基于 CUDA 的 Mallat 算法实现

日期：*December 12, 2021*

摘　　要

本文基于Mallat (1989) 提出的 Mallat 快速小波变换算法（Mallat 算法），使用 C++ 编程语言通过 CUDA 平台复现了该 Mallat 快速小波变换算法，实现了对图像的多尺度分解与重构，并对算法进行了并行优化处理，最后验证了算法的有效性并评估了优化效果。

1　背景

本节将简单回顾 Mallat 算法的基本原理，以及利用 CUDA 进行 GPU 并行编程的基本背景。

1.1　Mallat 算法基本原理

与傅里叶变换相比，小波变换的基函数不是三角函数，而是小波函数，其具有变化的频率和有限的持续时间，小波变换在提取频率的同时也保存了波形信息。Mallat (1989) 首次证明了小波变换是多分辨率理论分析的基础，小波变换已经在图像处理领域中广泛用于图像压缩、去噪等。

本文参考了Gonzalez. (2017) 的定义。首先介绍二维离散小波变换（DWT），定义尺度和平移基函数：

$$\varphi_{j,m,n}(x,y) = 2^{j/2}\varphi(2^j x - m, 2^j y - n) \tag{1}$$

$$\Psi^i_{j,m,n}(x,y) = 2^{j/2}\Psi(2^j x - m, 2^j y - n), i = H, V, D \tag{2}$$

于是 $M \times N$ 大小的图像 $f(x,y)$ 的 DWT 定义为：

$$W_\varphi(j_0,m,n) = \frac{1}{\sqrt{MN}} \sum_{x=0}^{(} x = M-1) \sum_{y=0}^{N-1} f(x,y)\varphi_{j_0,m,n}(x,y) \tag{3}$$

$$W^i_\Psi(j,m,n) = \frac{1}{\sqrt{MN}} \sum x = 0^{(} x = M-1) \sum_{y=0}^{N-1} f(x,y)\Psi^i_{j,m,n}(x,y), i = H, V, D \tag{4}$$

其中 i 表示方向小波，可以取 H、V、D 分别对应水平、垂直、对角；j 是尺度、j0 是初始尺度；$W_\varphi(j_0)$ 系数定义了图像 $f(x,y)$ 在尺度 j_0 处的近似；$W^i_\Psi(j)$ 系数则为图像在尺度 j 处 i 方向的细节。

可以进一步定义离散小波反变换（IDWT），实现从小波系数到图像的重建：

$$
\begin{aligned}
f(x,y) = & \frac{1}{\sqrt{MN}} \sum_{m} \sum_{n} W_{\varphi}(j_0,m,n)\varphi_{j_0,m,n}(x,y) \\
& + \frac{1}{\sqrt{MN}} \sum_{i=H,V,D} \sum_{j=j_0}^{\inf} \sum_{m} \sum_{n} W_{\Psi}^{i}(j,m,n)\Psi_{j,m,n}^{i}(x,y)
\end{aligned}
\tag{5}
$$

Mallat 算法是一种 DWT 快速实现方式，Mallat 算法的实现如图1所示。(a) 为分解部分，其中 $A_{2^j}^{d} f$ 即前文记法中的 $W_{\varphi}(j,m,n)$，$D_{2^j}^{i} f$ 即为 $W_{\Psi}^{i}(j,m,n)$。分解可以从图像在 j 尺度的近似获得 j-1 尺度的近似系数以及高频系数。分解的基本过程为使用分解低通和高通分解滤波器分别卷积原始图像 j 尺度的 $A_{2^j}^{d} f$ 系数的每一行，之后对其列分别进行下采样，得到两组系数。这两组系数分别刻画了水平低频信息和水平高频信息。再次分别用低通和高通分解滤波器卷积两组系数的每一列，之后对其行分别进行下采样，得到四组系数，包括低频信息 $W_{\varphi}(j-1,m,n)$ 以及三组高频信息 $W_{\Psi}^{i}(j-1,m,n)$。重构可以从 j-1 尺度的四组系数重构 j 尺度的低频近似。重构的基本过程与分解过程相逆，其恢复滤波器也为分解滤波器的倒叙。重构中，首先对四组系数进行行上采样，并分别用低通、高通恢复滤波器卷积图像的每一列，将结果两两相加可以得到两组系数。将这两组系数分别进行列上采用，并分别用低通、高通恢复滤波器卷积图像的每一行，并将结果相加，最后得到了 j 尺度的低频近似系数。

