```
3. globalSize←get_global_size(0)
4. //线程内归约
5. temp←0
6. for i=0 to times
7.    temp←src[idx+i*globalSize]+temp
8. end for
9. lSum[idx_loc]←temp
10.   barrier(CLK_LOCAL_MEM_FENCE)
11.   //然后进行 work-group 内归约和 work-group 间归约
```

2)Local-Stride Kernel

每一个线程以 work-group 内线程数(local stride)为步长,读取相距 local stride 的多个数据(数据量由线程内归约粒度 *times* 控制),然后对这些数据进行归约处理,最后把归约结果写入位于 LDS 中的 lSum 数组,再进行下一层次的 work-group 内归约优化。其伪代码如算法 4 所示。

算法 4 Local-Stride Kernel

```
Input:src(Original data),lSum(local memory)
Output:dest(Length is 1)
1. idx←get_global_id(0)
2. idx_loc←get_local_id(0)
3. globalSize←get_global_size(0)
4. //线程内归约
5. temp←0
6. idx←idx_loc+idx_gro*lSize*times
7. for i=0 to times
8.    if (idx+i*lSize)<data_len
9.       temp←src[idx+i*lSize]+temp
10.   end if
11. end for
12.   lSum[idx_loc]←temp
13.   barrier(CLK_LOCAL_MEM_FENCE)
14.   //然后进行 work-group 内归约和 work-group 间归约
```

两个 kernel 的区别在于每个线程读取相邻数据的步长不同。步长为 global stride 和 local stride 时,两个 kernel 在不同的 GPU 硬件平台上各有优劣,本文将在 4.3.1 节进行实验分析。

3.2.2 work-group 内归约优化

1)Wavefront 优化和局部内存优化

由图 1 可知,Naïve Reduction 执行时 wavefront 内部线程存在条件分支,而且对 LDS 的 bank 利用率低。首先,Naïve Reduction 执行归约的线程 ID 并不连续,意味着同一个 wavefront 的线程在 kernel 执行过程中存在条件分支,一部分线程负责归约操作,另一部分线程则处于空闲状态。其次,从图 1 的第一层归约可以看出,由于存在空转线程,因此部分 bank 同样处于空闲状态,LDS 的利用率低。

图 3 为改进后的归约算法示意图。wavefront 内线程不存在条件分支,一个 wavefront 所能处理的数据将会翻倍,提升了 wavefront 的工作效率,有效地减少了实际工作的 wavefronts 数目,使其约为 Naïve Reduction 的一半。对于局部内存的访问,通过连续的线程访问连续的数据,连续的 32 个线程将会访问连续的 bank,在提升 LDS 利用率的同时,有效地避免了 bank conflict,进一步提升了算法性能。将完成 wavefront 优化和局部内存优化的算法版本定义为 Divergence-Free Kernel,其相对于 Naïve Kernel 取得了良好的性能提升,详情参照 4.3.2 节。

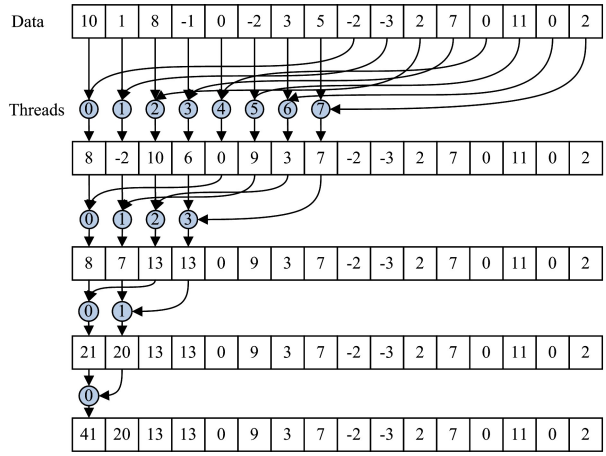


图 3 Divergence-Free Kernel 的归约过程

Fig. 3 Reduction process of Divergence-Free Kernel

2)循环展开

本节针对 work-group 内归约进行循环展开优化。首先从硬件资源组织上分析,每一个 wavefront 由 64 个线程组成(warp 由 32 个线程组成),wavefront 是 GPU 调度与执行的基本单位,wavefront 内所有线程均执行相同的指令。由此可知,在 work-group 内归约中的 for 循环中,当运行线程数小于或等于 64,即运行线程都属于同一个 wavefront 时,可以省去显式的本地同步操作,以提升算法性能。

