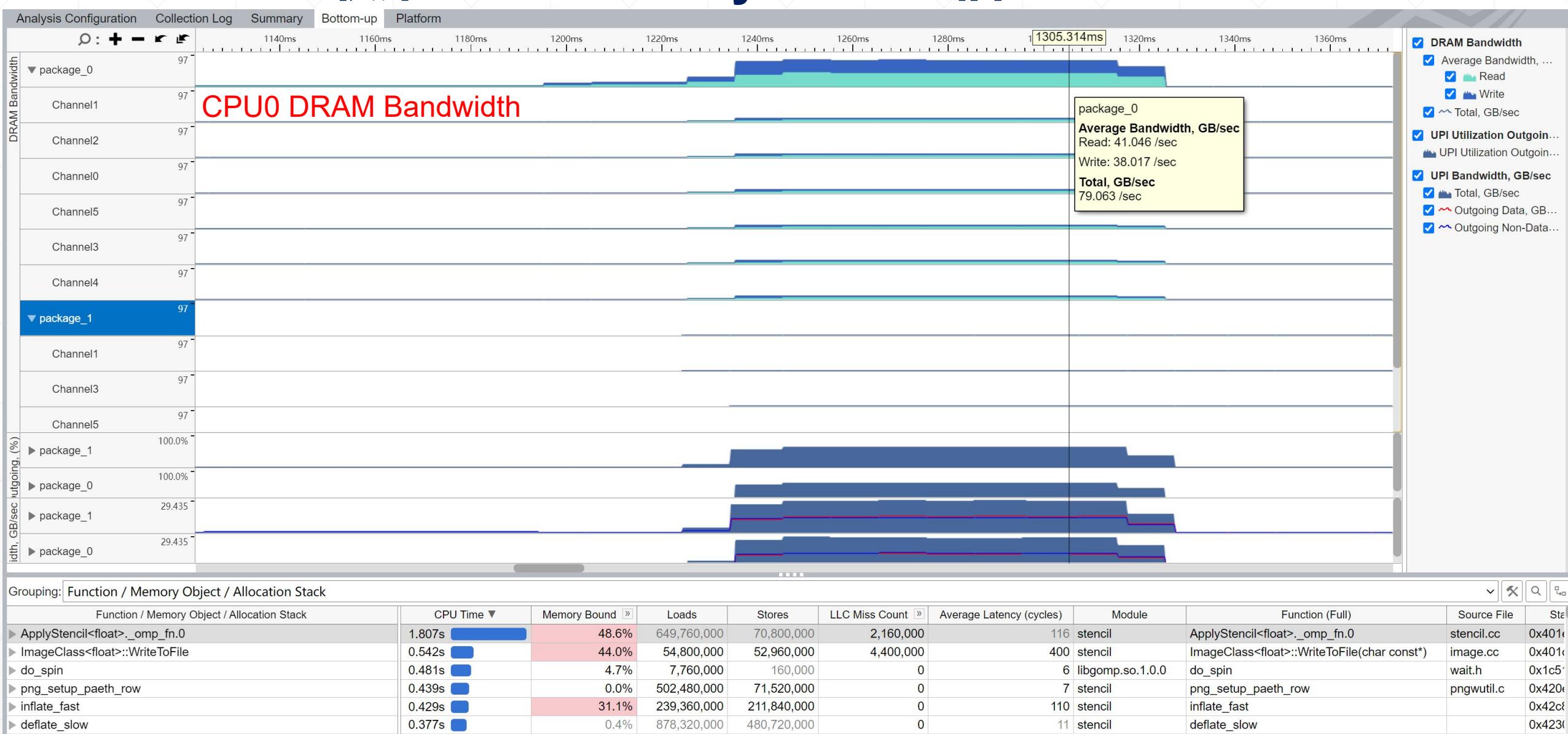
Edge detection with a 3x3 stencil	
Image size: 10410 x 5905	
Trial	Time, ms
1	10.903
2	5.579
3	5.530
4	5.551
5	5.511
6	5.543
7	5.534
8	5.503
9	5.531
10	5.486
<hr/>	
Total :	60.7 ms
<hr/>	

## 1.5 进一步优化

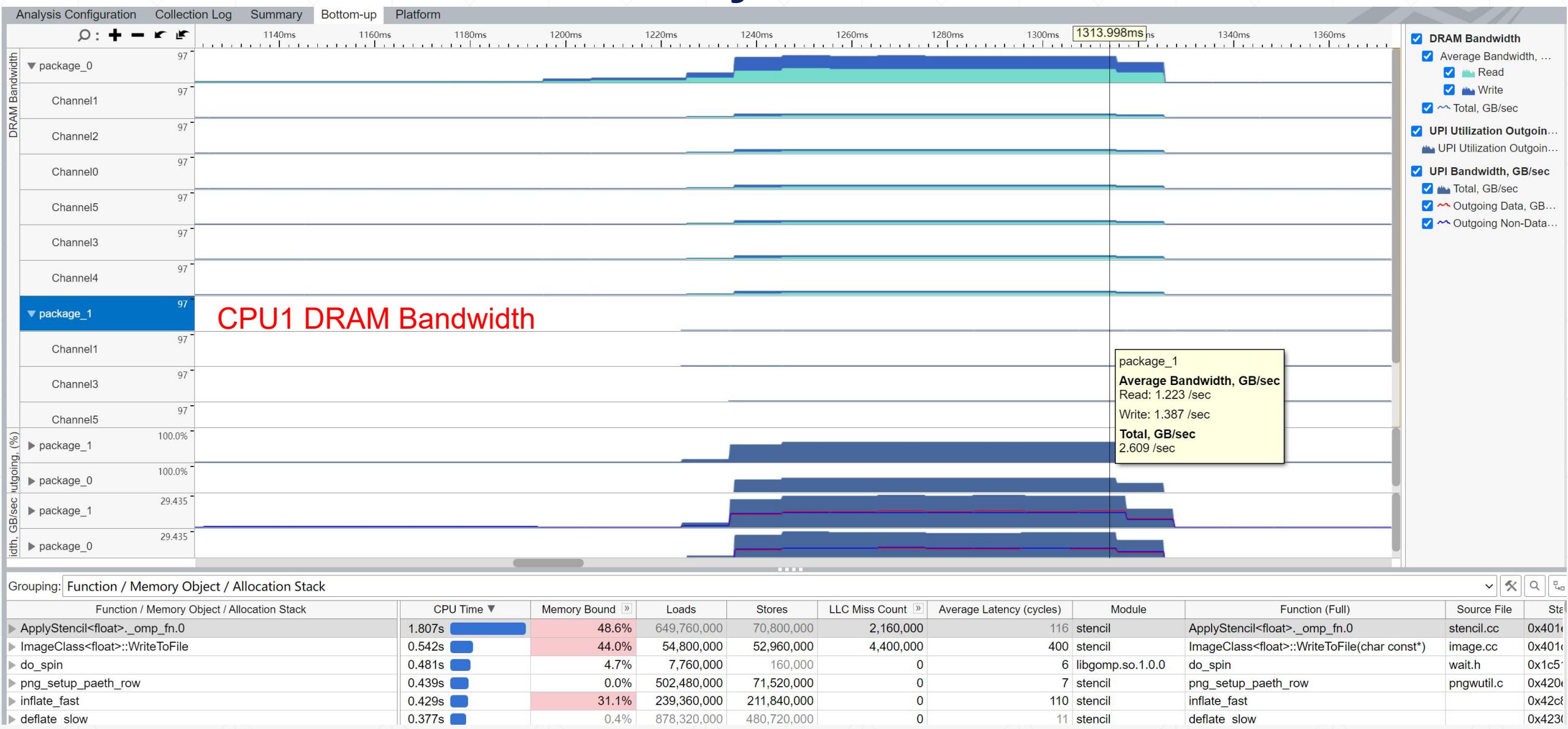
- 使用Vtune采集hpc uarch memory-access
- 由于采集上述指标需要使用硬件采样，比赛提供的集群不支持（没有权限）
- 所以这里使用我本地的Intel 双路 金牌5120 CPU来采集
- 跟比赛使用的双路epyc 7452会有差别，但反应的问题是一样的