Tenllado et al. (2008) 尝试利用 GPU 实现了 2-D DWT 算法，使用 NVIDIA FX5950 Ultra 和 NVIDIA 7800 GTX 器材，其加速比在 1.2 到 3.4 之间。

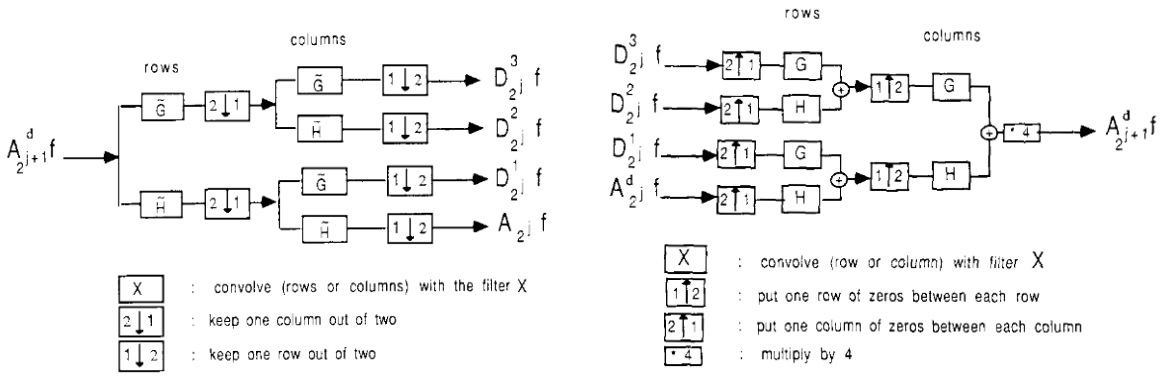
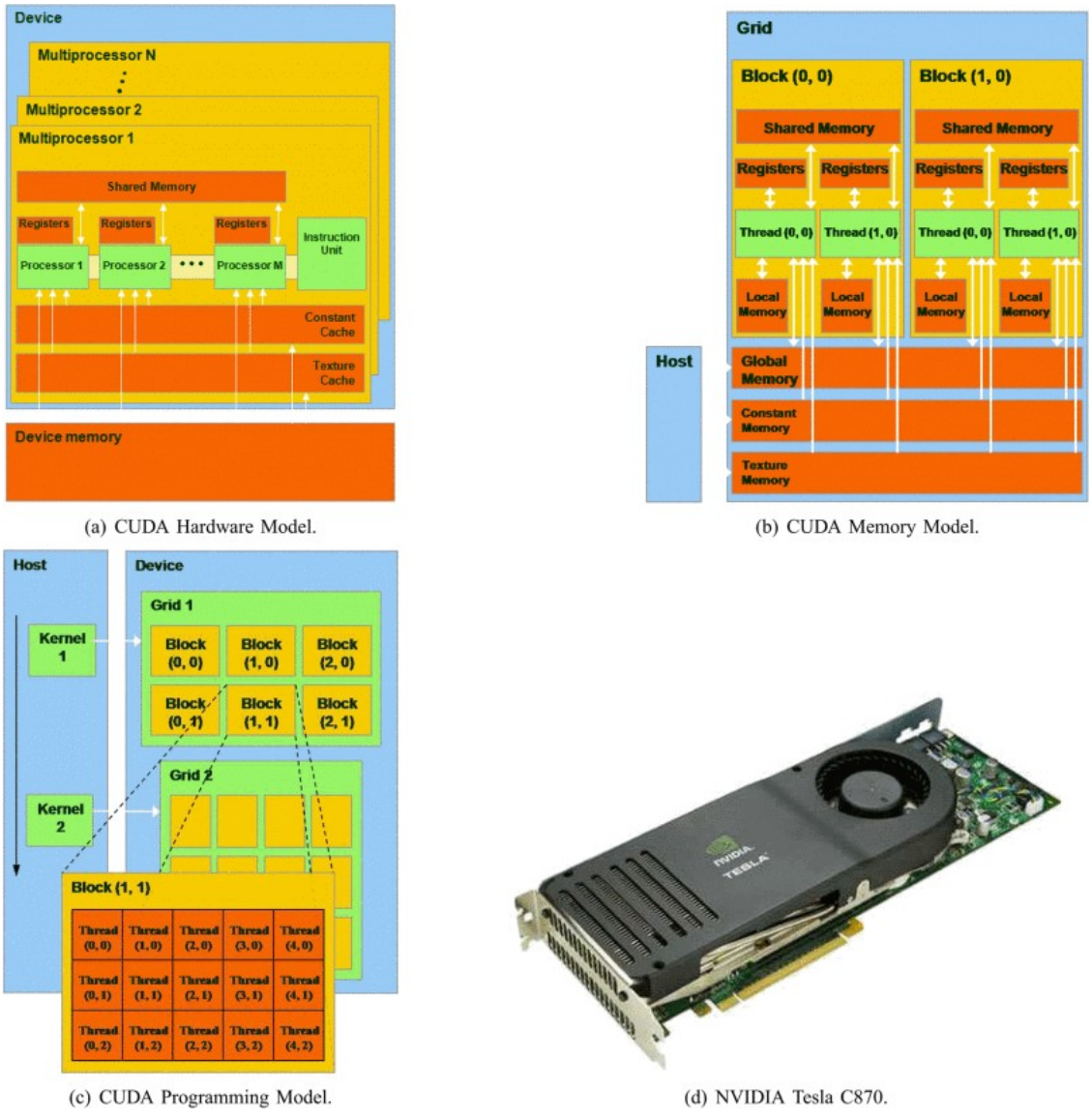


