OpenCL Programming for the CUDA Architecture

Version 4.0

2/14/2011

In general, there are multiple ways of implementing a given algorithm in OpenCL and these multiple implementations can have vastly different performance characteristics for a given compute device architecture. This whitepaper is a summary of the main guidelines for choosing the best implementation on NVIDIA GPUs. More details are provided in the NVIDIA OpenCL Programming Guide [1] and NVIDIA OpenCL Best Practices Guide [2].

Heterogeneous Computing

An OpenCL application executes across a collection of heterogeneous processors: typically, a host CPU and one or more GPUs.

Applications should take advantage of such a configuration by making sure that all processors are busy most of the time and by assigning to each processor the type of work it does best. The next section goes over the key differences between the CPU and GPU architectures and what type of workload is best suited to each.

Data must be transferred between processors (in the case of discrete GPUs) and for some applications these data transfers can represent a significant fraction of the overall execution time. Applications must therefore strive to either minimize them or hide them by overlapping them with kernel execution. Data transfers can overlap with kernel execution only for data allocated with the **`CL_MEM_ALLOC_HOST_PTR`** flag as detailed in the Programming and Best Practices Guides. Data allocated this way also usually transfer at a higher throughput across the PCIe bus. Hence, it is recommended that OpenCL applications allocate data with this flag.

GPU Computing

In order to distribute a workload appropriately between the CPU and the GPU, it is important to understand the differences between the two architectures.

Highly Multithreaded

One key difference is in how the two architectures address the issue of accessing off-chip memory, a very expensive operation with hundreds clock cycles of latency. CPUs devote a lot of transitors to on-chip caches in order to reduce as much as possible the overall latency caused by off-chip memory accesses. For applications with no or very little parallelism, this *latency reduction* is the best strategy. For applications with a high number of parallel computations, another strategy is to ensure that the processor is always busy with some computations while other computations are waiting on memory accesses or at synchronization points. In other words, latency is “hidden” rather than reduced. This *latency* *hiding* strategy adopted by GPUs is schematized in Figure 1. Latency hiding requires the ability to quickly switch from one computation to another. A GPU multiprocessor (i.e. a compute unit in OpenCL terminology) is therefore designed to support hundreds of active threads at once and unlike CPUs, the cost of switching from one thread to another is insignificant.

Hence, OpenCL applications must launch kernels with a large number of work-items or they will suffer severe performance penalties.

High Arithmetic Throughput

Because off-chip memory latency is assumed to be hidden by running many work-items, GPUs devote many more transistors to arithmetic units than to memory cache, thereby achieving much higher arithmetic throughput than CPUs.

SIMT Execution Model

To increase arithmetic density even further, each GPU multiprocessor has a single instruction unit for multiple arithmetic units. The threads running on a multiprocessor are partitioned into groups in which all threads execute the same instruction simultaneously. On the CUDA architecture, these groups are called warps, each warp has 32 threads, and this execution model is referred to as SIMT (Single Instruction Multiple Threads) (see [3] for more details on the SIMT architecture and how it differs from SIMD vector organizations).

The SIMT execution model is not exposed in OpenCL, but for some kernels, it must be taken into consideration to achieve best performance.

High Memory Bandwidth

To achieve high arithmetic throughput, applications need high memory throughput as well. For this reason, GPUs offer higher memory bandwidth than CPUs – typically an order of magnitude higher – and several specialized memory types that OpenCL applications may leverage in order to maximize memory throughput. Maximizing memory throughput is even more critical going forward, given that memory bandwidth will increase at a slower rate than arithmetic throughput in future processor architectures.

On the CUDA architecture, memory accesses are performed per warp. If addressing by the threads of a warp meets specific requirements, the accesses for an entire warp may result in only a few memory transactions. This *coalescing* is crucial for performance, so it is usually the first optimization to consider.