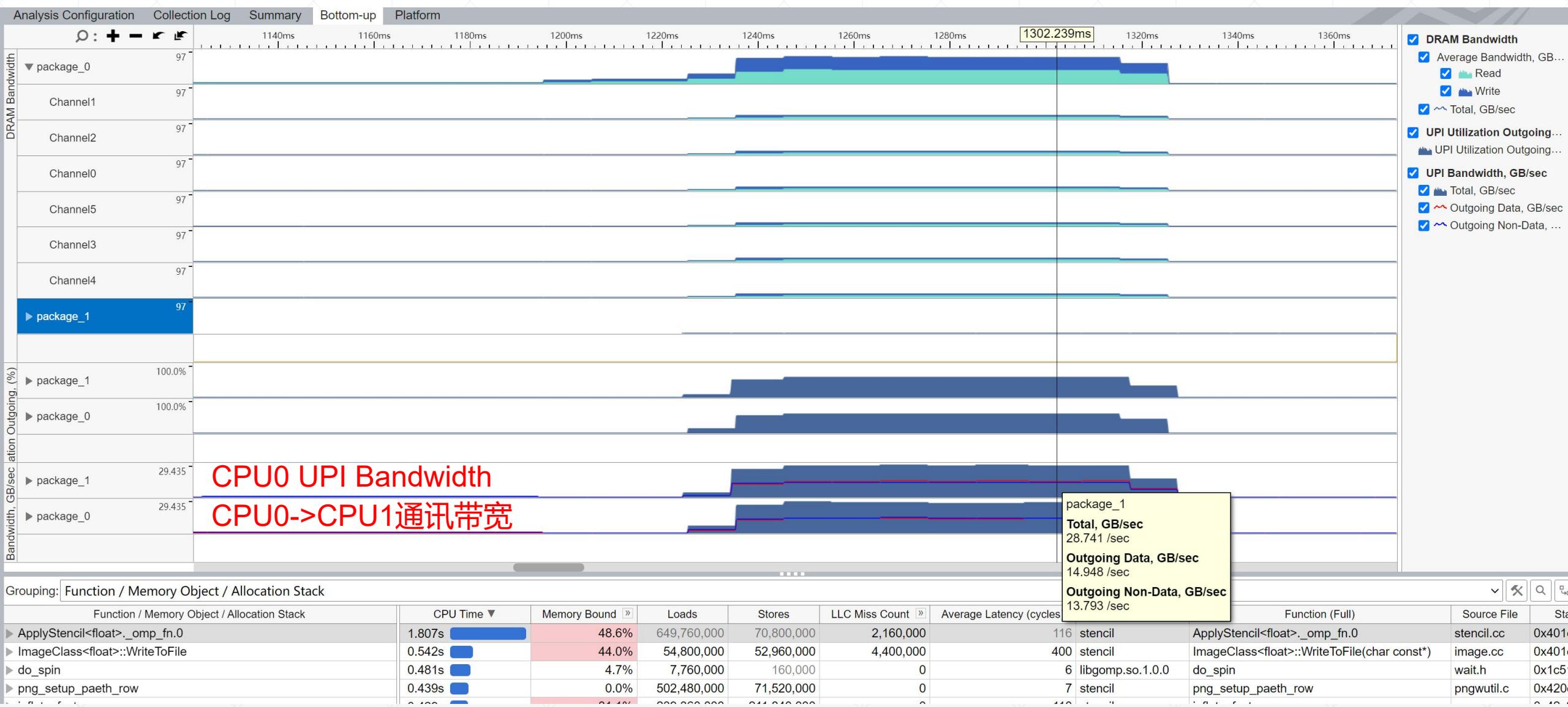
# 1.5 Vtune收集HPC memory-access信息



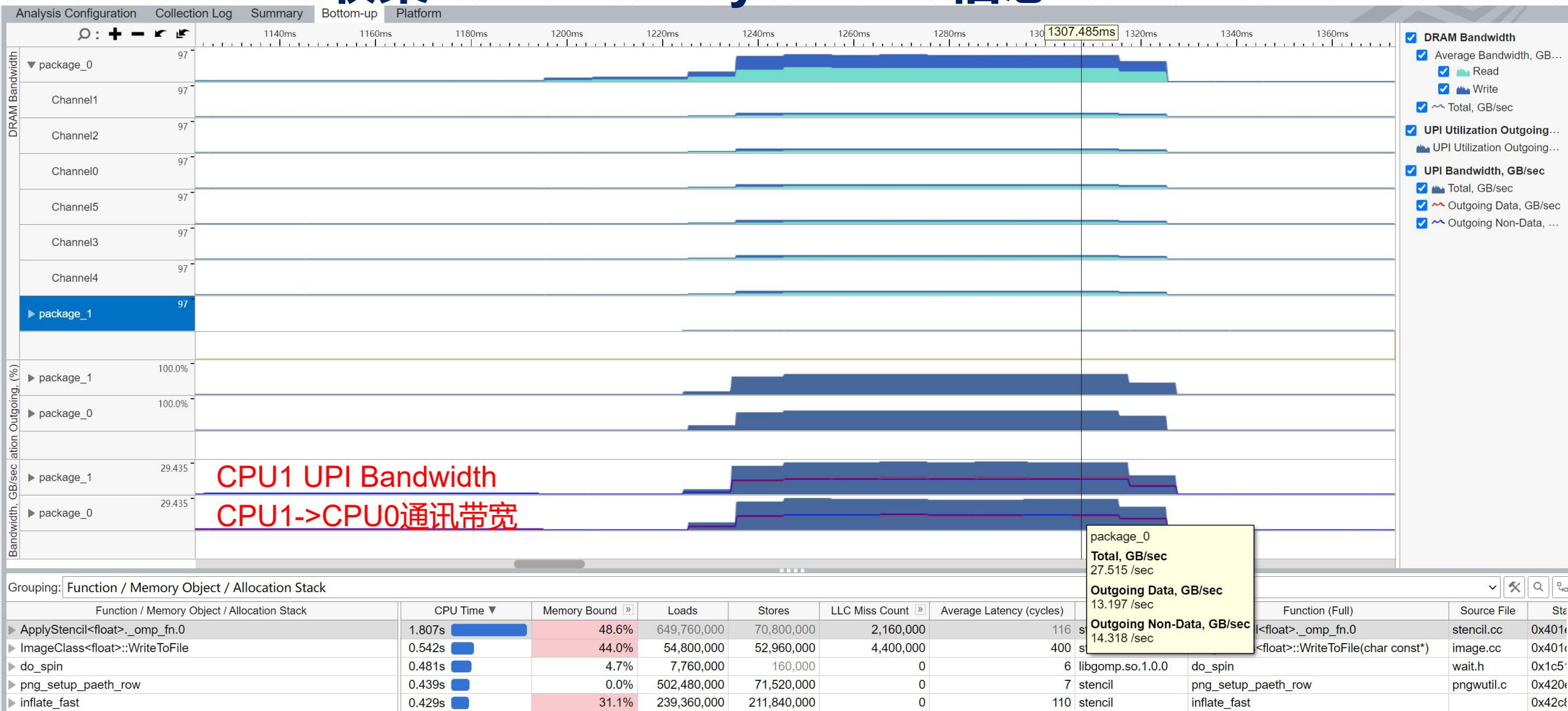
# 1.5 Vtune 收集HPC memory-access信息



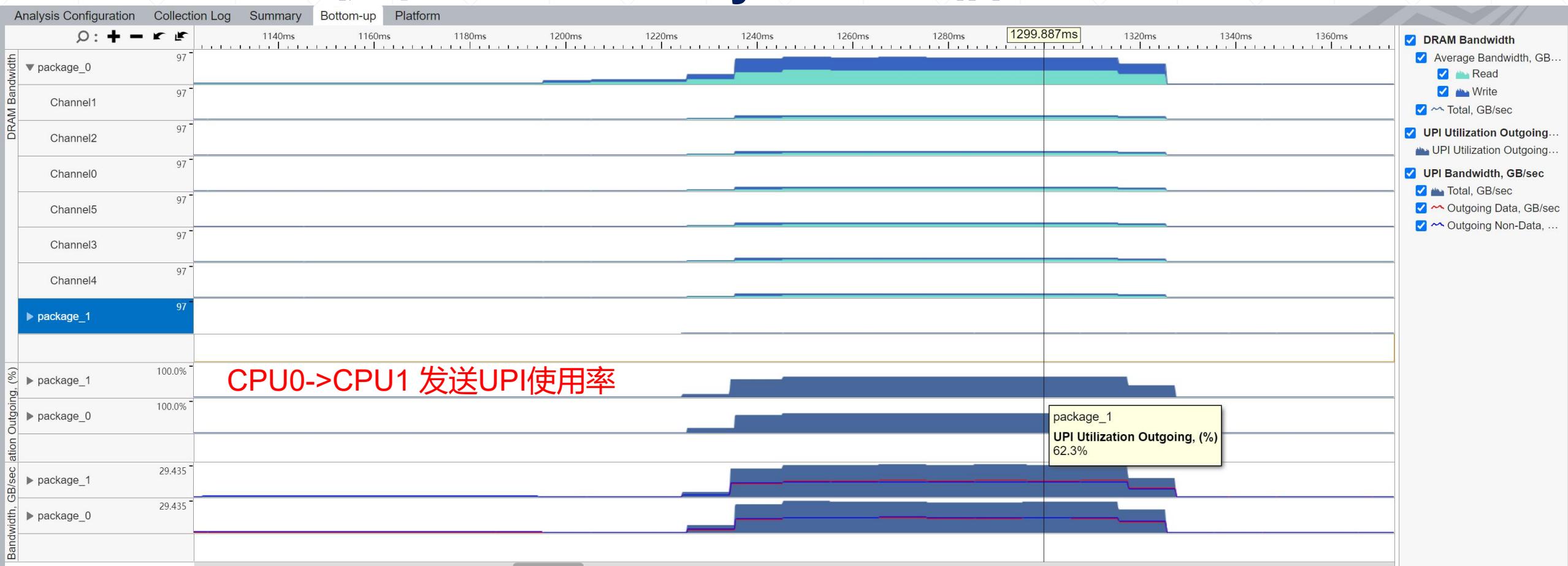
# 1.5 Vtune收集HPC memory-access信息



# 1.5 Vtune收集HPC memory-access信息

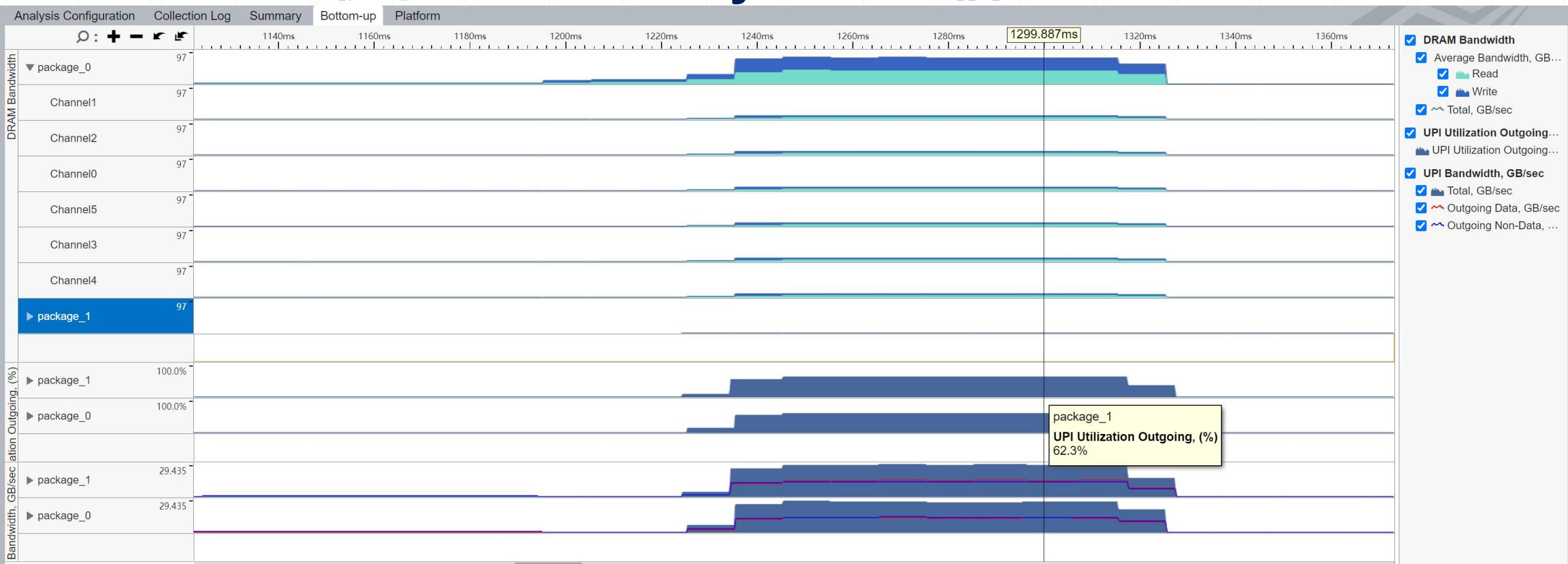


# 1.5 Vtune收集HPC memory-access信息



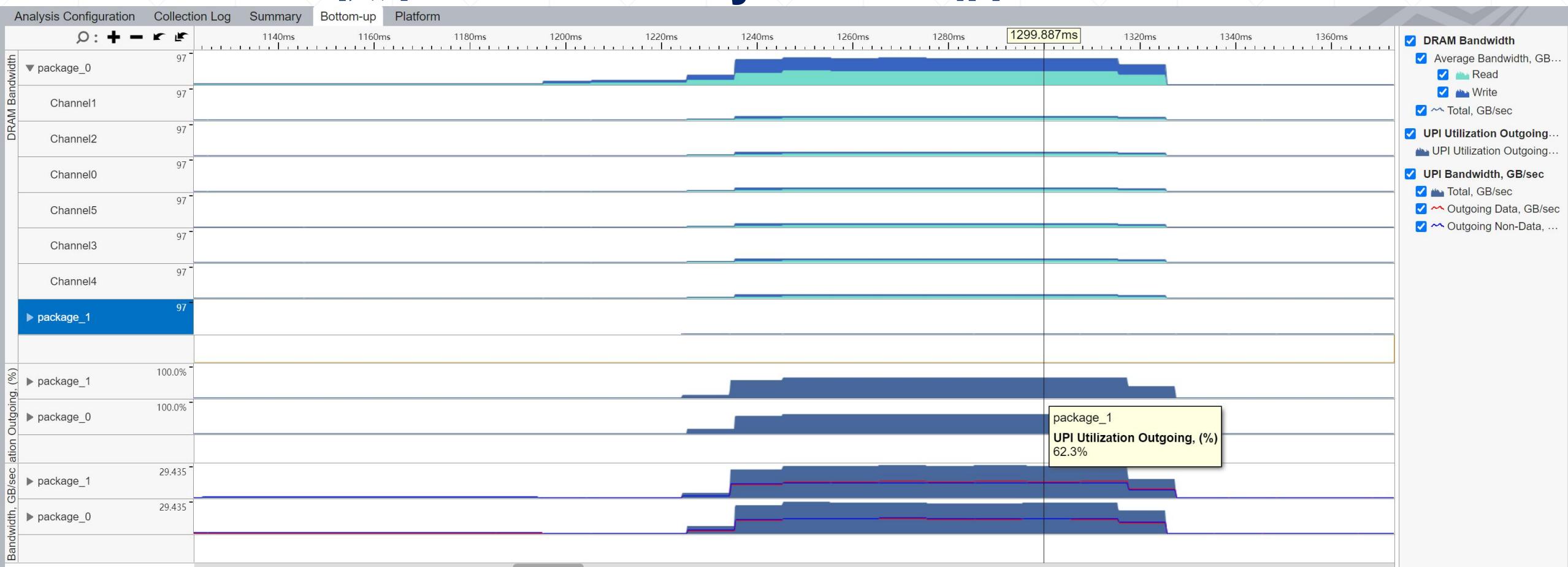
Function / Memory Object / Allocation Stack	CPU Time ▼	Memory Bound »	Loads	Stores	LLC Miss Count »	Average Latency (cycles)	Module	Function (Full)	Source File	Sta
▶ ApplyStencil<float>._omp_fn.0	1.807s	48.6%	649,760,000	70,800,000	2,160,000	116	stencil	ApplyStencil<float>._omp_fn.0	stencil.cc	0x401e
▶ ImageClass<float>::WriteToFile	0.542s	44.0%	54,800,000	52,960,000	4,400,000	400	stencil	ImageClass<float>::WriteToFile(char const*)	image.cc	0x401d
▶ do_spin	0.481s	4.7%	7,760,000	160,000	0	6	libgomp.so.1.0.0	do_spin	wait.h	0x1c5f
▶ png_setup_paeth_row	0.439s	0.0%	502,480,000	71,520,000	0	7	stencil	png_setup_paeth_row	pngutil.c	0x420e
▶ inflate_fast	0.429s	31.1%	239,360,000	211,840,000	0	110	stencil	inflate_fast	0x42c8	
▶ deflate_slow	0.377s	0.4%	878,320,000	480,720,000	0	11	stencil	deflate_slow	0x4230	
▶ adler32	0.176s	0.4%	390,320,000	62,560,000	0	11	stencil	adler32	0x4219	

# 1.5 Vtune收集HPC memory-access信息



Function / Memory Object / Allocation Stack	CPU Time ▼	Memory Bound »	Loads	Stores	LLC Miss Count »	Average Latency (cycles)	Module	Function (Full)	Source File	Sta
▶ ApplyStencil<float>._omp_fn.0	1.807s	48.6%	649,760,000	70,800,000	2,160,000	116	stencil	ApplyStencil<float>._omp_fn.0	stencil.cc	0x401e
▶ ImageClass<float>::WriteToFile	0.542s	44.0%	54,800,000	52,960,000	4,400,000	400	stencil	ImageClass<float>::WriteToFile(char const*)	image.cc	0x401d
▶ do_spin	0.481s	4.7%	7,760,000	160,000	0	6	libgomp.so.1.0.0	do_spin	wait.h	0x1c5f
▶ png_setup_paeth_row	0.439s	0.0%	502,480,000	71,520,000	0	7	stencil	png_setup_paeth_row	pngutil.c	0x420e
▶ inflate_fast	0.429s	31.1%	239,360,000	211,840,000	0	110	stencil	inflate_fast	0x42c8	
▶ deflate_slow	0.377s	0.4%	878,320,000	480,720,000	0	11	stencil	deflate_slow	0x4230	
▶ adler32	0.176s	0.4%	390,320,000	62,560,000	0	11	stencil	adler32	0x4219	

# 1.5 Vtune 收集HPC memory-access信息



Function / Memory Object / Allocation Stack	CPU Time	Memory Bound	Loads	Stores	LLC Miss Count	Average Latency (cycles)	Module	Function (Full)	Source File	Start Address
▶ ApplyStencil<float>._omp_fn.0	1.807s	48.6%	649,760,000	70,800,000	2,160,000	116	stencil	ApplyStencil<float>._omp_fn.0	stencil.cc	0x401e
▶ ImageClass<float>::WriteToFile	0.542s	44.0%	54,800,000	52,960,000	4,400,000	400	stencil	ImageClass<float>::WriteToFile(char const*)	image.cc	0x401d
▶ do_spin	0.481s	4.7%	7,760,000	160,000	0	6	libgomp.so.1.0.0	do_spin	wait.h	0x1c5f
▶ png_setup_paeth_row	0.439s	0.0%	502,480,000	71,520,000	0	7	stencil	png_setup_paeth_row	pngutil.c	0x420e
▶ inflate_fast	0.429s	31.1%	239,360,000	211,840,000	0	110	stencil	inflate_fast	0x42c8	0x4230
▶ deflate_slow	0.377s	0.4%	878,320,000	480,720,000	0	11	stencil	deflate_slow		0x4219
▶ adler32	0.176s	0.4%	390,320,000	62,560,000	0	11	stencil	adler32		0x4219

# 1.5 Vtune收集HPC memory-access信息



- 1、该应用是访存瓶颈
- 2、CPU1 (对于NUMA系统来说也就是NUMA1) 的内存没有被使用
- 3、缓存命中率过低、访存延迟过高，说明跨socket通讯严重影响整体计算速度

# 1.5 进一步优化——使用stream写指令代替store写

- 1、针对访存瓶颈：使用stream写指令代替store写，尽可能打满内存带宽
- `_mm256_storeu_ps(out + j, val);`
- `_mm256_stream_ps(out + j, val);` (注意要对齐)

```
Edge detection with a 3x3 stencil
Image size: 10410 x 5905

Trial      Time, ms
 1          10.903
 2          5.579
 3          5.530
 4          5.551
 5          5.511
 6          5.543
 7          5.534
 8          5.503
 9          5.531
10          5.486

Total :     60.7 ms
```

```
running...
Edge detection with a 3x3 stencil
Image size: 10410 x 5905

Trial      Time, ms
 1          5.066
 2          2.439
 3          2.447
 4          2.429
 5          2.448
 6          2.421
 7          2.416
 8          2.442
 9          2.413
10          2.436

Total :     27.0 ms

Output written into data.txt and output.png
```

# 1.5 进一步优化——绑定本地NUMA内存

- 使用mmap替换malloc申请内存，同时初次访问时使用omp，保证真实访问内存地址跟线程所对应的CPU核心在同一个NUMA节点

```
template<typename P>
ImageClass<P>::ImageClass(int const _width, int const _height)
    : width(_width), height(_height) {

    // Initialize a blank image
    //pixel = (P*)malloc(sizeof(P)*width*height);
    pixel = (P*)mmap(NULL, sizeof(P)*width*height, PROT_READ | PROT_WRITE, MAP_ANONYMOUS|MAP_PRIVATE , -1, 0);

#pragma omp parallel for
    for (int i = 0; i < height * width; i++)
        pixel[i] = (P)0;
}
```

申请内存的时候操作系统并不会把内存空间给程序，而是在访问该内存时候触发页缺失(page fault)中断，才会将内存页分配给程序使用，默认分配策略下，会优化分配跟触发中断的线程相同numa的内存。

使用mmap是同时设置MAP\_PRIVATE标致是为了保证申请的内存区间是新的，如果用malloc可能是之前free的，导致分配时没有根据线程亲和性分配同一numa的内存。

# 1.5 进一步优化——绑定本地NUMA内存

```
Edge detection with a 3x3 stencil

Image size: 10410 x 5905

Trial      Time, ms
1          5.066
2          2.439
3          2.447
4          2.429
5          2.448
6          2.421
7          2.416
8          2.442
9          2.413
10         2.436
-----
Total :    27.0 ms
-----
```