考虑到本文设定 work-group 内部线程数为 256,可对 for 循环进行完全展开。这里需要注意的是,当 work-group 内实际工作线程的数目大于 64(32,NVIDIA GPU)时,仍需要显式的本地同步。

因此,在 Divergence-Free Kernel 的基础上提出循环展开优化后的 work-group 内归约优化算法版本 Completely-Unroll Kernel,其 work-group 内归约的伪代码如算法 5 所示。

算法 5 Completely-Unroll Kernel

```
Input:src(Original data),lSum(local memory)
Output:dest(Length is 1)
1. //线程内归约
2. 采用算法 2 的线程内归约
3. //work-group 内归约
4. volatile __local uint * ldata=lSum;
5. if idx_loc<128 then
6.    ldata[idx_loc] ←ldata[idx_loc+128]
7. end if
8. barrier(CLK_LOCAL_MEM_FENCE);
9. if idx_loc<64 then
10.   ldata[idx_loc] +=ldata[idx_loc+64]
11.   ldata[idx_loc] +=ldata[idx_loc+32]
12. …
13.   ldata[idx_loc] +=ldata[idx_loc+1]
14.   end if
```

```
15.  //Work-group 间归约
16.  采用算法 2 的 work-group 间归约
```

3.2.3　work-group 间归约优化

归约算法中的 work-group 间归约主要负责完成对每一个 work-group 在第二层中得到的局部归约结果的再归约操作，最终得出原始数据集的归约结果。work-group 间归约共有 3 种方法：1)将所有 work-group 得到的局部归约结果写入位于 global memory 中的临时数组，然后再重新启动归约 kernel，进行递归归约操作，直至得到最终的归约结果。然而，由于启动 kernel 是一个十分耗时的操作，因此不建议使用。2)将局部归约结果临时数据回传至 CPU 内存中，在 CPU 端完成最后的归约操作。但由于数据的回传需要经过 PCI-E 总线，非常耗时，因此这种方法需要适当限制开启的 work-group 数目。3)采用原子操作求得最终的归约结果，这也是本文采用的方法。

本文在归约算法第三层的 work-group 间归约采用原子操作的主要原因有两点：1)本文实现采用了线程内归约优化，可大大减少开启的 work-group 数目，从而减少了需要进行 work-group 间归约的局部归约结果的数量；2)虽然开启的 work-group 数目较多，但在 GPU 目前的调度机制中能够同时进行 work-group 间归约，调用原子操作的 work-group 数量的最大值为硬件 CU 的个数。同时，因为这些 work-group 在最终执行时会存在一定的时间间隔，调用原子操作对性能的影响会进一步较小。所以，相对于前两种方法，使用原子操作来完成 work-group 间归约可大大提升整体性能。

3.2.4　其他优化方法

除此之外，本文还采用了以下通用优化方法。

1)指令选择

由于指令本身的特点和处理器元件数目的差异，不同指令的吞吐量也存在差异，选择具有高吞吐量的计算指令可有效提高 GPU 程序的性能。本文工作的指令选择优化主要体现在：①使用位运算指令代替乘法和除法指令；②使用乘加指令代替乘法和加法指令；③避免取余运算。

2)去除 clFinish()

本文实验设置命令队列中的命令是顺序执行的，由于 kernel 执行后还需调度 clEnqueueReadBuffer()把最终归约结果读回 CPU 端，此处存在隐式的同步操作，因此无需显式地调用 clFinish()函数进行阻塞。阻塞函数十分费时，应尽量避免使用阻塞函数。

3.3　PCL-Reduction

我们称采用以上优化方法的 GPU 归约算法为 PCL-Reduction。PCL-Reduction 采用了基于 stride 的线程内归约优化、work-group 内归约优化和 work-group 间归约等优化方法，算法性能得到了提升。