图 1: Mallat (1989) 小波分解重构基本过程。(a) 分解;(b) 重构。

1.2 CUDA 基本背景

近几年中 GPU（通用图像处理单元）技术迅速发展，在计算机视觉领域、计算机图形学领域、数据挖掘领域以及物理仿真等领域日益广泛。CUDA（统一计算设备架构）是 Nvidia 公司推出的通用并行计算架构，旨在提升程序在不同架构 GPU 下的可拓展性。CUDA 使用 C++ 编程，也提供 FORTRAN, DirectCompute, OpenACC 等接口。根据Franco et al. (2009)，从 CUDA 的硬件构成角度

看，每个 CUDA 兼容设备都包含了一组多处理器内核，每个处理器内核多台处理器可以处理多个线程（见图2 (a)）；从 CUDA 的内存模型角度看，包括 Host 内存和设备内存，设备内存包括全局内存、常量内存、纹理内存，还包括每个 Block 中的共享内存、线程中的本地局部内存（见图2 (b)）；从 CUDA 的编程模型角度看，Host 可以调用多个核函数，由每个 Grid 以多线程方式执行同一个核函数。



(a) CUDA Hardware Model.

(b) CUDA Memory Model.

(c) CUDA Programming Model.

(d) NVIDIA Tesla C870.

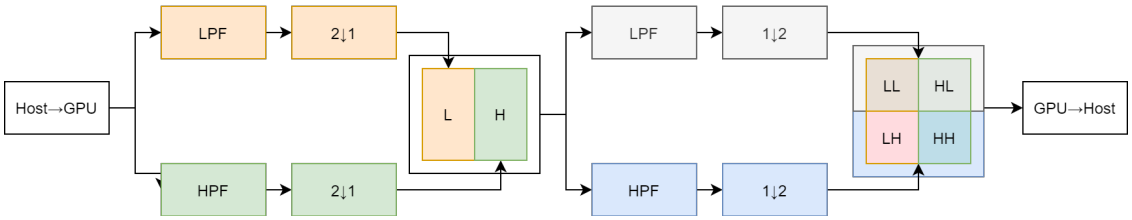
图 2: CUDA 基本架构Franco et al. (2009)