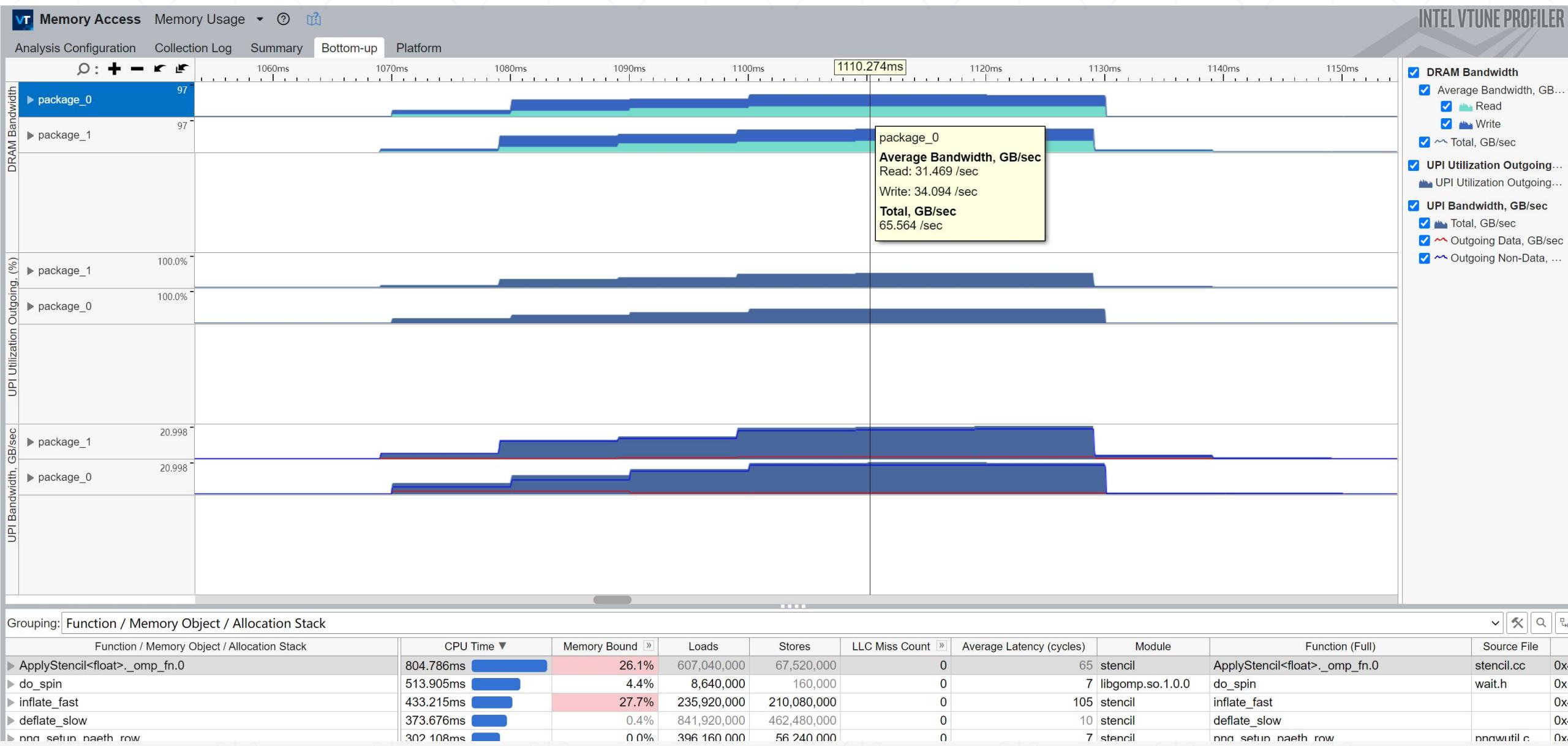
```
Edge detection with a 3x3 stencil

Image size: 10410 x 5905

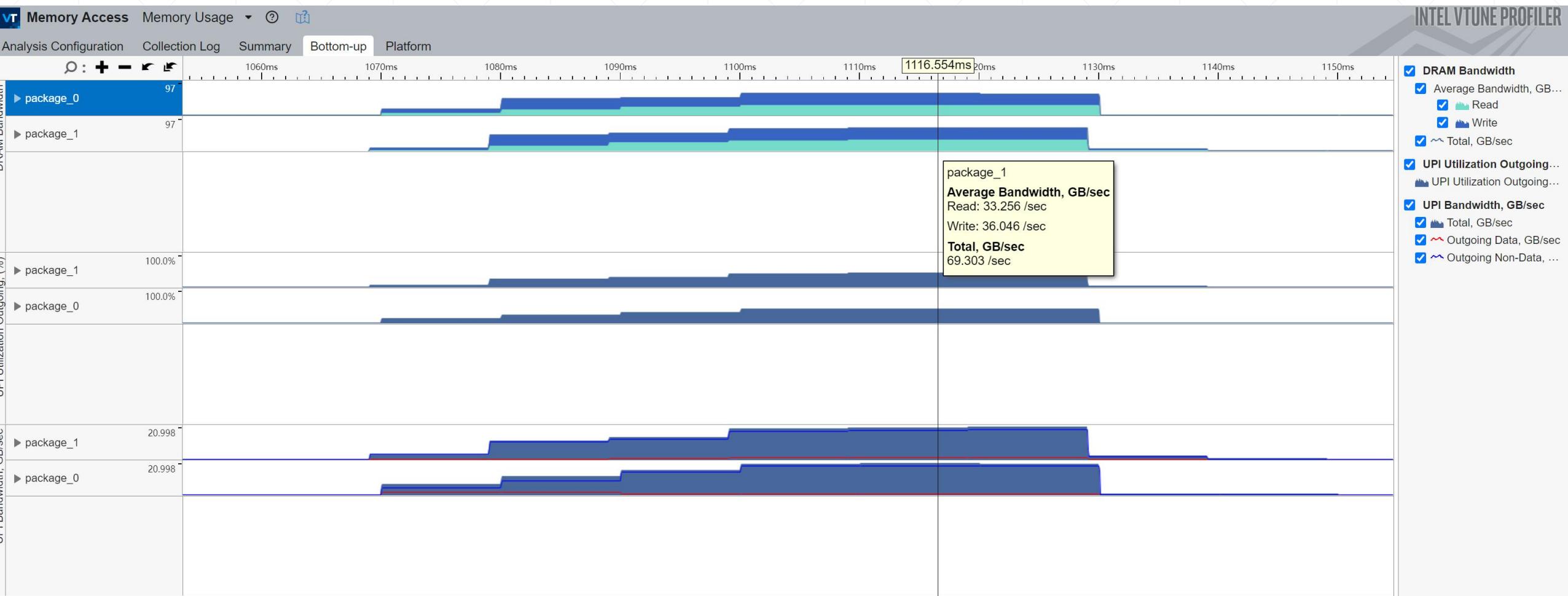
Trial      Time, ms
1          2.385
2          1.526
3          1.491
4          1.480
5          1.495
6          1.485
7          1.493
8          1.484
9          1.486
10         1.481
-----
Total :    15.8 ms
-----
```

可见比单NUMA快了接近一倍（双路访存速度接近翻倍）  
相比baseline的10989.4 ms，实现了**695.53倍**的加速

# 1.6 总结



# 1.6 总结



Grouping: Function / Memory Object / Allocation Stack											
Function / Memory Object / Allocation Stack		CPU Time ▼	Memory Bound ▶»	Loads	Stores	LLC Miss Count ▶»	Average Latency (cycles)	Module	Function (Full)	Source File	
ApplyStencil<float>_omp_fn.0		804.786ms	26.1%	607,040,000	67,520,000	0	65	stencil	ApplyStencil<float>_omp_fn.0	stencil.cc	0x...
do_spin		513.905ms	4.1%	6,610,000	100,000	0	7	libgomp.so.1.0.0	do_spin	wait.h	0x...
inflate_fast		433.215ms	27.7%	235,920,000	210,080,000	0	105	stencil	inflate_fast		0x...
deflate_slow		373.676ms	0.4%	841,920,000	462,480,000	0	10	stencil	deflate_slow		0x...
png_setup_paeth_row		302.108ms	0.0%	396,160,000	56,240,000	0	7	stencil	png_setup_paeth_row	pngwutil.c	0x...

# 1.6 总结

NUMA优化前：

Grouping: Function / Memory Object / Allocation Stack

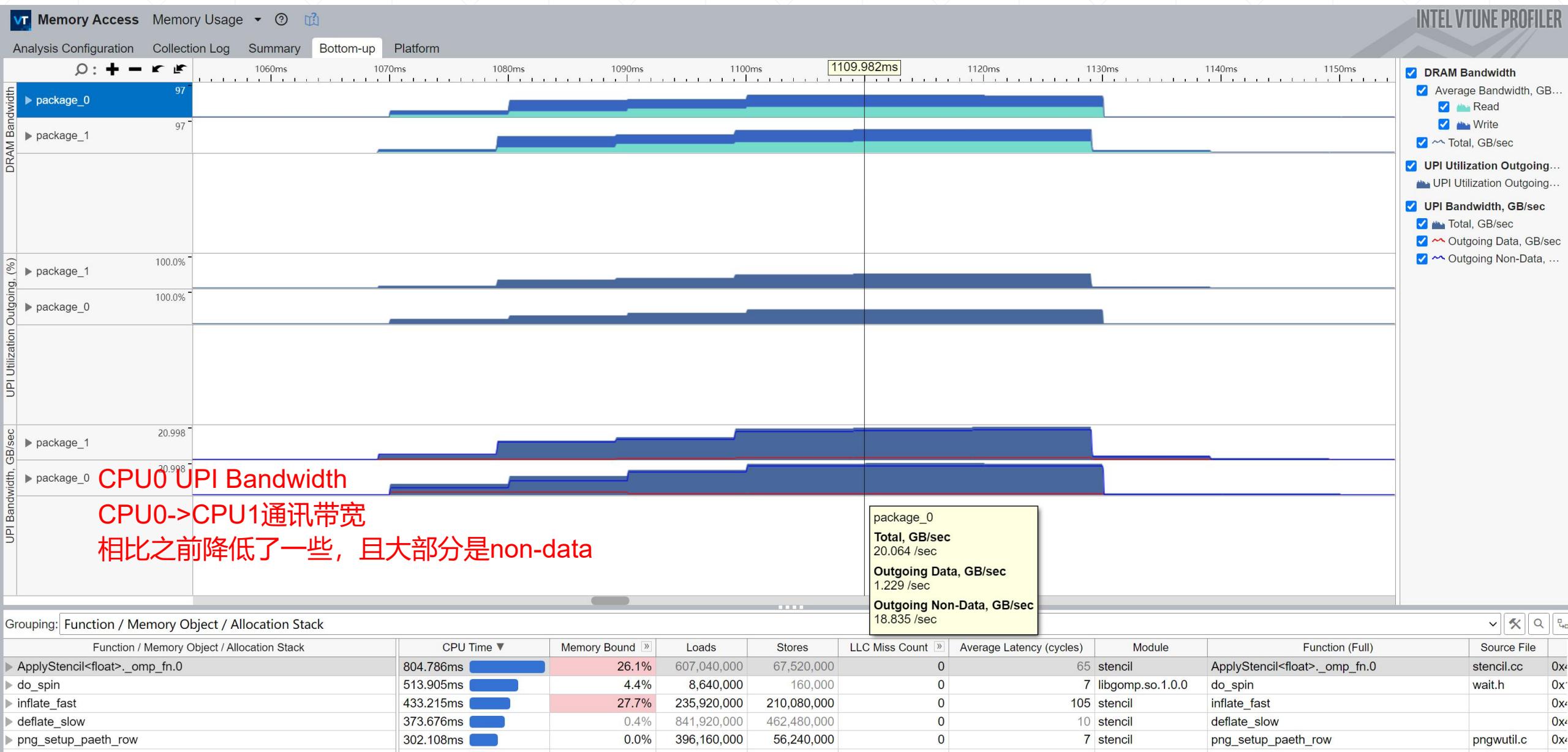
Function / Memory Object / Allocation Stack	CPU Time ▼	Memory Bound »	Loads	Stores	LLC Miss Count »	Average Latency (cycles)	Module
▶ ApplyStencil<float>._omp_fn.0	1.807s	48.6%	649,760,000	70,800,000	2,160,000	116	stencil A
▶ ImageClass<float>::WriteToFile	0.542s	44.0%	54,800,000	52,960,000	4,400,000	400	stencil In
▶ do_spin	0.481s	4.7%	7,760,000	160,000	0	6	libgomp.so.1.0.0 dc
▶ dna setup paeth row	0.439s	0.0%	502,480,000	71,520,000	0	7	stencil pi

NUMA优化后：

Grouping: Function / Memory Object / Allocation Stack

Function / Memory Object / Allocation Stack	CPU Time ▼	Memory Bound »	Loads	Stores	LLC Miss Count »	Average Latency (cycles)	Module
▶ ApplyStencil<float>._omp_fn.0	804.786ms	26.1%	607,040,000	67,520,000	0	65	stencil
▶ do_spin	513.905ms	4.4%	6,810,000	100,000	0	7	libgomp.so.1.0.0
▶ inflate_fast	433.215ms	27.7%	235,920,000	210,080,000	0	105	stencil

# 1.6 总结



## 1.6 总结&答疑

- 优化方法：
  - 1、编译优化
  - 2、OpenMP按行分块并行
  - 3、向量化
  - 4、stream写
  - 5、NUMA内存优化
- 优化过程：
  - 1、运行+收集数据
  - 2、分析瓶颈制定优化策略
  - 3、编程实现优化
  - 4、重复1-2-3过程

## 2、NUFFT应用优化案例分享

- 非标准快速傅里叶变换（Non-Uniform Fast Fourier Transform, NUFFT）是一种处理非标准采样的快速算法，在信号处理方面广泛应用。
- NUFFT常用于3D-MRI成像，NUFFT算法通常使用两个NUFFT算子：forward NUFFT (FWD) 和adjacent NUFFT (ADJ) 。
- 计算是以迭代的形式交替调用两个算子：即每轮迭代由一次Forward NUFFT和ADJ NUFFT以及迭代后数据处理组成。