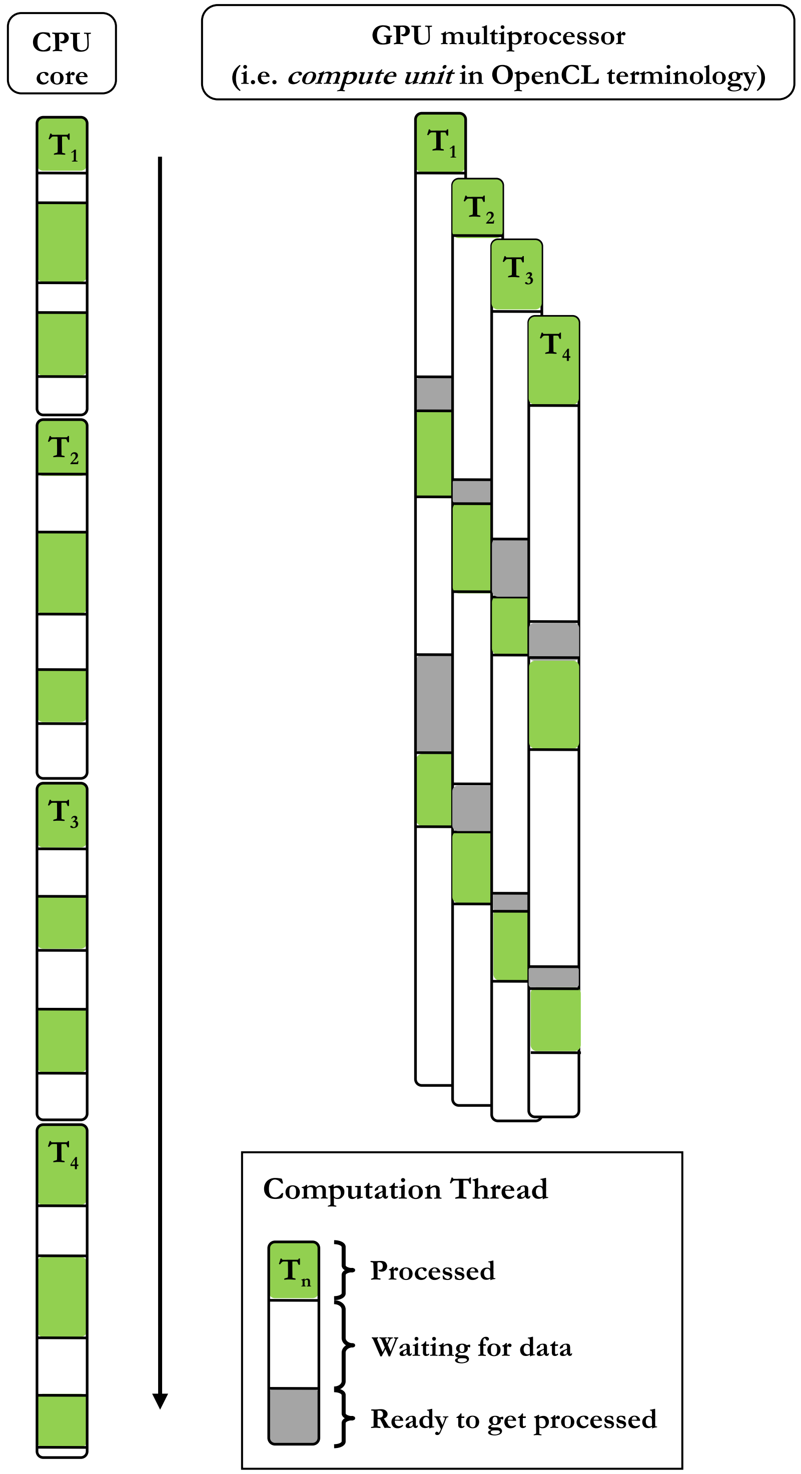


Figure 1. For workload with a lot of parallel computations, GPUs achieve better performance than CPUs by hiding latency with data processing instead of reducing latency with cache.

Data-Parallel Programming

Data parallelism is a common type of parallelism in which concurrency is expressed by applying instructions from a single program to many data elements. It typically generates highly parallel workloads. Not surprisingly, GPUs excel at data-parallel computation; hence a data parallel programming model drives the design of OpenCL.

Current GPUs can only execute one kernel at a time, so they do not support task parallelism as defined in Section 3.4.2 of the OpenCL specification. GPUs do support task parallelism *within* a kernel however, since nothing prevents work-items within the same NDRange (see Section 3.2 of OpenCL specification) to follow different execution paths.

To get maximum benefit from OpenCL, each of the most time-consuming parts of an application should therefore be expressed in a data-parallel way. In other words, each of them should be implemented as an OpenCL kernel that maps each work-item to a portion of the input data as opposed to a specific task. For example, the regular C function of Listing 1 that computes the product **`W`** of a **`width x height`** matrix **`M`** by a vector **`V`** can be implemented in OpenCL by the kernel of Listing 2, invoked using **`clEnqueueNDRangeKernel()`** (see Section 5.6 of OpenCL specification) on a one-dimensional NDRange of **`height`** work-items.

```
void MatrixVectorMul(const float* M,
                     uint width, uint height,
                     const float* V,
                     float* W)
{
   for (uint y = 0; y < height; ++y) {
      const float* row = M + y * width;
      float dotProduct = 0;
      for (uint x = 0; x < width; ++x)
         dotProduct += row[x] * V[x];
      W[y] = dotProduct;
   }
}
```

Listing 1. A function that computes the product **`W`** of a **`width x height`** matrix **`M`** by a vector **`V`**.

```
__kernel void MatrixVectorMul(const __global float* M,
                              uint width, uint height,
                              const __global float* V,
                              __global float* W)
{
   // Each work-item computes one element of W
   uint y = get_global_id(0);
   const __global float* row = M + y * width;
   float dotProduct = 0;
   for (uint x = 0; x < width; ++x)
      dotProduct += row[x] * V[x];
   W[y] = dotProduct;
}
```

Listing 2. OpenCL implementation of the function of Listing 1.