4　实验性能评估

4.1　实验环境搭建

4.1.1　硬件环境

本文实验选择两个不同厂商的 GPU 硬件测试平台：AMD W8000 和 NVIDIA Tesla K20M。CPU 硬件测试平台为 Intel Xeon E5-2620 v2。相关硬件平台的主要性能参数如表 1 所列。

表 1　主要性能参数

Table 1　Main performance parameters

硬件平台	AMD W8000	NVIDIA Tesla K20M	Intel Xeon E5-2620 v2
主频/MHz	880	706	2100
计算核心/个	1792	2496	6
带宽/(GB/s)	176	208	—
单精度浮点/TFLOPS	3.23	3.52	—

4.1.2　软件环境

本文提出的 GPU 归约算法采用 OpenCL1.2 和 CUDA 7.0 作为开发环境，并选取 OpenCV2.4.11 中的 CPU 版本，CUDA 版本和 OpenCL 版本的归约求和函数(分别简称为 CPU-Reduction，CUDA-Reduction 和 OpenCL-Reduction)作为性能参照对象。其中，在 CPU 版本的编译中，采用了“-O3”编译选项。OpenCV 库中 3 个版本均进行了精心优化，与它们进行对比，本文的工作将更具说服力。

此外，在本文的测试中，实验数据类型为 unsigned int；若无特别说明，实验采用的数据规模均为 2^{24}(用户可根据自身应用需求将数据规模调整为其他任意大小的数据规模)；work-group 内线程数目设定为 256。

4.2　总体性能评估

通过将本文的最终优化版本 PCL-Reduction 在不同的数据规模下分别与 OpenCV 库的 CPU 版本(CPU-Reduction)和 GPU 版本(CUDA-Reduction 和 OpenCL-Reduction)进行性能对比，来验证本文优化工作的高效性；并通过在 AMD W8000 和 NVIDIA Tesla K20M 两个不同厂商 GPU 上的实验，来验证本文工作能实现在不同 GPU 硬件平台间的高性能。

4.2.1　并行归约与串行归约的性能对比

本节在不同数据规模下，以加速比(串行归约时间/并行归约时间)为性能评价标准，将本文的最终优化版本与 OpenCV 库中归约求和算法的 CPU 版本(CPU-Reduction)进行性能对比分析，结果如图 4 所示。

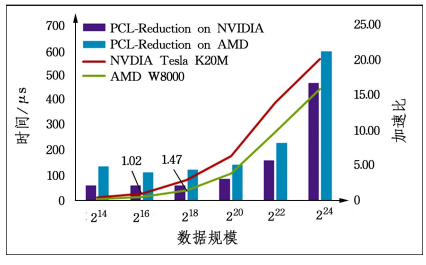


图 4　PCL-Reduction 与 CPU-Reduction 的性能对比

Fig. 4　Performance comparison between PCL-Reduction and CPU-Reduction

根据图 4，当数据规模较小(如小于 2^{14})时，OpenCV 的串行归约算法的速度优于本文的最终优化版本。这是因为当处理数据量过少时，开启的线程数量少，导致 CPU 不能充分隐藏访存延迟，并且不足以充分利用 GPU 丰富的计算资源。

随着数据规模的增长，开启的线程数量逐渐增多，wavefront 通过相互间的切换来隐藏访存开销，使 GPU 保持高效的工作效率，从而使性能得到提升。PCL-Reduction 无论在 NVIDIA GPU 还是 AMD GPU 上，均取得了良好的性能加速。其中，在 NVIDIA Tesla K20M 上的加速比可达 2.97～20.24，在 AMD W8000 上的加速比可达 3.91～15.93。

4.2.2 并行归约算法间的性能对比

本节以归约函数运行时间作为性能评价标准，进一步在两种 GPU 平台上将 PCL-Reduction 分别与 OpenCV 库中的 CUDA-Reduction 和 OpenCL-Reduction 进行性能对比分析，以更合理地对 PCL-Reduction 算法的性能进行评估。