## 2.1 程序及算法分析

计算一个信号f的forward NUFFT包括以下三步：

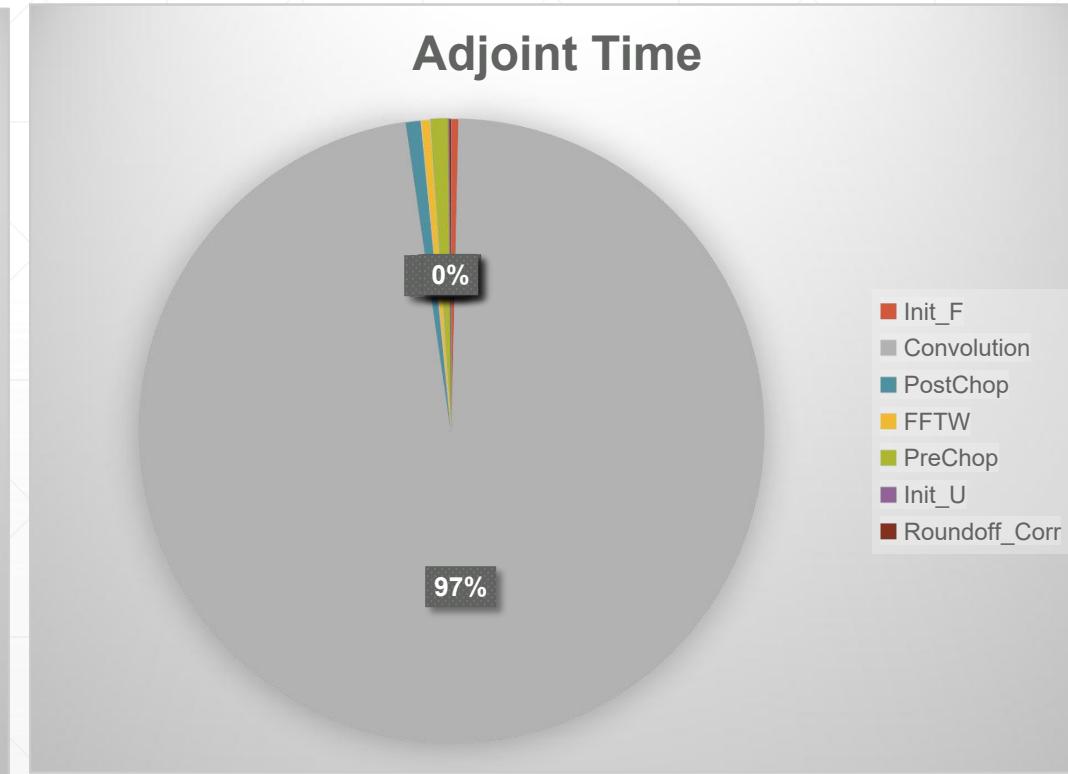
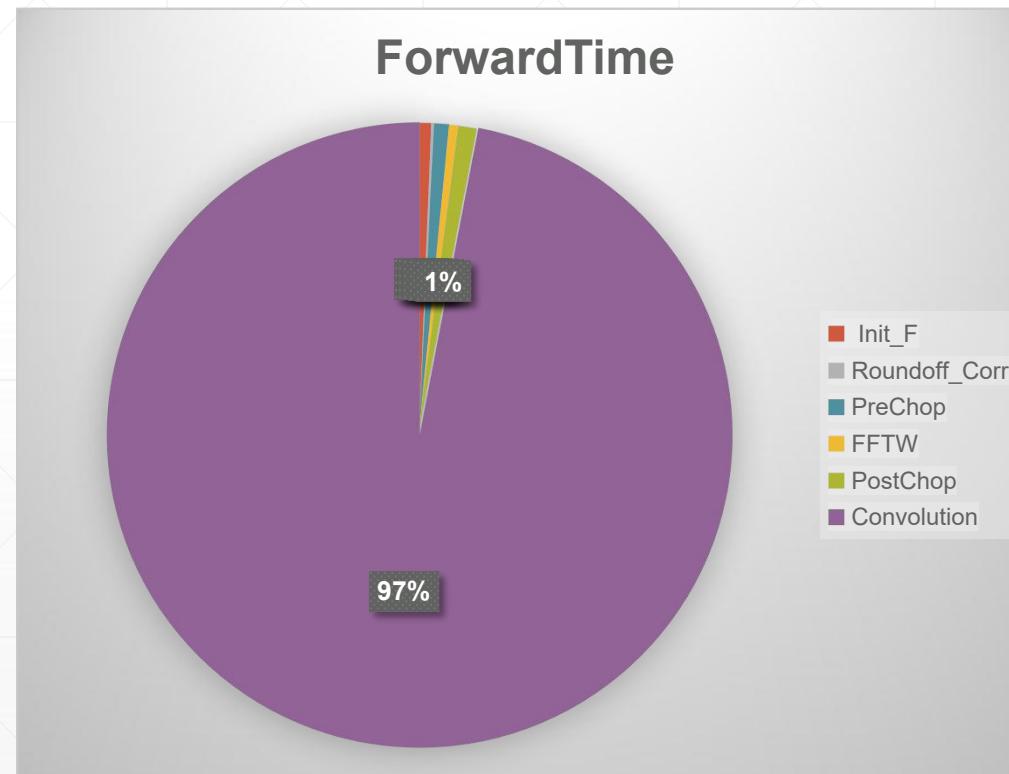
- (1) point-wise scaling of f by s (s is symmetric scaling function)
- (2) execution of a M-point FFT
- (3) convolution interpolation onto the set of  $\omega$ 's

计算一个信号的adjoint NUFFT可以看做是前向NUFFT的逆运算：

- (1) convolution interpolation from the set of  $\omega$ 's
- (2) execution of an inverse M-point FFT
- (3) point-wise scaling of f by s and M

## 2.1 程序及算法分析

统计时间发现，convolution部分耗时是最大的，从卷积部分入手优化



## 2.1 程序及算法分析

Convolution操作其算法如下：

```

1: // Part 1: Common to FWD and ADJ
2: // N2: width of Cartesian Grid
3: // wx[p]: x co-ordinate of sample
4: // Form x interpolation kernel
5: kx = wx[p]
6: x1 = ceil(kx - W)
7: x2 = floor(kx + W)
8: lx = x2 - x1 + 1
9: for i = 0 to lx - 1 do
10:   nx = x1 + i
11:   kx[i] = mod(nx, N2)
12:   winX[i] = LUT(abs(nx - kx))
13: end for
14: // Form y and z interpolation kernels
15: ...

```

```

1: // Part 2a: Only in FWD Convolution
2: // Perform separable convolution
3: // raw[p]: Sample value
4: // f[x, y, z]: Cartesian Grid point
5: raw[p] = 0
6: for x = 0 to lx - 1 do
7:   for y = 0 to ly - 1 do
8:     for z = 0 to lz - 1 do
9:       raw[p] += f[kx[x], ky[y], kz[z]]
          * winX[x] * winY[y] * winZ[z]
10:    end for
11:  end for
12: end for

```

```

1: // Part 2b: Only in ADJ Convolution
2: // Perform separable convolution
3: // raw[p]: Sample value
4: // f[x, y, z]: Cartesian Grid point
5:
6: for x = 0 to lx - 1 do
7:   for y = 0 to ly - 1 do
8:     for z = 0 to lz - 1 do
9:       f[kx[x], ky[y], kz[z]] += raw[p]
          * winX[x] * winY[y] * winZ[z]
10:    end for
11:  end for
12: end for

```

Figure 2. Convolution Code

## 2.1 程序及算法分析

对于forward convolution，可以对样本划分后直接OpenMP并行。

但是对于adjoint convolution 而言，多个样本可能会更新同一网格点，如果多个线程同时计算可能导致写回错误的值。

可以考虑各自线程维护一个副本，最终进行归约，但是因为空间很大，开销巨大。如果对冲突数据使用互斥锁，也会有巨大的额外开销。

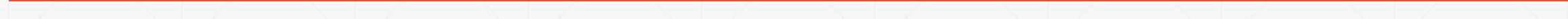
所以本次优化主要围绕对adjoint convolution 并行展开。

## 2.2 MKL&FFTW

由于代码使用了FFTW接口，所以需要先编译可用的fftw库供程序调用。

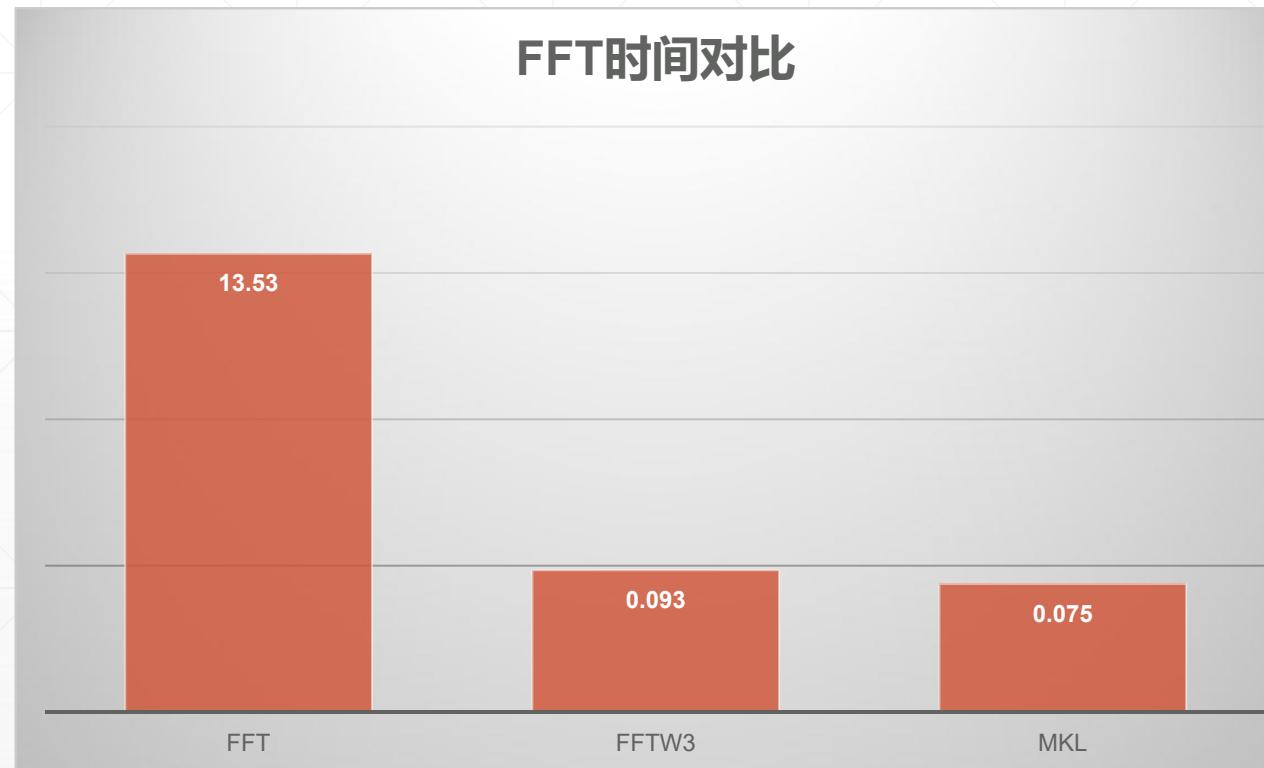
这里采用 [FFTW3 Interface to Intel® Math Kernel Library](#)，具体编译方法参考[官网文档](#)。