In the kernel of Listing 2, each work-item computes one element of **`W`**. In some cases, giving more work to each work-item and lowering the number of work-items accordingly yields better performance as it amortizes startup overhead better. For this example, it means adding a loop to have each work-item compute multiple elements of **`W`** as in Listing 3.

```
__kernel void MatrixVectorMul(const __global float* M,
                              uint width, uint height,
                              const __global float* V,
                              __global float* W)
{
   // Each work-item computes multiple elements of W
   for (uint y = get_global_id(0); y < height; y += get_global_size(0)) {
      const __global float* row = M + y * width;
      float dotProduct = 0;
      for (uint x = 0; x < width; ++x)
         dotProduct += row[x] * V[x];
      W[y] = dotProduct;
   }
}
```

Listing 3. Alternative implementation of the kernel of Listing 2.

The number of elements computed by each work-item is then equal to **`height`** divided by the total number of work-items (plus one more for some work-items if **`height`** is not a multiple of the number of work-items).

An advantage of this last formulation is that it allows us to decouple the NDRange from the input data size. In other words, the number of work-items can now be arbitrarily set, usually to maximize performance.

NDRange Optimization

The GPU is made up of multiple multiprocessors. For maximum utilization of the GPU, a kernel must therefore be executed over a number of work-items that is at least equal to the number of multiprocessors. However one work-item per multiprocessor is insufficient for latency hiding. In general, hundreds of work-items should be launched per multiprocessor, and the number of work-items should be a multiple of the warp size (i.e. 32), as detailed in the Programming and Best Practices Guides.

Choosing the best NDRange depends on the kernel as well (in particular, the number of registers it uses) and ultimately requires some experimentation. For this reason, applications should not rely on the OpenCL implementation to determine the right work-group size (by setting *local_work_size* to NULL in **`clEnqueueNDRangeKernel()`**).

Memory Optimizations

Assuming that global memory latency is hidden by running enough work-items per multiprocessor, the next optimization to focus on is maximizing the kernel's overall memory throughput. This is done by maximizing the use of high bandwidth memory (OpenCL local and constant memory, Section 3.3 of OpenCL specification) and by using the proper memory access pattern for each memory type to achieve maximum throughput.

Global Memory Coalescing

Ensuring that global memory accesses are coalesced as often as possible is one of the most important optimizations for the CUDA architecture, because it can affect performance by up to an order of magnitude. The Programming and Best Practices Guides give a detailed description of the criteria required for a global memory access to be coalesced and the performance costs if one or several of these criteria are not met. It suffices to say here that the more local and regular the memory accesses by a warp are, the fewer memory transactions result; ideally they should be fully coalesced, i.e. translate into a single memory transaction.

In the kernel of Listing 3, each warp reads **`M`** in a non-coalesced way since the elements read by a warp are not contiguous in memory as illustrated in Figure 2.