2 使用 CUDA 进行并行优化

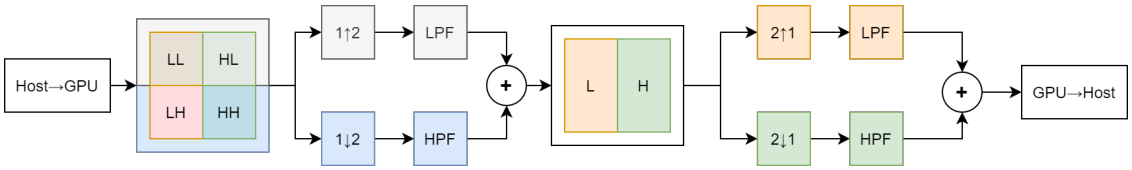
2.1 并行算法总体结构

使用 CUDA 进行加速时，由于需要将数据从 Host（宿主机）搬运到 GPU，这会耗费额外的内存搬运时间，因此应当尽量避免内存的搬运。所以优化算法的处理过程尽量集中在 GPU 中，避免频繁的内存搬运。因此这里在每次分解和重构过程只在初始前和完成后进行两次搬运。为了提升性能，对于经常访问地内存应当放到共享内存中，以避免频繁地读写全局内存。在每次分解重构过程中，其过程可以概括为：数据搬运到 GPU、计算 DWT、将分解系数搬运到 CPU 以进行展示和应用、将分解系数搬运到 Host、计算 IDWT、将恢复图像搬运回 Host。

这里与传统 Mallat 算法不同之处在于，为了提高内存读写效率，没有反复创建新内存用于存储各级系数，而是将结果放入与原始图像相同尺寸的分解图中。这样具有额外的优点，即不需要将小数据合并成大数据提高传输效率。同时，无需对四幅子图分别进行滤波操作，而是每次对两幅子图进行滤波，这样可以减少边缘延拓的时间，提高算法效率。当然这也要求了对于 l 级分解，需要有图像的尺寸可以被 2^l 整除。本文算法过程见图3,GPU 优化部分主要包括行列滤波，上采样和下采样，矩阵求和。下面将对每个步骤展开叙述。



(a) 分解



(b) 重构

图 3: Mallat 小波分解重构基本过程

2.2 行列滤波

行滤波和列滤波中，需要分别将图像的每一行或列卷积滤波器系数。行列卷积在时间尺度上是分开进行的，也即存在前后关系，因此这里分开讨论行列卷积，此处仅以行卷积为例进行讨论。简单的考虑每个 Thread 计算一个像素卷积结果，考虑边缘镜像延拓，则代码可以如下：

```
__global__ void conv_row_kernel(const float *src, float*dst, const float*kernel,
    int m, int n, int kernelSize)
```

```
{
    //Thread ID
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid >= m * n) {
        return;
    }
    //decide accessed pixel according to tid
    int row = tid / n;
    int col = tid % n;
    int visit_row;
    dst[row*n + col] = 0; //Not used as the sum operation in idwt_gpu. So it needs
         cudaMemset to 0 manually.

    //Conv
    for (int i = 0; i < kernelSize; i++) {
        visit_row = row + i - (kernelSize - 1);
        //Use Mirror-padding.

        if (visit_row < 0) {
            visit_row = -visit_row - 1;
        }
        else if (visit_row >= n) {
            visit_row = 2 * n - visit_row - 1;
        }

        if(visit_row>=0 && visit_row < n )
        dst[row*n + col] += kernel[i] * src[visit_row*n + col];
    }
}
```

很容易发现，通常来说小波核函数卷积核（kernel）尺寸远小于图像（src）尺寸，所以实际计算卷积的乘加运算开销相比前面的内存读取十分小，因此该核函数大笔时间浪费在了内存读写上。因此，优化思路是尽可能的降低内存读取时间。