同时我们也实现了手动调用了MKL接口版本，跟上述调MKL库fftw3接口性能一样，不再赘述。



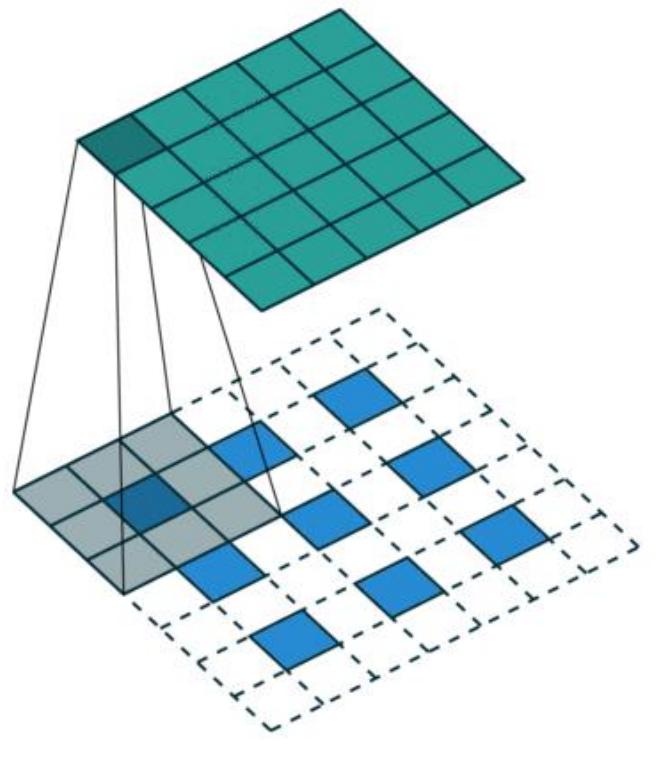
## 2.2 MKL&FFTW

对比普通FFT实现，MKL实现将实际从原来13.52s降低到0.07s，加速了约189.8倍。对比FFTW3库实现，MKL也有着一定的优势：



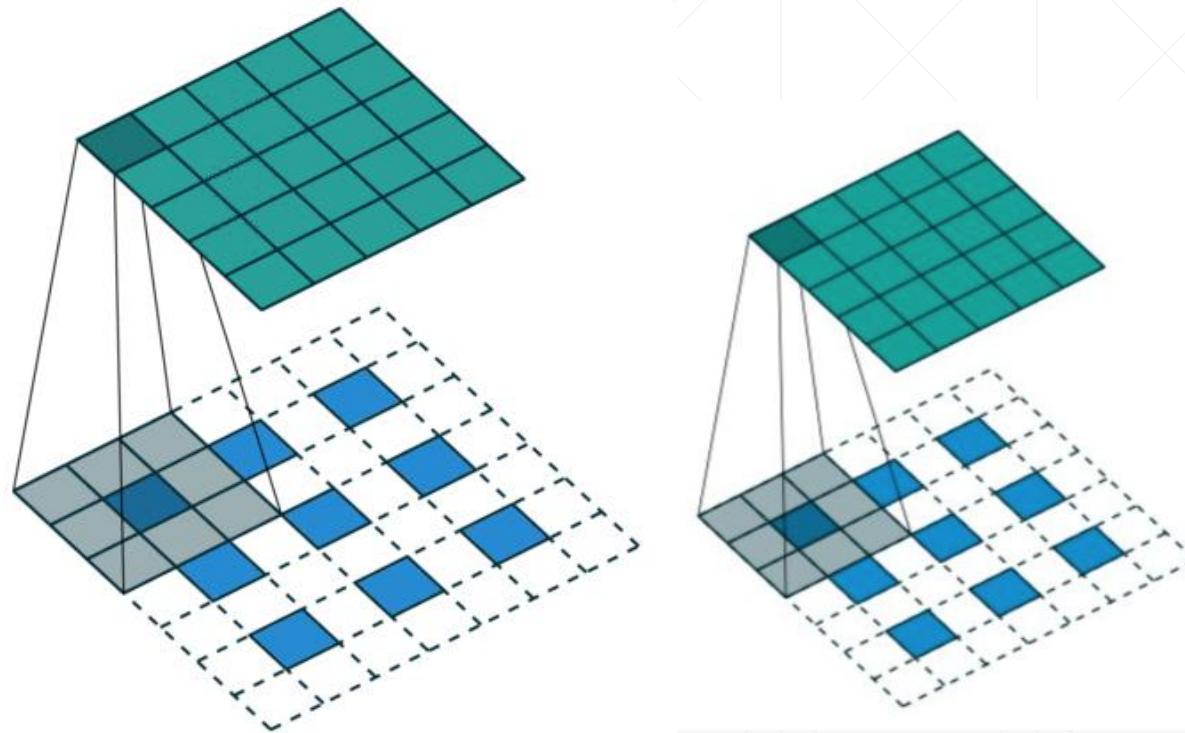
## 2.3 并行任务划分

前面提到，Adjoint convolution部分有反卷积，如图以二维 $3 \times 3$ 反卷积为例：



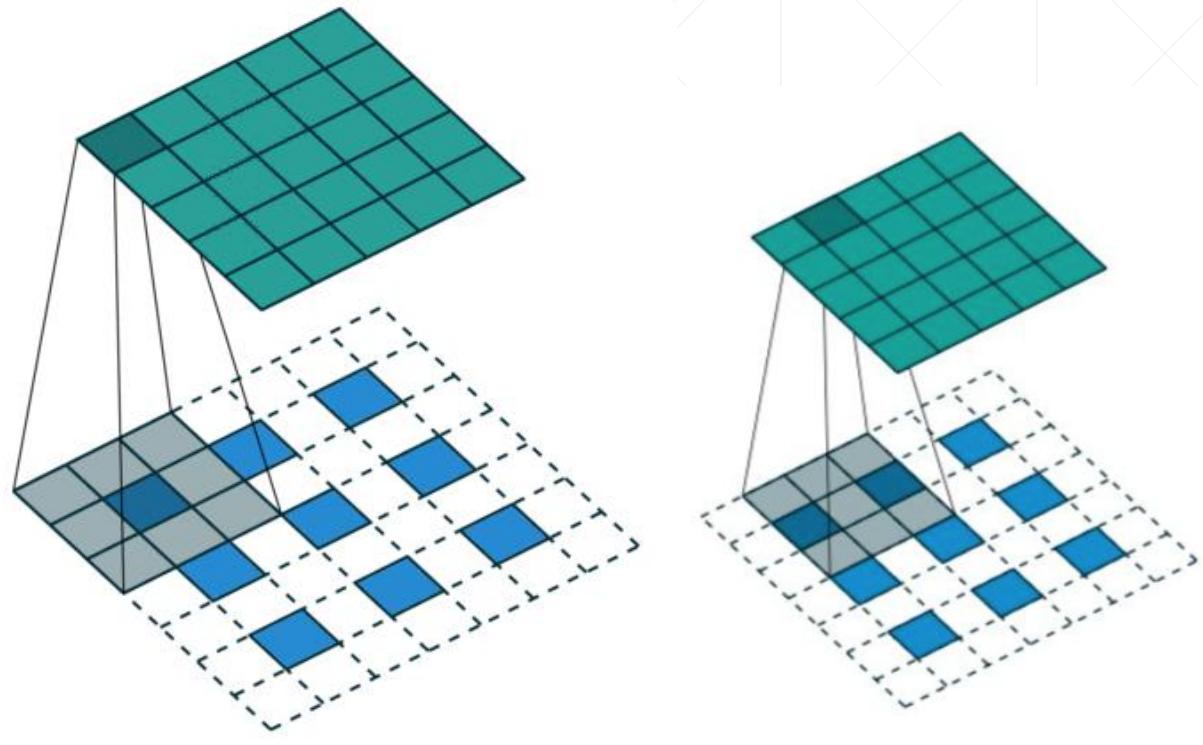
## 2.3 并行任务划分

前面提到，Adjoint convolution部分有反卷积，如图以二维 $3 \times 3$ 反卷积为例：



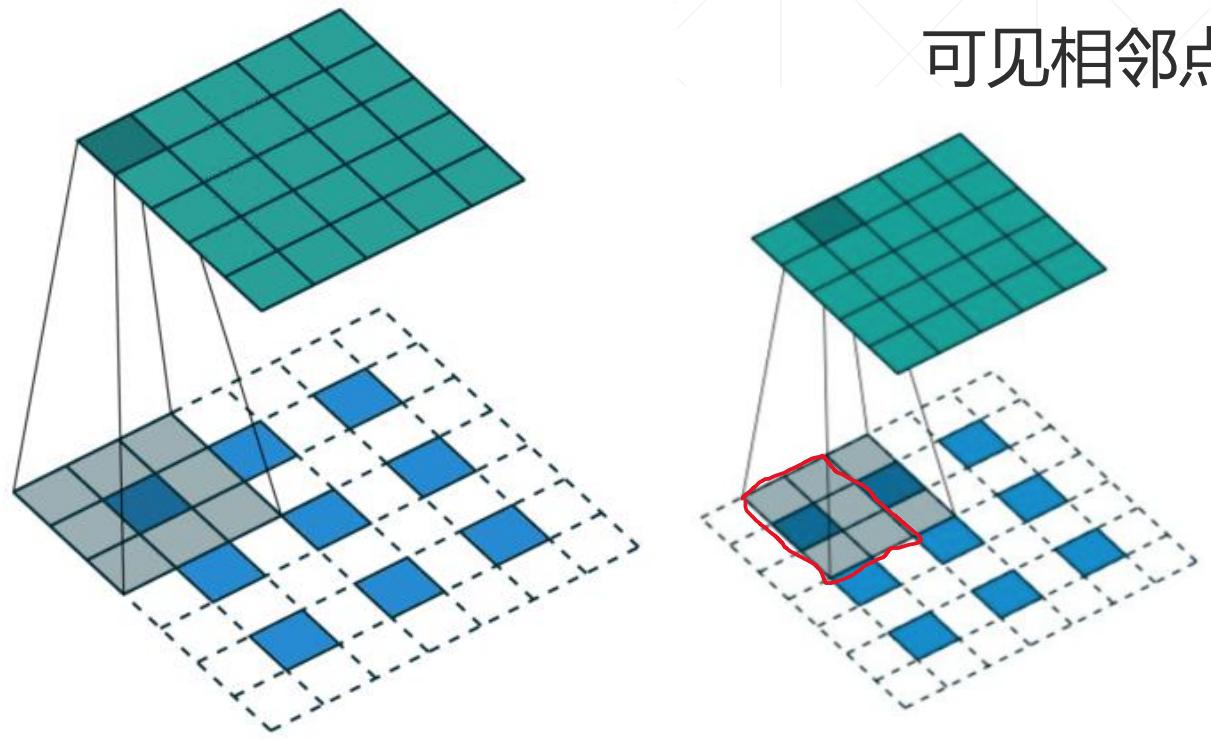
## 2.3 并行任务划分

前面提到Adjoint convolution部分有反卷积，如图以二维 $3 \times 3$ 反卷积为例：



## 2.3 并行任务划分

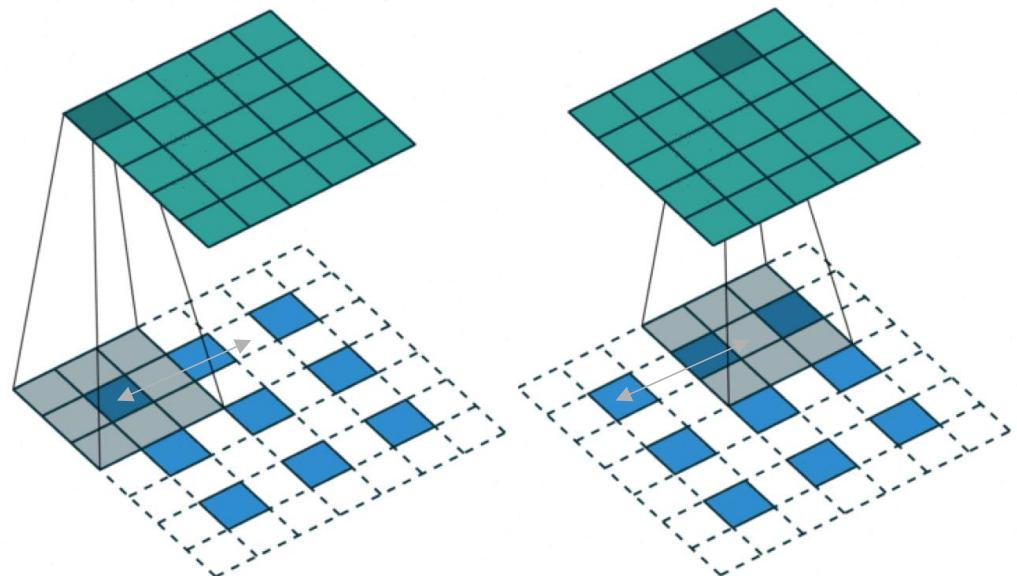
前面提到Adjoint convolution部分有反卷积，如图以二维 $3 \times 3$ 反卷积为例：



可见相邻点反卷积时会有重叠写入部分。

## 2.3 并行任务划分

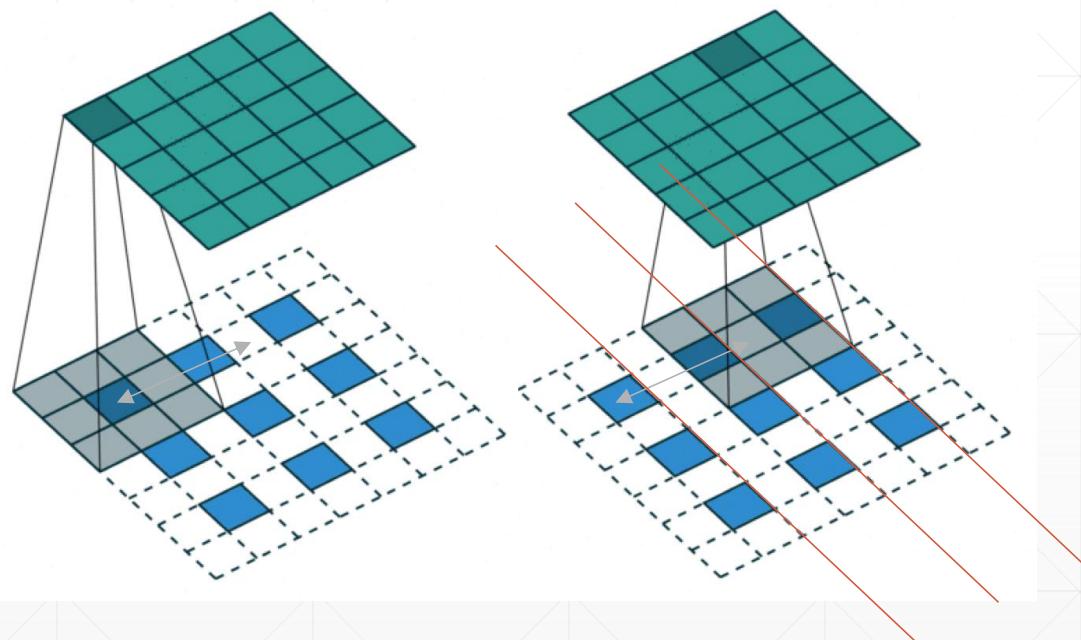
前面提到Adjoint convolution部分有反卷积，如图以二维 $3 \times 3$ 反卷积为例：



如果点之间的间隔超出2倍卷积半径，  
则不会有写入冲突

## 2.3 并行任务划分

前面提到Adjoint convolution部分有反卷积，如图以二维 $3 \times 3$ 反卷积为例：



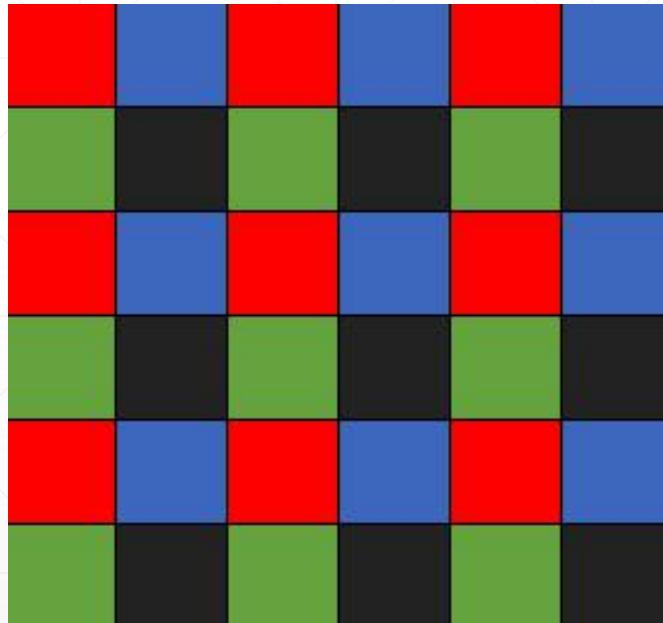
如果点之间的间隔超出2倍卷积半径，  
则不会有写入冲突

所以，分块时每一块的最小大小应超过  
2倍卷积半径，如此，**非相邻分块**就能  
无冲突并行执行了。

## 2.3 并行任务划分

分块之后，相邻分块也是有写入冲突的，所以并行时需要避开相邻的块

这里采用分块队列调度的方法避免冲突：（以二维为例）



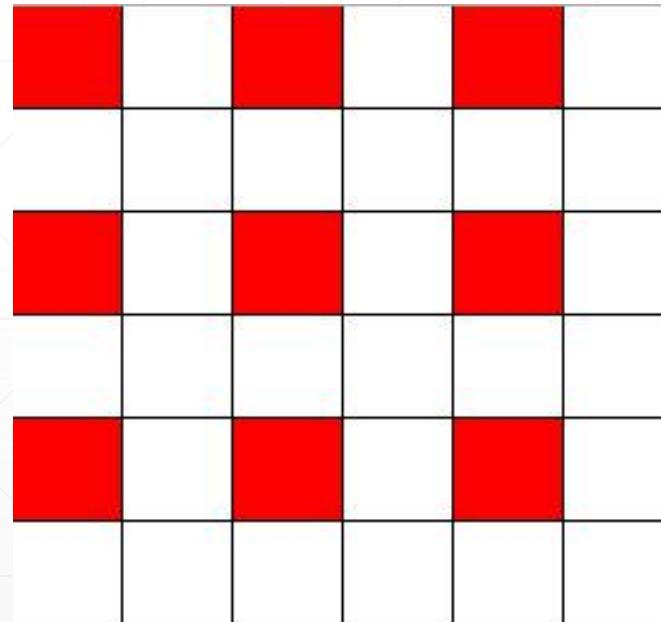
首先把不相邻块分为4组，如图中红、  
蓝、绿、黑。

（三维情况对应 $2^d=8$ 组）

## 2.3 并行任务划分

分块之后，相邻分块也是有写入冲突的，所以并行时需要避开相邻的块

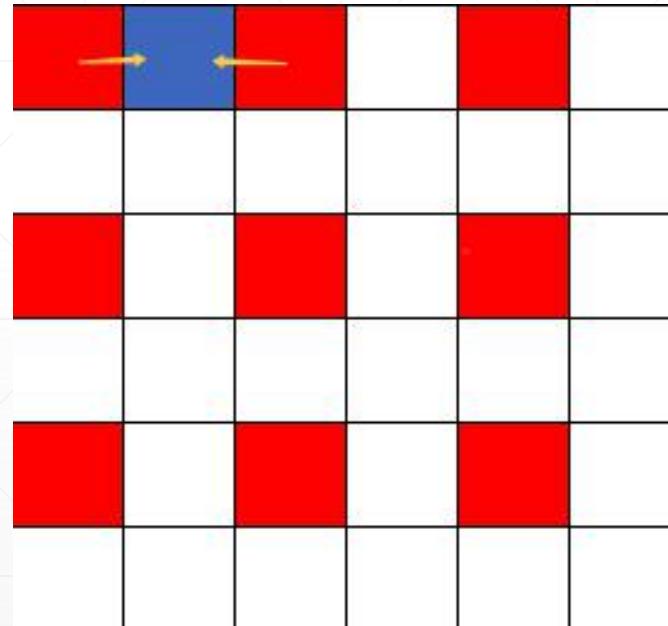
这里采用分块队列调度的方法避免冲突：（以二维为例）



第一轮把所有红色的块的加入队列中，  
各个线程从队列中拿取分块执行，

## 2.3 并行任务划分

分块之后，相邻分块也是有写入冲突的，所以并行时需要避开相邻的块  
这里采用分块队列调度的方法避免冲突：（以二维为例）

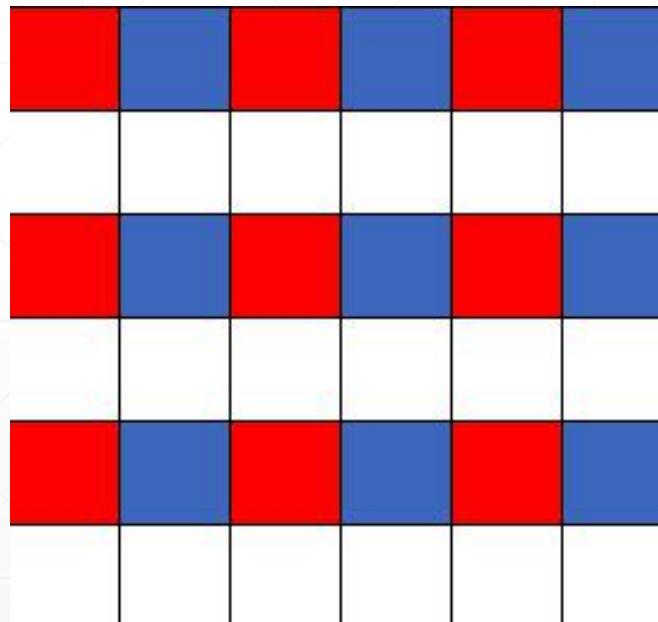


第一轮把所有红色的块的加入队列中，各个线程从队列中拿取分块执行，执行完成后，检测其左右的红色块是否也执行完成。  
如果执行完成，那么就把这两个**红色**块之间的蓝色块加入队列中。随后从队列中继续拿取分块执行。