在 NVIDIA Tesla K20M 上，CUDA-Reduction，OpenCL-Reduction 和 PCL-Reduction 三者在 NVIDIA Tesla K20M 硬件平台上的性能如图 5 所示。在不同的数据规模下，PCL-Reduction 的性能均优于 CUDA-Reduction 和 OpenCL-Reduction，相比于 CUDA-Reduction 版本，其性能加速了 2.25～5.97 倍，相比于 OpenCL-Reduction 版本，其性能加速了 1.25～1.75 倍。

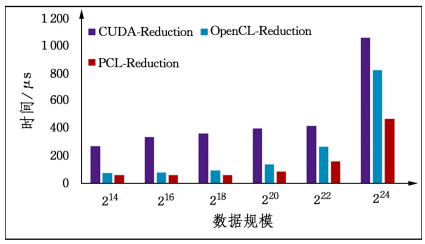


图 5 GPU 归约算法在 NVIDIA Tesla K20M 上的性能对比

Fig. 5 Performance comparison of GPU reduction algorithm on NVIDIA Tesla K20M

在 AMD W8000 上，由于 CUDA-Reduction 函数无法运行于非 CUDA 架构的 GPU 上，因此只对 OpenCL-Reduction 函数和 PCL-Reduction 两者在 AMD W8000 硬件平台上进行了性能测试，结果如图 6 所示。在不同数据规模下，PCL-Reduction 的性能均优于 OpenCL-Reduction 函数，加速比达到 1.24～5.15。

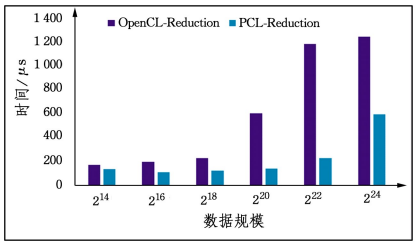


图 6 GPU 归约算法在 AMD W8000 的性能对比

Fig. 6 Performance comparison of GPU reduction algorithm on AMD W8000

依据图 5 及图 6 可知，通过在不同 GPU 硬件平台与 OpenCV 库的并行归约函数进行对比，PCL-Reduction 算法均实现了较好的加速，具有更优的性能。GPU 编程的难点不仅在于算法实现，更在于算法性能的可移植性。本小节在反映 PCL-Reduction 更充分地挖掘和利用了 GPU 潜在能力的同时，也验证了其代码可移植性和性能可移植性。

4.3 性能评估

4.3.1 线程内归约优化

根据 3.2.1 节关于步骤 1 线程内归约过程的优化分析，本节将对步长不同的两个线程内归约优化算法版本即 Global-Stride Kernel 与 Local-Stride Kernel 进行性能比较，以寻找两者的性能差异。值得注意的是，Harris 提到以局部线程数为跨步进行取数归约，未曾考虑以全局线程数为跨步进行取数归约。在 2^{20} 的数据规模下，通过在 AMD W8000 和 NVIDIA Tesla K20M 上进行实验，来展示两者的性能特征，实验结果如图 7 所示。

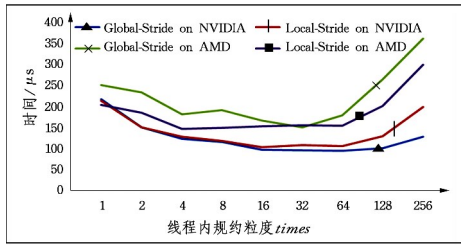


图 7 线程内归约算法在不同 GPU 平台上的性能对比

Fig. 7 Performance comparison of inner-threads reduction algorithm on different GPU platforms

需要注意的是，线程内归约粒度 *times* 为 1 时，两个 kernel 均将退化为 Naïve Reduction 的线程内归约过程，即只简单地从 global memory 拷贝数据至 LDS，而不进行任何归约操作。

由图 7 可知，在 NVIDIA Tesla K20M 平台上，Global-Stride Kernel 在多数情况下的性能略优于 Local-Stride Kernel；而在 AMD W8000 平台上，Local-Stride Kernel 在多数情况下的性能优于 Global-Stride Kernel。因此，在不同的 GPU 架构下，采用不同的跨步对归约算法性能的影响是有差异的。然而更值得引起重视的是，两种算法在不同平台上的性能曲线的变化趋势是相同的，随着线程内归约粒度 *times* 的逐渐增大，算法的运行时间呈现先下降后上升的趋势。以图 7 中运行于 AMD W8000 平台上的 Global-Stride Kernel 为例，当粒度 *times* 为 64 时出现拐点，粒度小于拐点粒度时，算法运行时间随着粒度的增加而减少；当粒度大于拐点粒度后，算法运行时间随着粒度的增大而增加。