首先，由于一次卷积中的所有的卷积核被重复使用，因此可以放入常量内存（使用 __constant__ 修饰符），以降低访问时间。其次考虑将数据放于共享内存（使用 __shared__ 修饰符），然而共享内存往往很小（例如 RTX 2060 为 48KB），一般仅能存数行图像。鉴于一般图像很少超过 4K 分辨率，而使用浮点型存储一行或一列数据最多需要 $1920 \times 4 \times 4 = 30kB$ 的存储空间。所以可以令每个 Block 存储一行数据。然而这意味着每行处理像素会受到线程数的制约，也就是通常像素行列像素不可以高过 1024，对此可以考虑将图像分成多幅子图，分别进行行列滤波，只需要保证分割后的边长不大于 1024。本文为了简化问题，并未采用这种理想的分割方式，而是使用一个线程分批次处理图像的一行或一列。主要代码如下：

```
__global__ void convrow_kernel(const float *src, float*dst, int filter_id, int m
    , int n, int kernelSize)
{
    extern __shared__ float shared[];

    int row = blockIdx.x;
    int col = threadIdx.x;
    if (row >= m || col >= n) {
        return;
    }
    // select kernel
    const float* kernel = NULL;
    switch (filter_id)
    {
    case FILTER_DL:
        kernel = c_kernel_dl;
        break;
    case FILTER_DH:
        kernel = c_kernel_dh;
        break;
    case FILTER_RL:
        kernel = c_kernel_rl;
        break;
    case FILTER_RH:
        kernel = c_kernel_rh;
        break;
    }

    while (col < n)
    {
        //internal index in shared mem of external coloum
        //0 - padding.
        if (col < kernelSize / 2) {
            shared[col] = 0;
            shared[n + col + kernelSize / 2] = 0;
        }
        shared[col + kernelSize / 2] = src[row*n + col];
        __syncthreads();

        //mult-add
        float sum = 0;

        for (int i = 0; i < kernelSize; i++) {
            int index = col + i;
```

```
            sum += shared[index] * kernel[i];
        }
        dst[row*n + col] = sum;
        col += THREADS_PERBLOCK;
    }
}
```

2.3 升采样和降采样

升采样和降采样过程只是进行了简单的内存搬运。降采样过程中对两个目标内存的访问是相互独立的；升采样对两个源地址内存的访问也是分开的，因此可以同时进行。具体过程分为分别对行列的升采样、降采样四个函数，其中对列降采样的主要代码如下：

```
__global__  void downsampling_cols_kernel(const float *src, float*dst, int m, int
    n, int type) {
    //Thread ID
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid >= n/2) {
        return;
    }
    //each tid represent a column.
    int col = tid;

    //col bias.
    int col_bias = 0;
    if (type == DOWNSAMPLING_RIGHT) {
        col_bias = n / 2;
    }
    for (int i = 0; i < m; i++) {
        int col_src = col * 2;
        int col_dst = col_bias + col;
        dst[i*n + col_dst] = src[i*n + col_src];
    }

}
```

3 效果评估

在本小节中首先对算法的有效性进行了评估，证明算法可以实现小波分解与重构，并且随着图像尺寸增大，小波分解重构算法的性能随之提升；随着分解级数的增大，分解重构算法的性能随之降低。此外，本小节还评价了并行算法加速效果，以及优化后的算法相较于直接并行实现的提升。最后本小节分析了算法的不足，并提出了改进思路。