## 2.3 并行任务划分

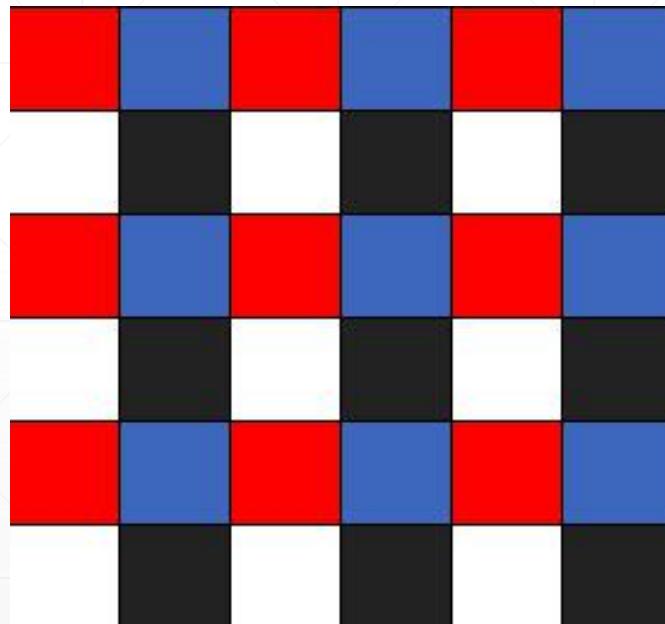
分块之后，相邻分块也是有写入冲突的，所以并行时需要避开相邻的块

这里采用分块队列调度的方法避免冲突：（以二维为例）



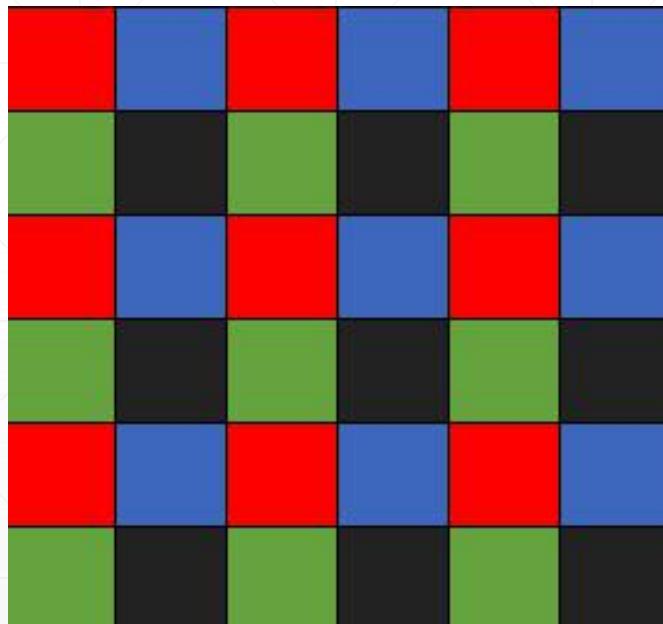
如此红色计算完计算蓝色。

## 2.3 并行任务划分



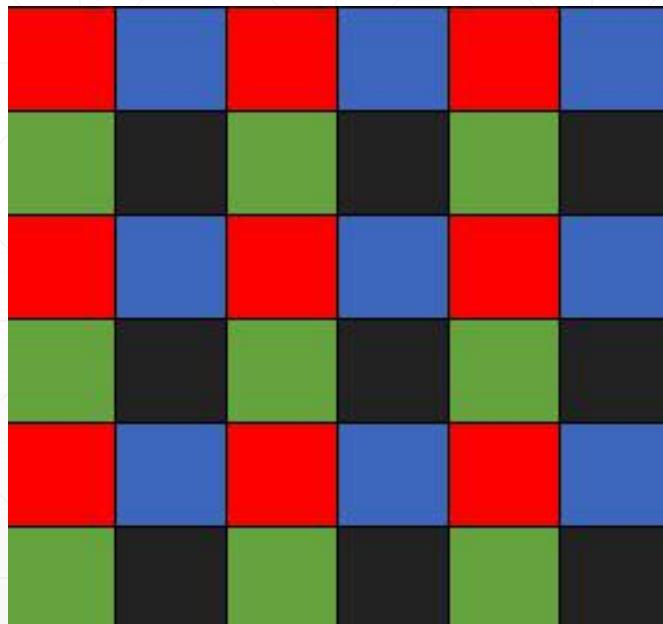
蓝色块执行完成后检测上下蓝色块，把黑色块加入队列。

## 2.3 并行任务划分



同理，黑色执行完加入绿色。直到全部计算完成。

## 2.3 并行任务划分



为了保证多个线程更新队列时不冲突，这里使用了锁来控制，获得锁的线程才能更新。由于只有在计算完时才需要获取锁，所以冲突较小，开销极低。

## 2.3 并行任务划分

- 为了使负载更均衡，可以使用优先队列，先计算最大的块
- 针对不同线程所属NUMA，优先指派只属于当前numa的任务



## 2.3 并行任务划分

- 加锁实现的优先队列
- 加锁有多种实现方式：
  - 可以用c++ std的mutex互斥锁；也可以用OpenMP的omp\_lock\_t线程互斥锁；
  - 也可以直接用omp critical 临界区（critical命令最终会被翻译成加锁和解锁的操作）

## 2.3 并行任务划分

- 队列初始化：使用主线程向队列中加入第一批可执行区块，并直接分配区块给各个线程（分块的时候应保证第一轮可执行块大于线程数量）
- 从队列获取计算任务：

```
while (已完成任务<总任务) {  
    if 当前线程有任务： 执行任务();  
    else  
        #pragma omp critical {  
            if 队列为空 {  
            } else {  
                记下队首任务信息;  
                弹出队首;  
            }  
        }  
    }  
    // 继续轮询直到全部完成  
}
```

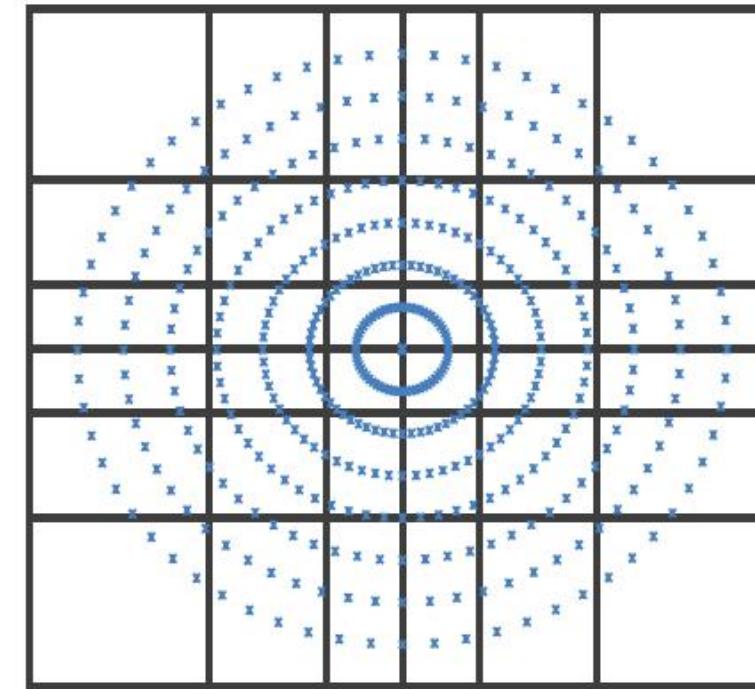
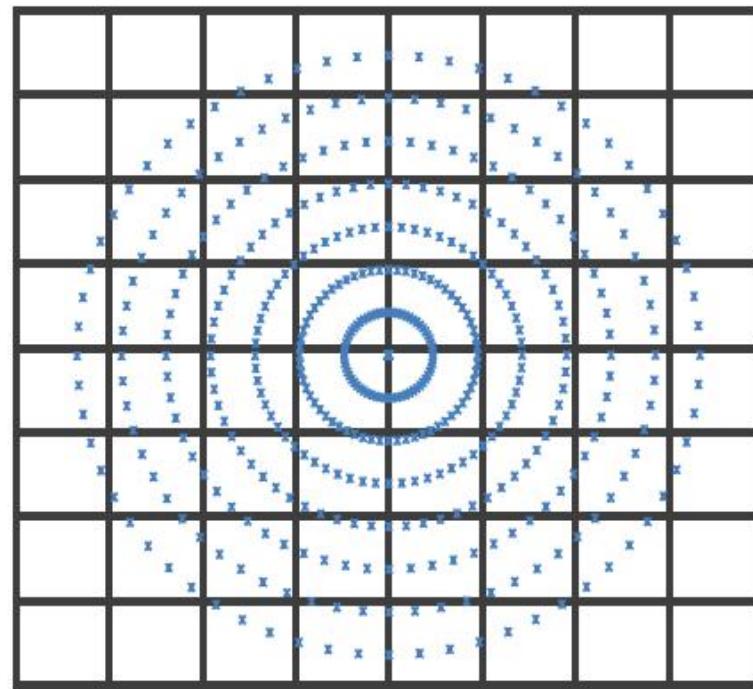
## 2.3 并行任务划分

- 执行任务完成时更新队列，把后续可执行区块加入队列：

```
#pragma omp critical {
    标记已完成区块();
    if 检测相邻区块，发现依赖已消除并且未执行 {
        将该区块加入队列();
    }
    if 队列不为空 {
        记录队首任务信息;
        弹出队首;
    }
}
// 回到前面的while中
```