性能提升的主要原因有 3 点：1）当数据规模足够大时，即使每个线程处理多个数据，开启的线程数量也足以隐藏访存延迟。2）并行归约算法在执行过程中数据量会逐渐减少，对应的空闲线程也会逐渐增多。而在进行线程内归约时，随着线程内归约粒度的增大，所需开启的线程数目逐渐减少，而每一个线程的工作量随之增加，在减少了空闲线程数量的同时，增加了每个线程的工作量，在一定程度上减缓了空闲线程“到来”的时间。3）随着线程内归约粒度的增大，每个 work-group 所负责归约的数据量越来越大，从而使得 work-group 的数目越来越少，减少了局部归约结果的数量。这样在一定程度上减少了 3.2.2 节步骤 2 中的 work-group 内归约过程（涉及同步操作）和 3.2.3 节步骤 3 中的 work-group 间归约的原子操作，进而使性能得到提升。

然而，随着线程内归约粒度 *times* 的持续增大，work-

group 开启的数目越来越少。当 work-group 数目减少至无法充分隐藏访存延迟或者无法充分利用 GPU 丰富的计算资源时，算法性能将会降低，运行时间将会增加。因此，粒度 *times* 不能无限量地增大，理应综合考虑到 work-group 数目，以充分利用资源。因此，起初线程内归约算法的性能会随着粒度的增大而提升，当粒度增大至一定程度从而导致 work-group 数目过少时，性能变化将会出现拐点，此后算法性能将随着粒度的增加而下降。在此，建议 work-group 的数目应至少为 CU 数目的 4～8 倍。

4.3.2　work-group 内归约优化

根据 3.2.2 节，对 work-group 内归约过程进行优化分析。需要注意的是，work-group 内归约优化阶段的所有优化版本均不包含线程内归约优化。本节实验以 Naïve Reduction 基础版本为参照，给出 work-group 内归约优化版本 Divergence-Free Kernel 和 Completely-Unroll Kernel 在 AMD W8000 和 NVIDIA Tesla K20M 上的性能对比详情，如图 8 所示。

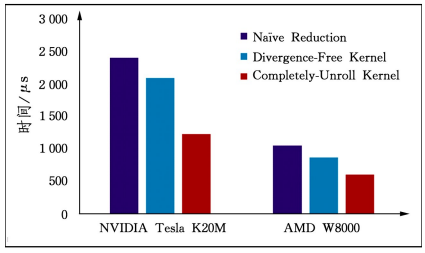


图 8　work-group 内归约算法在不同 GPU 平台上的性能对比

Fig. 8　Performance comparison of inner-work-group reduction algorithm on different GPU platforms

根据图 8 的性能数据，与 Naïve Reduction 相比，work-group 内归约优化算法通过解决 LDS 的 bank conflict 和 wavefront 内线程的条件分支，有效地提升了 LDS 的访问效率及 wavefront 内部线程操作的并行性，从而提升了 kernel 的性能。循环展开优化是在 work-group 内归约优化版本的基础上结合硬件调度和执行单元的特点进行优化的算法版本。循环展开优化对 work-group 内归约过程进行循环展开，一方面可减少归约过程的本地同步操作，进而减少时间开销；另一方面消除了循环体内的冗余操作。因此，kernel 的性能可得到有效提升。最终，经过优化的 work-group 内归约版本相对于 Naïve 版本，在 NVIDIA Tesla K20M 平台上的性能提升了 48.9%，而在 AMD W8000 平台上的性能提升了42.2%。