本文的软硬件测试环境如表1所示，其中 GPU 的具体信息如下所示：

```
Device 0: "NVIDIA GeForce RTX 2060"
    CUDA Driver Version / Runtime Version          11.4 / 10.1
    CUDA Capability Major/Minor version number:    7.5
    Total amount of global memory:                 6144 MBytes (6442450944 bytes)
    (30) Multiprocessors, ( 64) CUDA Cores/MP:     1920 CUDA Cores
    GPU Max Clock rate:                            1335 MHz (1.34 GHz)
    Memory Clock rate:                             7001 Mhz
    Memory Bus Width:                              192-bit
    L2 Cache Size:                                 3145728 bytes
    Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536)
        , 3D=(16384, 16384, 16384)
    Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
    Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
    Total amount of constant memory:               65536 bytes
    Total amount of shared memory per block:       49152 bytes
    Total number of registers available per block: 65536
    Warp size:                                     32
    Maximum number of threads per multiprocessor:  1024
    Maximum number of threads per block:           1024
    Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
    Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
    Maximum memory pitch:                          2147483647 bytes
    Texture alignment:                             512 bytes
    Concurrent copy and kernel execution:          Yes with 6 copy engine(s)
    Run time limit on kernels:                     Yes
    Integrated GPU sharing Host Memory:            No
    Support host page-locked memory mapping:       Yes
    Alignment requirement for Surfaces:            Yes
    Device has ECC support:                        Disabled
    CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver
        Model)
    Device supports Unified Addressing (UVA):      Yes
    Device supports Compute Preemption:            Yes
    Supports Cooperative Kernel Launch:            Yes
    Supports MultiDevice Co-op Kernel Launch:      No
    Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
    Compute Mode:
       < Default (multiple host threads can use ::cudaSetDevice() with device
           simultaneously) >

  deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.4, CUDA Runtime
      Version = 10.1, NumDevs = 1
```

```
Result = PASS
```

表 1: 测试环境

GPU	
GPU Architecture	NVIDIA GeForce RTX 2060
Global memory	6144 MBytes
Cores	30 MP
	64 Cores/MP
	1920 Cores
GPU Max Clock rate	1335MHz
Host	
CPU	Intel(R) Core(TM) i7-9750 CPU @ 2.60GHz
CPU Cores	6
Memory	16GB DDR4
Software	
OS	Windows 10
CUDA	10.1

3.1 算法的有效性评估

对实验结果进行评价首先要保证算法是有效的。对于小波分解与重构算法，最重要的是重构图像要尽量与原始图像一致，比较常见的评价标准为均方误差（MSE）和峰值信噪比（PSNR）。对于 $M \times N$ 大小的图像 I，其重构图像为 R，则定义均方误差 MSE 的定义为:

$$MSE = \frac{1}{MN} \sum_{i}^{M} \sum_{j}^{N} (I_{i,j} - R_{i,j})^2 \tag{6}$$

显然，均方误差越小，重构图像与原始图像越接近，图像的分解重构效果越好。

对于 8bit 图像，定义峰值信噪比 $PSNR$ 为:

$$PSNR = 10log_{10} \frac{255^2}{MSE} \tag{7}$$

$PSNR$ 与 MSE 负相关，MSE 越小则 $PSNR$ 越大，图像的分解重构效果越好。

使用了滤波器长度为 6 的 db3 小波函数，分别使用 CPU 和 GPU，针对不同分辨率的 Lena 图像，分别进行了一级小波分解重构（见图4）以及三级分解重构（见图5）。对于 CPU 和 GPU 两种算法分别计算了残差、MSE 以及 $PSNR$。可以看到，两种算法都基本实现了小波分解与重构算法。纵向来看，两种算法在图像尺寸较小的时候重构效果较差，随着图像尺寸的提升，MSE 随之减小，

PSNR 随之增大，两种算法的重构效果均有所提升。对比不同分级重构级数的效果（图4和图5），随着分解重构级数的增加，重构效果会降低。

这是因为卷积算法策略造成的失真。卷积运算的实际结果并非严格按照数学定义计算的结果，而是在理论计算结果上截取了原始数据长度的计算结果。所以，如果卷积核长度为偶数时，需要舍弃的结果有奇数个，于是不可避免地引入偏移。当图像尺寸增大后，这些边缘占比会减小，对重构图像的影响也会减小。这种失真也会随着随着逐级小波分解不断积累误差，分解级数较高的图像重构效果也相对较差。此外，将多幅图像拼接后进行滤波，会造成图像拼接处边缘的混叠，影响重构结果。

3.2 并行加速效果评估

通常使用串行计算时间与并行计算时间为加速比，用来评价并行算法对串行算法的加速效果。加速比越高意味着并行算法的加速效果越好。分别计算了 CPU 算法和 GPU 对多幅尺寸的图像进行 1 级分解重构的时间，并计算了加速比（见图6(a)）。