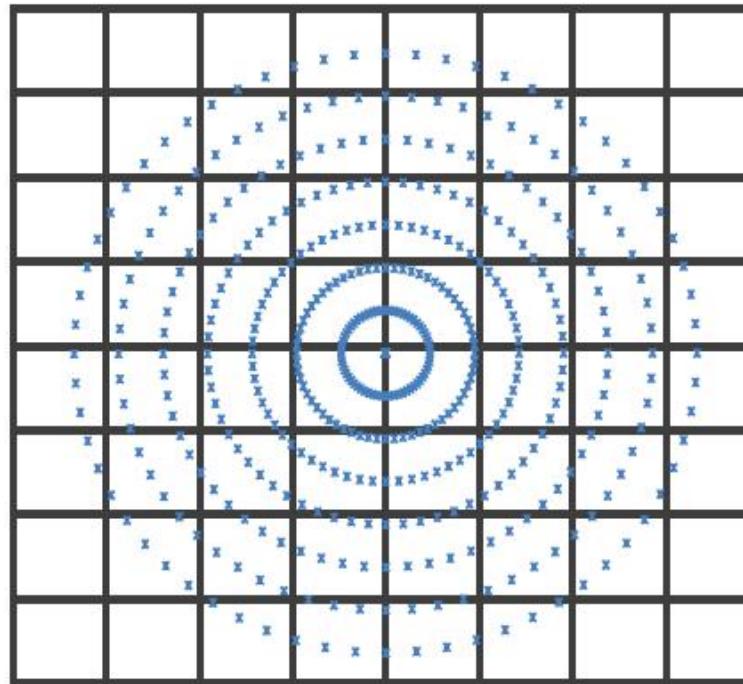
## 2.3 并行任务划分

另外，有些算例的样本 $p$ 再三维空间时分布并不均匀，如果直接等比分块，会导致负载极其不均衡。所以可以采用变宽分块的方式，让点密集处的块更小



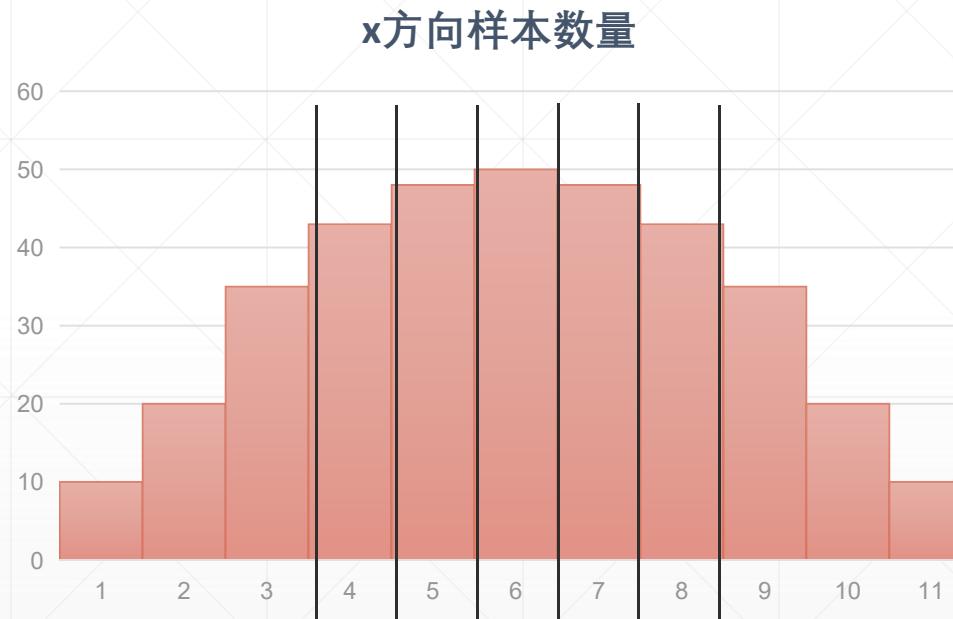
## 2.3 并行任务划分

这里根据维度样本数量分布直方图来划分。如图：



## 2.3 并行任务划分

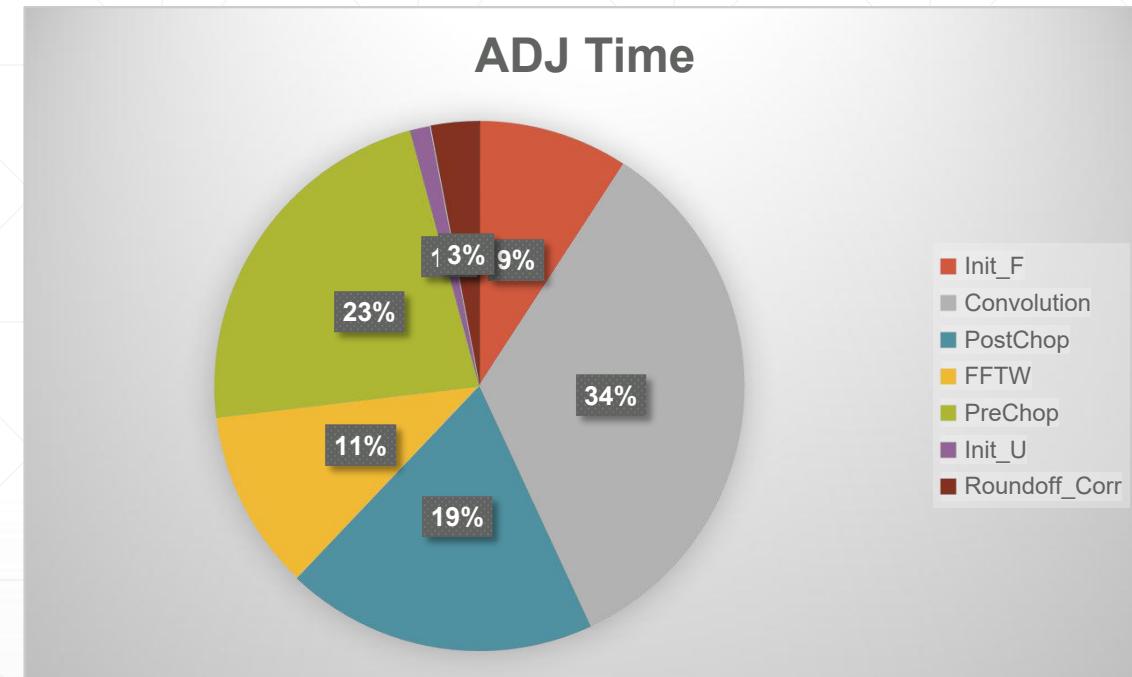
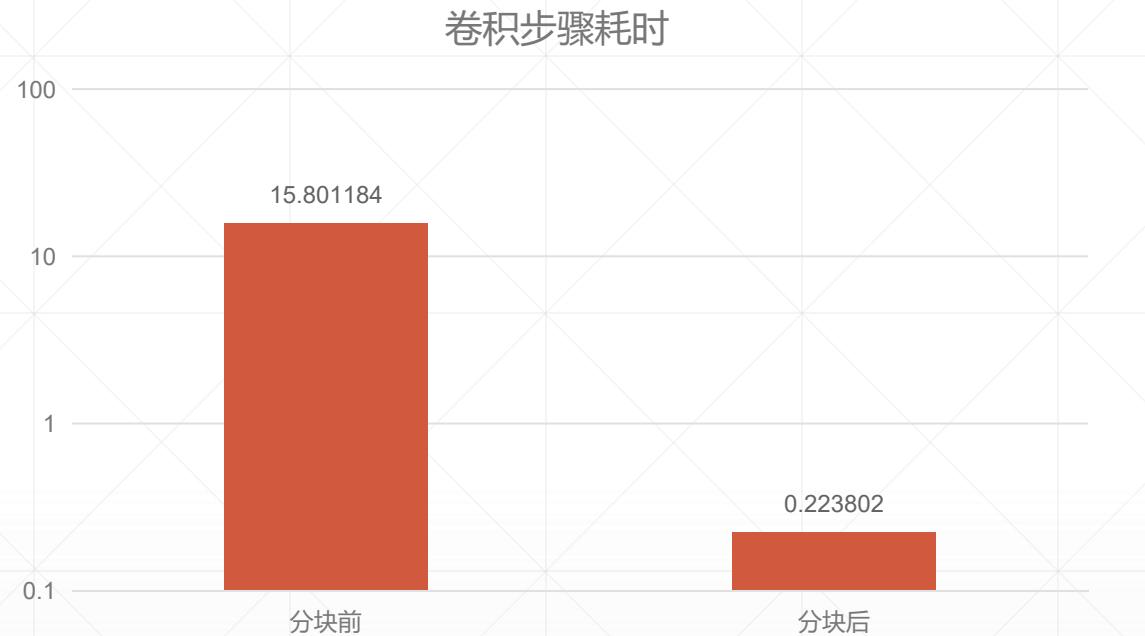
这里根据维度样本数量分布直方图来划分。如图：



对该维度累积数据逐个累加，当改数量超过期望的平均分块大小时，当前坐标即为该维度的分块点，清零累加值，继续寻找下一个分快点。

对于第二、第三维度同理。

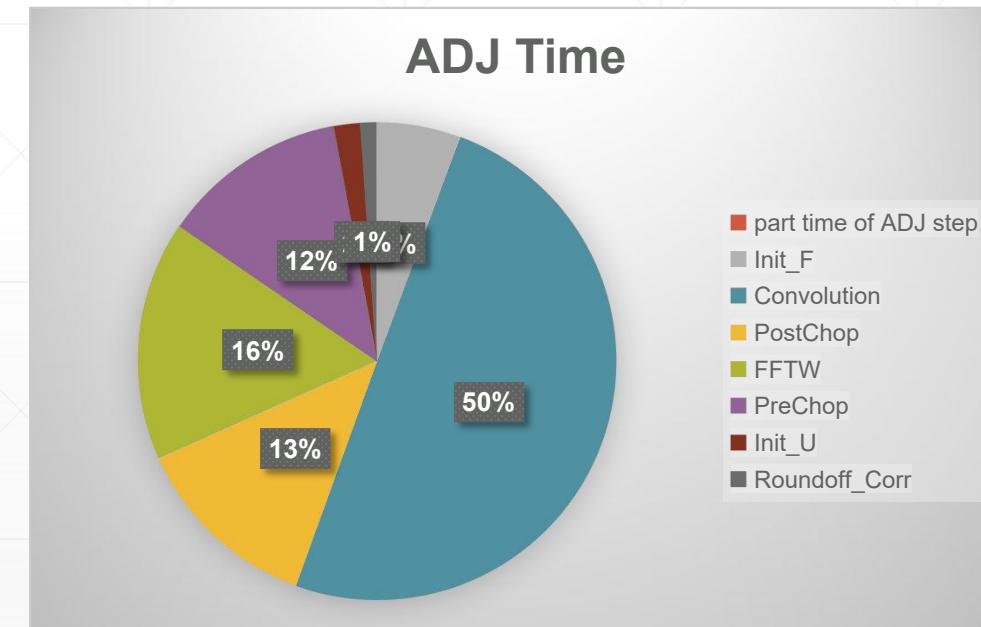
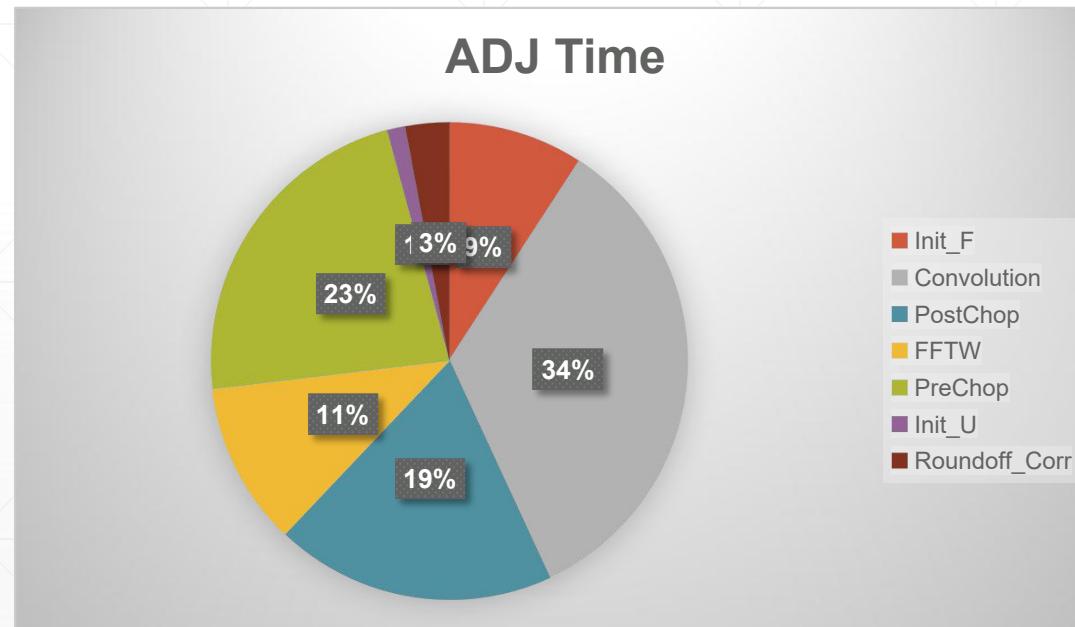
## 2.3 并行任务划分



经过上述的分块和多线程并行，取得了近70倍的加速，  
此时原来其他部分也逐渐变成热点了

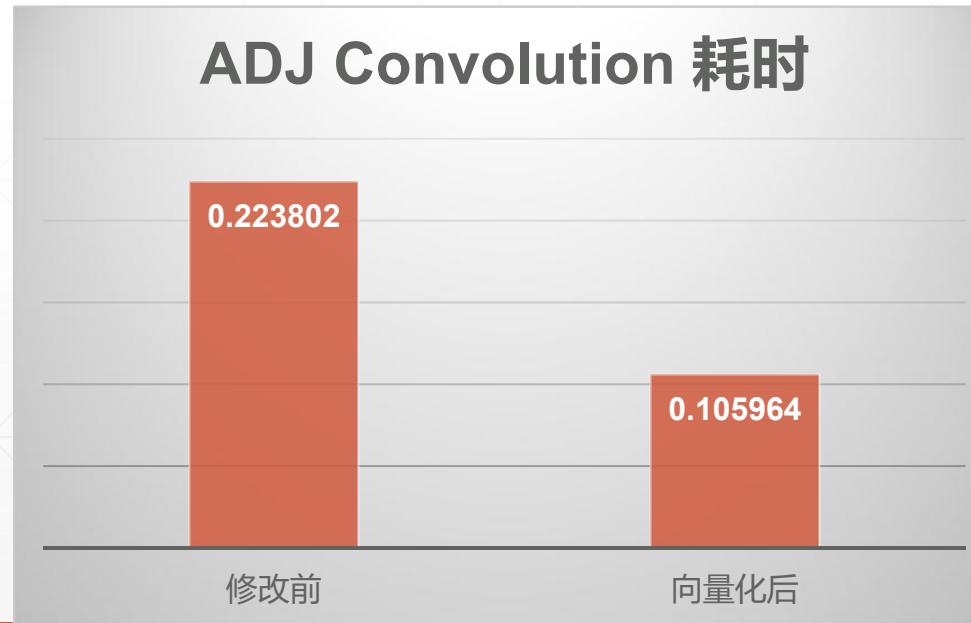
## 2.4 除卷积外部分并行化

这里我们对preChop和postChop以及Roundoff\_Corr也使用omp并行，  
迭代间的数据处理也进行了并行。



## 2.5 向量化

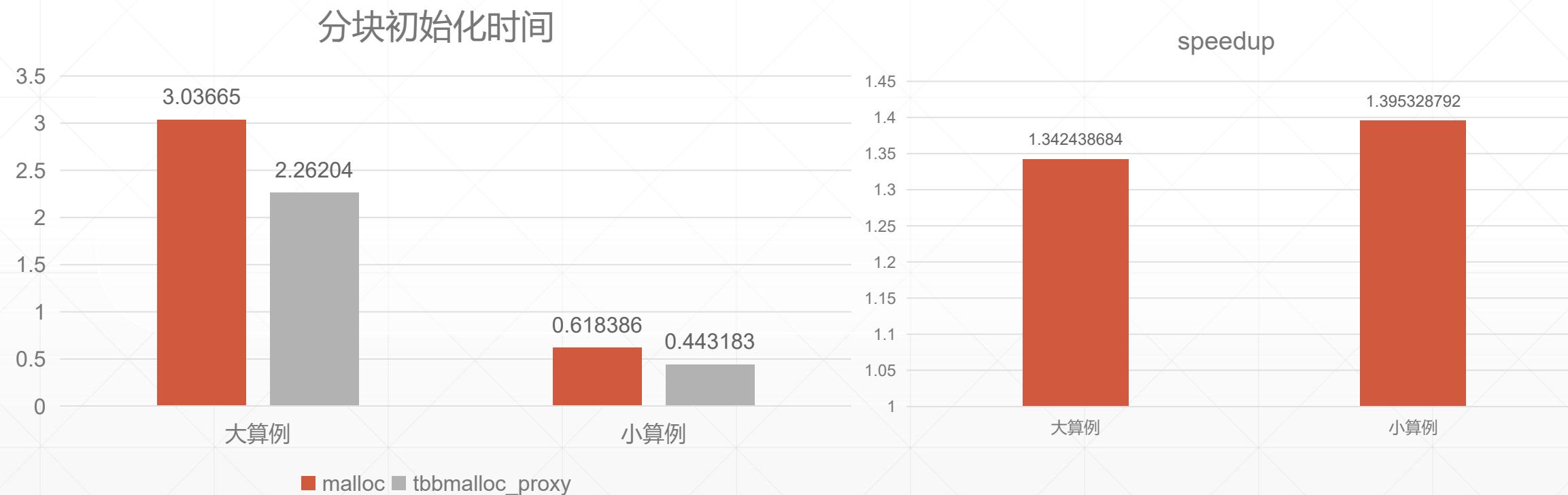
- 上图中，卷积部分时间占比还是很大，所以我们对卷积核进行手动向量化，对Y、Z方向的循环进行展开，凑够更多的无依赖计算，填满流水线。