因此，在不考虑线程内归约优化的情况下，通过对 work-group 内归约过程的优化，能够使归约算法的整体性能得到明显的提升。

4.4　归约算法的瓶颈分析

虽然 work-group 内的归约优化可有效地提升 kernel 的性能，但是其并不是归约算法在 GPU 计算平台上实现和优化的瓶颈，本文指出，线程内归约才是归约算法的性能瓶颈。随着线程内归约粒度的不断增大，work-group 内归约优化对整体性能提升的贡献越来越小。本节将以 PCL-Reduction 为参考，对归约算法中线程内归约（以 Global-Stride Kernel 为例）的重要性进行分析，进而衡量线程内归约优化在归约算法性能提升部分所占的比重。需要特别注意的是，PCL-Reduction 与 Global-Stride Kernel 的唯一区别在于是否包含 work-group 内归约优化，PCL-Reduction 包含 work-group 内归约优化，而 Global-Stride Kernel 仅仅进行了线程内归约。

下面将通过逐步增大线程内归约的粒度来分析 PCL-Reduction 与 Global-Stride Kernel 的性能差距变化，详情如图 9 和图 10 所示。

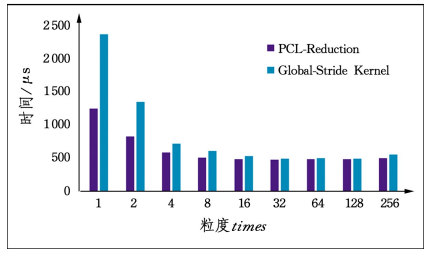


图 9　Global-Stride Kernel 与 PCL-Reduction 在 NVIDIA Tesla K20M 平台上的性能对比

Fig. 9　Performance comparison between Global-Stride Kernel and PCL-Reduction on NVIDIA Tesla K20M

图 9 为 Global-Stride Kernel 与 PCL-Reduction 在不同线程内归约粒度下在 NVIDIA Tesla K20M 硬件平台上的性能对比图。可以看出，当线程内归约粒度较小如 *times*＝1 时，PCL-Reduction 的运行时间明显少于 Global-Stride Kernel 的运行时间，这说明在粒度较小的情况下，work-group 内归约优化的效果明显。然而，随着线程内归约粒度的逐渐增大，Global-Stride Kernel 与 PCL-Reduction 的性能差距逐渐减小，如当线程内归约粒度 *times*＝32 时，Global-Stride Kernel 所达到的性能与 PCL-Reduction 的性能相差不超过 1%。

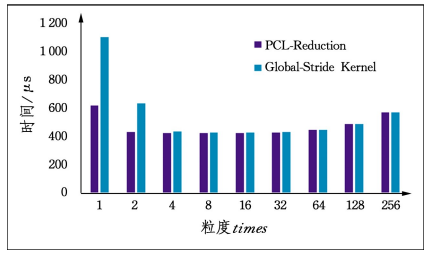


图 10　Global-Stride Kernel 与 PCL-Reduction 在 AMD W8000 平台上的性能对比

Fig. 10　Performance comparison between Global-Stride Kernel and PCL-Reduction on AMD W8000

图 10 为 Global-Stride Kernel 与 PCL-Reduction 在不同线程内归约粒度下在 AMD W8000 硬件平台上的性能对比图。可以看出，图 9 与图 10 具有相同的规律，随着粒度的逐渐增大，Global-Stride Kernel 的性能逐渐逼近 PCL-Reduction 的性能，如当粒度 *times*＝16 时，两者的性能相差不到 1%。

由图 9 和图 10 可知，当逐步增大线程内归约粒度至合适值时，线程内归约优化所取得的性能提升与 PCL-Reduction 几乎等同，即 work-group 内归约优化对归约算法的整体性能

影响不到 1%，因此线程内归约过程为归约算法的瓶颈所在。

结束语 本文以归约求和函数为例，详细叙述了归约算法在 GPU 计算平台上的并行设计与优化，并通过论证指出：线程内归约为归约算法的瓶颈所在，是归约算法优化的关键因素。本文提出的 PCL-Reduction 通过线程内归约、work-group 内归约和 work-group 间归约 3 个层次的优化，最终获得了非常好的性能提升。实验结果表明：在不同数据规模下，PCL-Reduction 与经过精心优化的 OpenCV 库的 CPU 版本相比，在 AMD W8000 和 NVIDIA Tesla K20M 平台上，分别达到了 3.91～15.93 和 2.97～20.24 的加速比；在 NVIDIA Tesla K20M 平台上，相比于 OpenCV 库的 CUDA 版本与 OpenCL 版本，分别达到了 2.25～5.97 和 1.25～1.75 的加速比；在 AMD W8000 平台上，相比于 OpenCL 版本达到了 1.24～5.15 的加速比。