可以看到并行算法有效地提升了运算速度，但是随着数据尺寸增大，加速比呈现亚线性增长。随着图像尺寸的增大，加速比会曲折增大。主要趋势为增大是因为随着图像尺寸的增大，并行计算占比整体是是在增大的。每经过一定间隔，加速性能有所下降则与算法中规定了每行使用一个 Block 的多个线程处理有关。算法中，为了降低难度，对于尺寸较大的图像并没有拆分处理，而是使用一个 Block 轮流分段处理。因此，当图像尺寸每增大超过 Block 最大线程数这一参数时，Block 中会重新分配共享内存，造成效率降低。

3.3 优化效果评估

图6(b) 展示了优化后的并行算法相较于简单的并行算法的提升。可以看到，当图像尺寸较小时，使用常量内存、共享内存带来的损耗与节省的时间基本一致，随着图像尺寸的增大，优化效果逐步体现出来。

3.4 不足与改进

本实验基于 CUDA 实现了对图像的多尺度小波分解与重构算法，并对算法进行了并行优化处理，最后验证了算法的有效性并评估了优化效果。并行算法依然存在着不足：

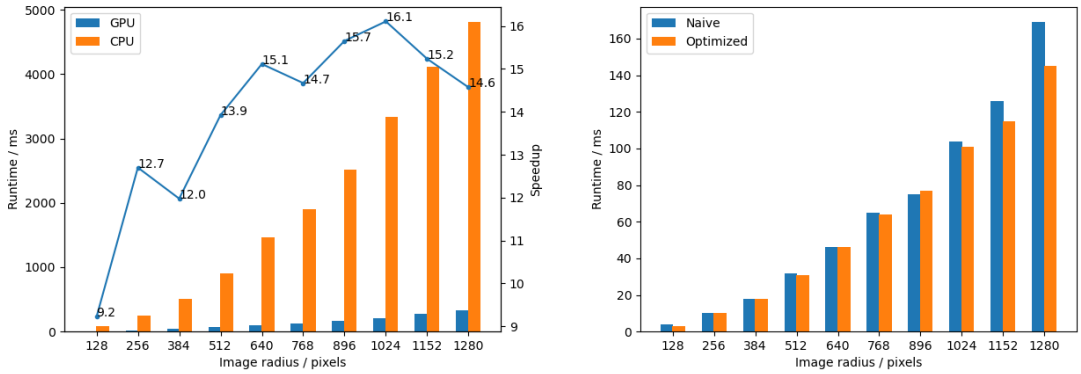
(1) 算法结构设计不合理，在 CUDA 编程中可以合并滤波与采样过程。

(2) 卷积算法针对卷积核长度为偶数情况的优化不够，需要进一步优化卷积计算方法。

(3) 计算大图像时依然有优化空间，可以将大图像划分为多个较小的子图像进行处理，而非使用一组线程轮流处理。

图 4: db3 小波分解与重构结果，$level = 1$：每一行对应一种分辨率图像的分解重构结果，每一行从左到右依次为原始图像、CPU 分解重建图像、GPU 分解重建图像、CPU 分解重建图像对于原始图像的残差、GPU 分解重建图像对于原始图像的残差。

图 5: db3 小波分解与重构结果，$level = 3$



(a) 并行加速效果　　(b) 优化效果

图 6: 并行加速效果与优化效果

参考文献

Franco, Joaquín, Gregorio Bernabé, Juan Fernández, and Manuel E. Acacio, "A Parallel Implementation of the 2D Wavelet Transform Using CUDA," in "2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing" 2009, pp. 111–118.

Gonzalez., Rafael C., 数字图像处理：第三版, 电子工业出版社, 2017.

Mallat, S.G., "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1989, *11* (7), 674–693.

Tenllado, Christian, Javier Setoain, Manuel Prieto, Luis Piñuel, and Francisco Tirado, "Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting," *IEEE Transactions on Parallel and Distributed Systems*, 2008, *19* (3), 299–310.