取得了接近1倍左右的加速

## 2.6 其他优化

- 使用Intel TBB Malloc代替原始的malloc，减少内存申请耗时



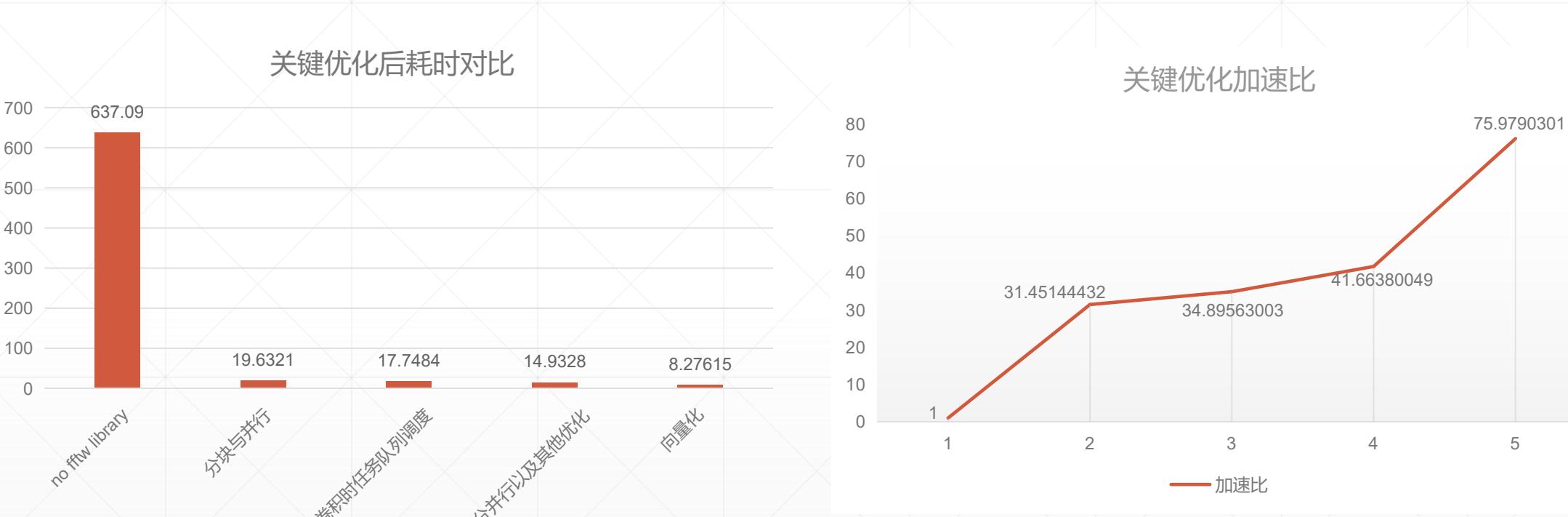
## 2.6 其他优化

- 设置KMP\_AFFINITY=compact 提高线程调度性能
- 以及其他一些常数级的优化



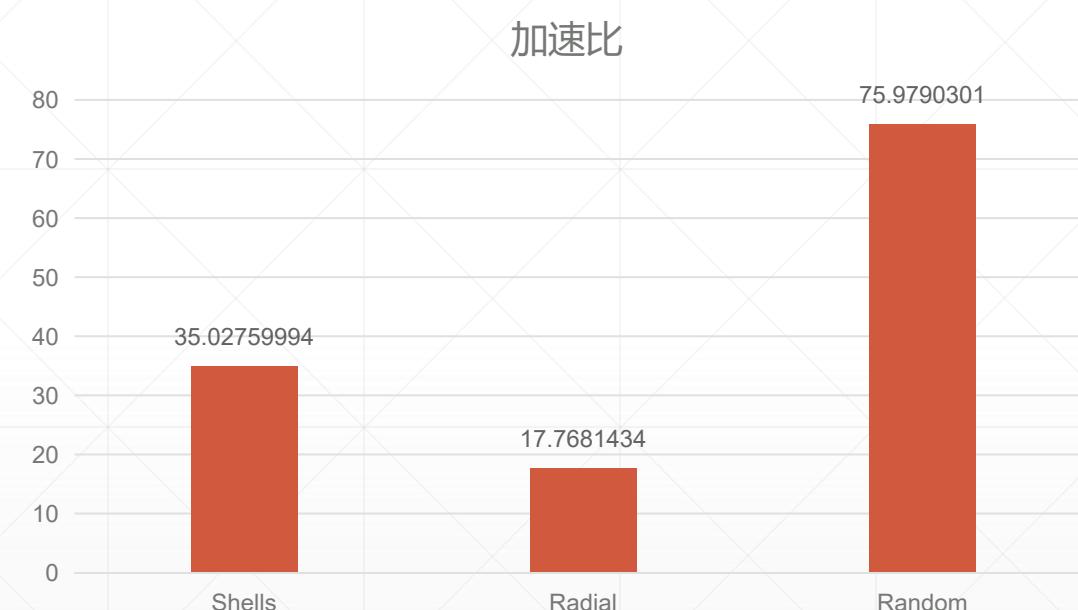
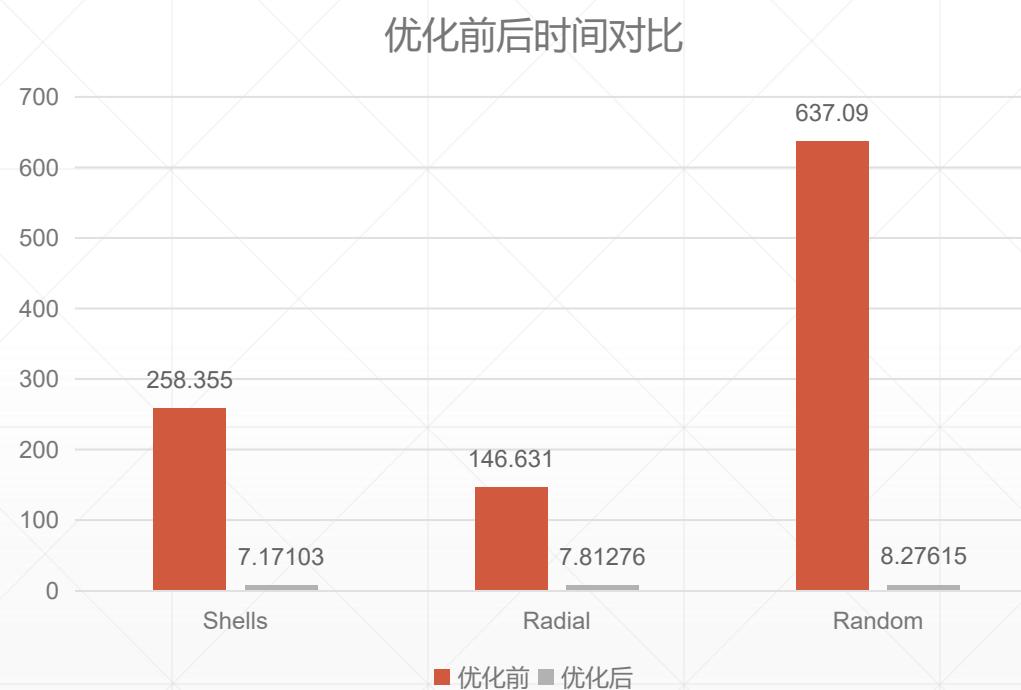
## 2.6 最终优化结果和总结

通过上述一系列优化，最终效果如图所示：（20次迭代总时间）

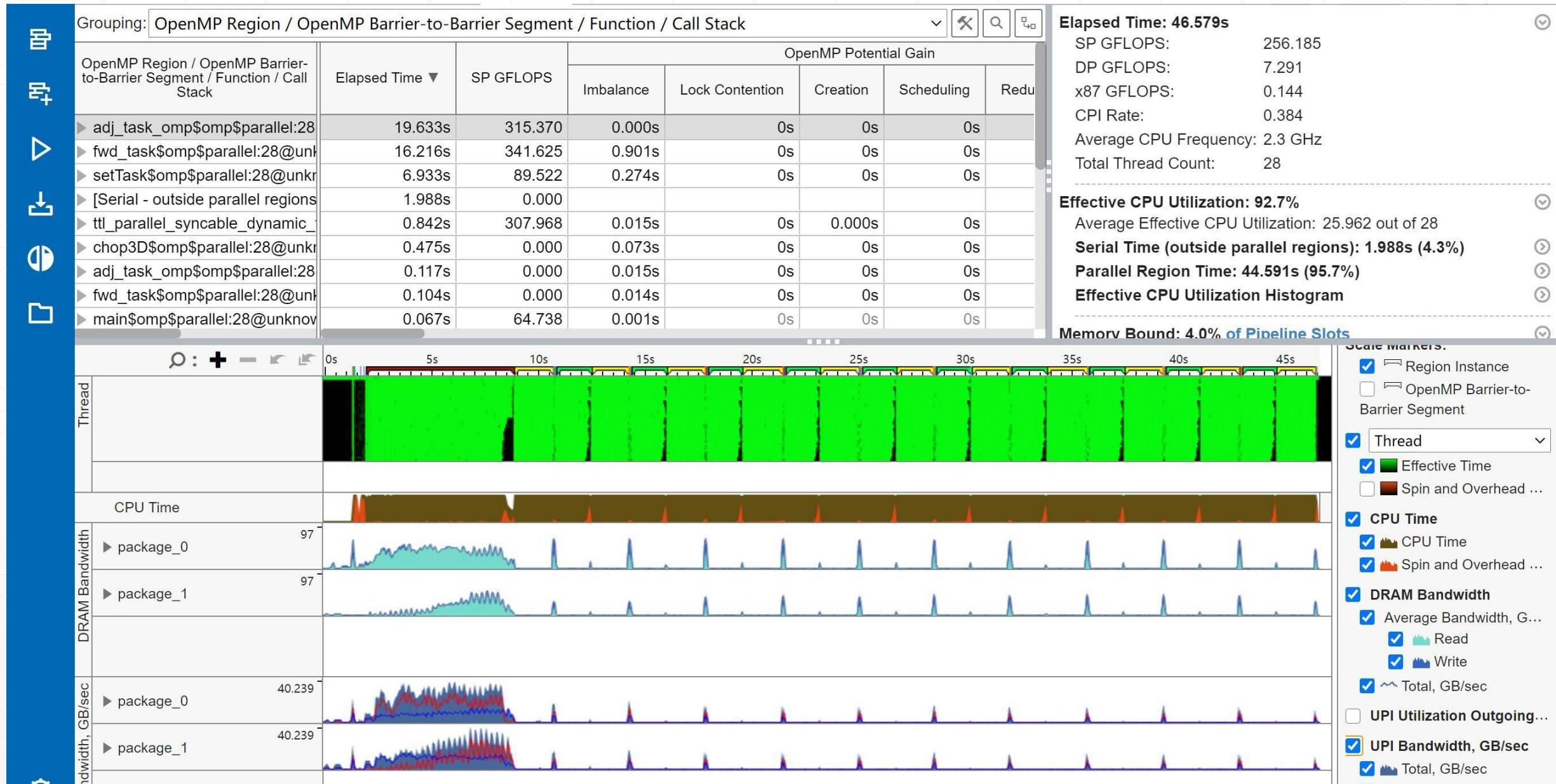


## 2.6 最终优化结果和总结

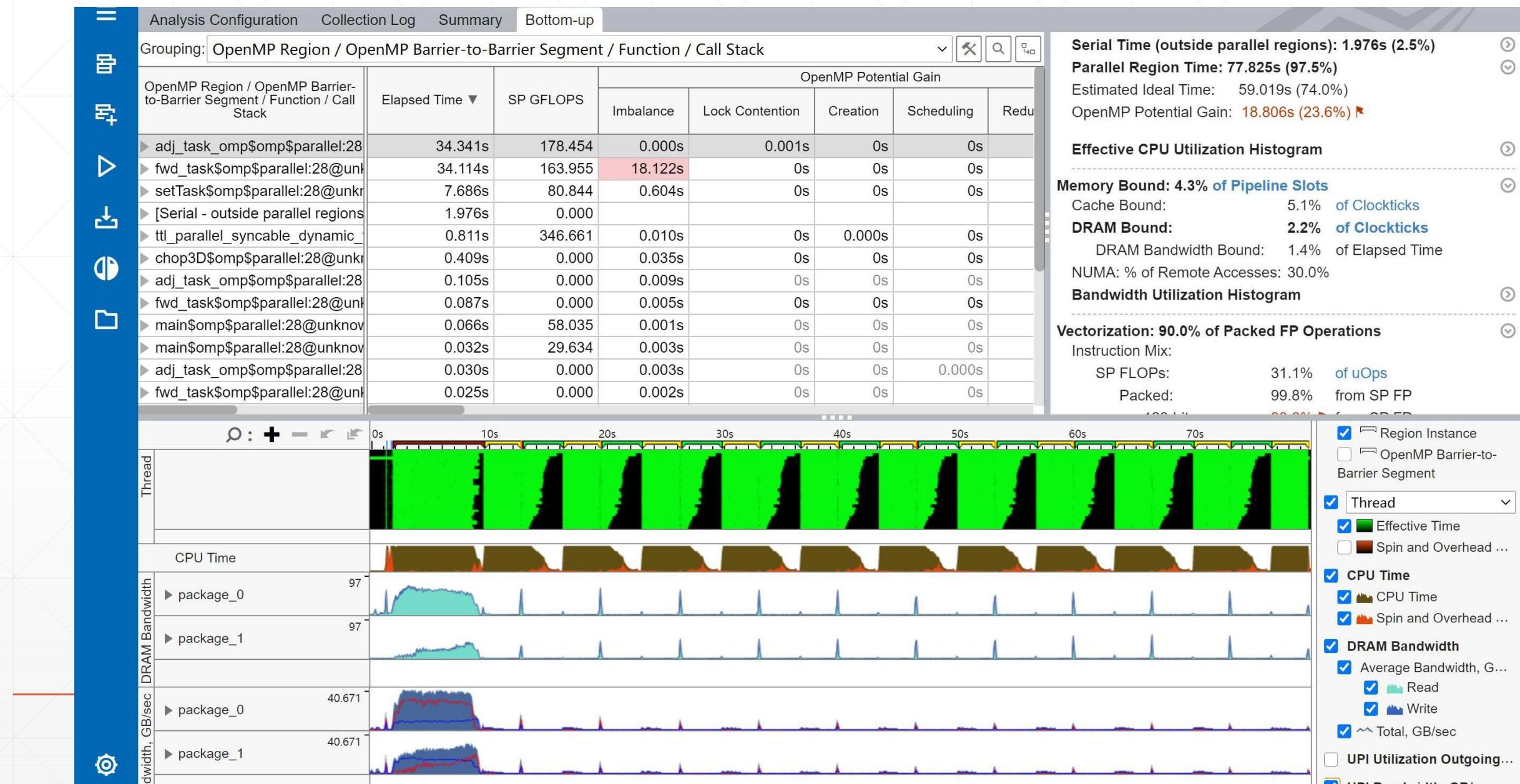
对比不同算例：



## 2.6 最终优化结果和总结-Random算例



## 2.6 最终优化结果和总结-Radial算例



### 3 并行优化经验之谈

- 老生常谈：
- 1、优化流程本身也需要优化  
    工作流程 提高效率 减少熬夜 减少秃头概率
- 2、正确性校验，自定义计时，性能数据采集非常重要  
    这些东西决定了优化方向是否正确，不要优化了半天正确性校验都没有过去
- 3、程序框架应该考虑支持多次反复运行以及不同参数多次测试

接第二条，次次都用vtune采集数据不现实，所以配合自己的计时代码，反复多次试验，不同参数不同方法多次对比，才能找到正确的方向

- 4、不是每次优化实现都是正优化

很多优化尝试可能是负优化，所以不要灰心，多试试总有成的时候，但是要注意结合对计算机体系结构的理解，对负优化做出合理的解释，这样也能找到正确的方法（把错误的方案都排除了，就只剩正确的了）

- 5、你写的代码不一定会按照你写的那样执行  
    编译器会做很多改变，所以当发现事情不对时，考虑反编译看看到底是怎样执行的

# 3 并行优化经验之谈

- 还是要谈：
  - 1、优化的本质是解决计算与访存的冲突

这是我说的，不一定正确，但现代处理器设计确实在解决计算与访存的冲突，cpu的频率越来越高，avx256 512越来越宽速度越来越宽，而内存速度却没有跟上，所以缓存架构也不断在改进，解决这些问题。GPU更是如此，GPU流处理器个数越来越多，与之对应的是显存带宽越来越大，很多优化只要做到尽可能打满访存带宽，就能代表取得一个不错的效果了，当然这里的打满访存带宽的前提是要做好数据局部缓存利用、double buffer计算访存交替掩盖。

- 2、遇到瓶颈试着从数据结构与算法、操作系统、计算机组成、编译原理的角度去考虑

考虑CPU的L1, L2, L3大小带宽，NUMA配置，节点内互联带宽，节点间带宽；内存分布，系统调用开销等然后要利用这些资源需要怎样的数据结构，以及宏观层面有没有数学上等价或者近似的算法实现

- 3、向量化很重要，但不是全部，数据的转移和交换一样重要

向量化很重要，但不是全部，数据的转移和交换一样重要，比如从读到L1在进行数据的格式转换，行列转换以及其他计算；当前深度学习炼丹很火，可以看看注明开源框架的后端是如何实现的，网络层与层直接数据组成形状的变化，输入输出的不同，以及动态图的处理等等

# 总结&答疑

本讲内容：

- 1、上届初赛题stencil讲解
- 2、NUFFT优化案例分享
- 3、一些经验之谈

赛前培训六讲结束~

后期培训与讲座筹备中，敬请期待~



↑↑扫码参加IPCC

↓↓学习交流群



IPCC系列活动意见征集调查~

# IPCC

## 谢谢观看

时间：2021年6月

主讲人：张力越