在实验过程中发现一个现象：不论在 NVIDIA Tesla K20M 还是 AMD W8000 上，OpenCL 程序 kernel 被重复调用多次，其第一次的运行时间开销大于之后的运行时间开销，可能的原因在于 OpenCL 程序第一次调用 kernel 时隐含了部分初始化工作。

未来我们希望能将本文工作应用于其他 OpenCL 平台，如在 CPU，APU，Cell 等处理器上进行测试，进一步验证归约算法的瓶颈所在和本文算法的性能。

参 考 文 献

[1] HARRIS M. Optimizing parallel reduction in CUDA[OL]. http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[2] CATANZARO B. Opencl optimization case study: Simple reductions[OL]. https://developer.amd.com/article/opencl-optimization-case-study-simple-reductions.

[3] YAN S G, YUNQUAN Z, GUOPING L. Reduction algorithm optimization based on the OpenCL[J]. Journal of Software, 2011, 22(S2): 163-171.

[4] OpenCV Library Open Source Project [OL]. http://opencv.org.

[5] BRADSKI G, KAEHLER A. Learning OpenCV: Computer vision with the OpenCV library[M]. O'Reilly Media, Inc., 2008.

[6] RAWAT P S, RASTELLO F, SUKUMARAN-RAJAM A, et al. Register optimizations for stencils on GPUs[C]// Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18). New York, NY, USA, 2018.

[7] CECKA C. Low communication FMM-accelerated FFT on GPUs [C]// Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17). ACM, New York, NY, USA, 2017.

[8] ZHANG X X, TAN G M, XUE S B, et al. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning[C]// Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'17). New York, NY, USA, 2017.

[9] LI B, SUN J, ANNAVARAM M, et al. Elastic-Cache: GPU Cache Architecture for Efficient Fine- and Coarse-Grained Cache-Line Management[C]// Proceedings of 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). Orlando, FL, 2017.

[10] KIRK D B, WEN-MEI W H. Programming massively parallel processors: a hands-on approach[M]. Newnes, 2012.

[11] WU E H, LIU Y Q. The General Computing based on GPU[J]. Proceeding of Journal of Computer-Aided Design and Computer Graphics, 2004, 16(5): 601-612.

[12] OWENS J D, HOUSTON M, LUEBKE D. et al. GPU Computing[J]. Proceedings of the IEEE, 2008, 96(5): 879-899.

[13] OWENS J D, LUEBKE D, GOVINDARAJU N, et al. A survey of general-purpose computation on graphics hardware[J]. Computer Graphics Forum: Journal of the European Association for Computer Graphics, 2007, 26(1): 80-113.

[14] Compute unified device architecture programming guide[OL]. http://www.nvidia.com/object/cuda_home.html.

[15] Nvidia cuda C. programming guide[OL]. https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[16] NVIDIA cuda C. Best practices guide [OL]. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide.

[17] The opencl 1.2 speciffication[M]. Khronos OpenCL Working Group, 2012.

[18] STONE J E, GOHARA D, SHI G. OpenCL: A parallel programming standard for heterogeneous computing systems[J]. Computing in Science & Engineering, 2010, 12(1-3): 66-73.

[19] GASTER B, HOWES L, KAELI D R, et al. Heterogeneous Computing with OpenCL: Revised OpenCL 1[M]. Newnes, 2012.

[20] ZOU H, WANG H Q, HUANG Y. GPU Accelerated Rainbow Tables Analysis of MD5 Hash Password[J]. Journal of Chongqing University of Technology (Natural Science), 2013, 27(7): 61-66. (in Chinese)
邹航，王华秋，黄勇. 基于 GPU 加速的彩虹表分析 MD5 哈希密码[J]. 重庆理工大学学报(自然科学)，2013，27(7)：61-66.