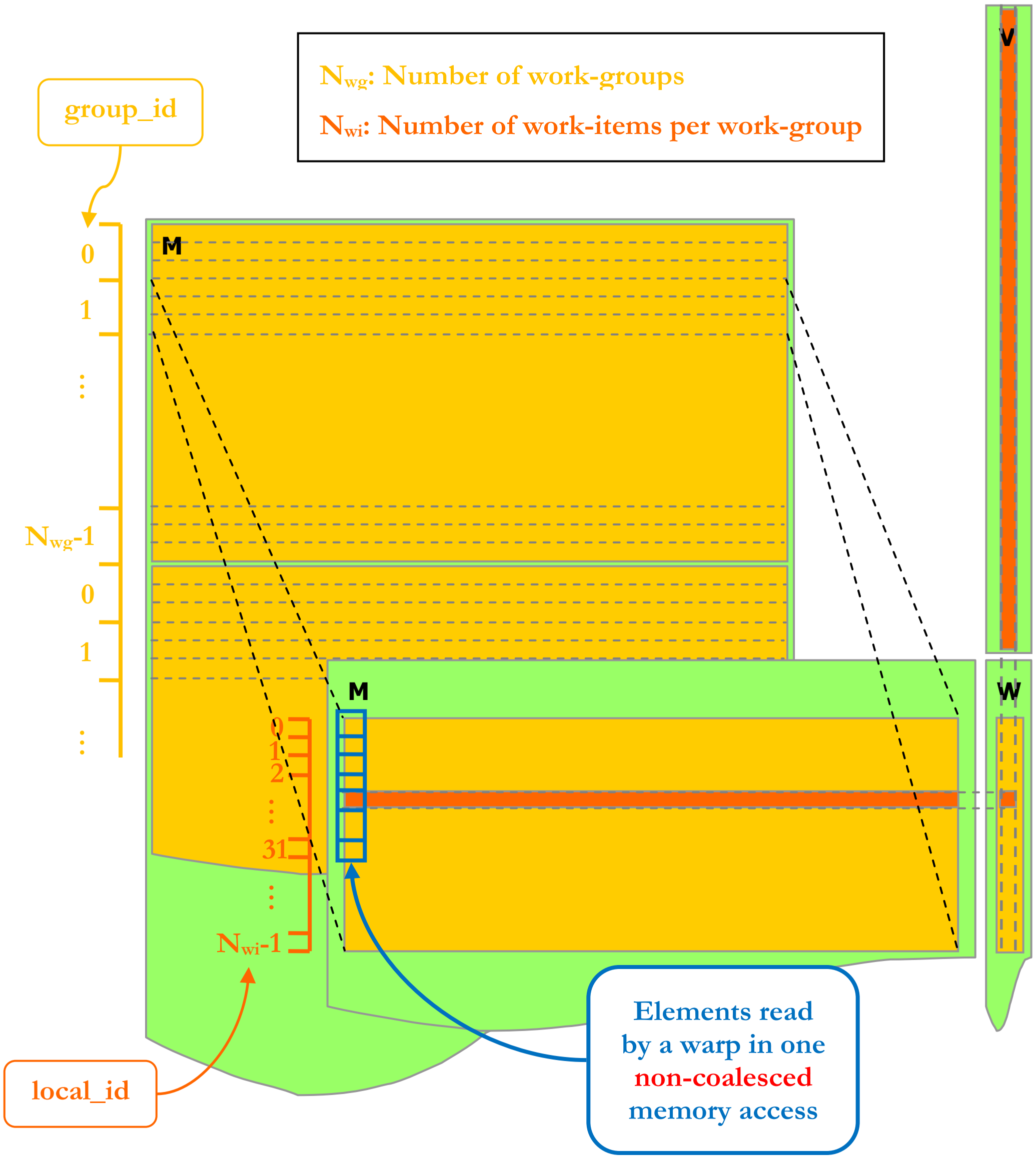

N_{wg}: Number of work-groups
N_{wi}: Number of work-items per work-group
group_id
0
1
⋮
N_{wg}-1
0
1
⋮
M
M
V
W
0
1
2
⋮
31
⋮
N_{wi}-1
local_id
Elements read by a warp in one non-coalesced memory access

Figure 2. A work-item computes multiple elements of W (one every N_{wg} x N_{wi} elements). Global memory accesses are non-coalesced.

To fix this, the kernel must be rewritten to have each work-group, as opposed to each work-item, compute elements of **W** as shown in Listing 4.

```
__kernel void MatrixVectorMul(const __global float* M,
                              uint width, uint height,
                              const __global float* V,
                              __global float* W,
                              __local float* partialDotProduct)
{
   // Each work-group computes multiple elements of W
   for (uint y = get_group_id(0); y < height; y += get_num_groups(0)) {
      const __global float* row = M + y * width;

      // Each work-item accumulates as many products as necessary
      // into local variable "sum"
      float sum = 0;
      for (uint x = get_local_id(0); x < width; x += get_local_size(0))
         sum += row[x] * V[x];

      // Each partial dot product is stored in shared memory
      partialDotProduct[get_local_id(0)] = sum;

      // Synchronize to make sure each work-item is done updating
      // shared memory; this is necessary because in the next step,
      // the first work-item in the work-group needs to read from
      // shared memory the partial dot products written by the other
      // work-items
      barrier(CLK_LOCAL_MEM_FENCE);

      // The first work-item in the work-group adds all partial
      // dot products together and writes the result to global memory
      if (get_local_id(0) == 0) {
         float dotProduct = 0;
         for (uint t = 0; t < get_local_size(0); ++t)
            dotProduct += partialDotProduct[t];
         W[y] = dotProduct;
      }

      // Synchronize to make sure the first work-item is done reading
      // partialDotProduct
      barrier(CLK_LOCAL_MEM_FENCE);
   }
}
```

Listing 4. Alternative implementation of the kernel of Listing 3.

Each work-item is now responsible for calculating part of the dot product of **V** and a row of **M** and storing it to OpenCL local memory. The first work-item in the work-group does the final sum to compute the dot product. A work-group barrier function call is necessary to prevent the first work-item from reading OpenCL local memory before the other work-items are done writing to it.

As illustrated in Figure 3, each warp now reads **M** in a coalesced way and performance is significantly improved as shown in Table 1 at the bottom of the paper.

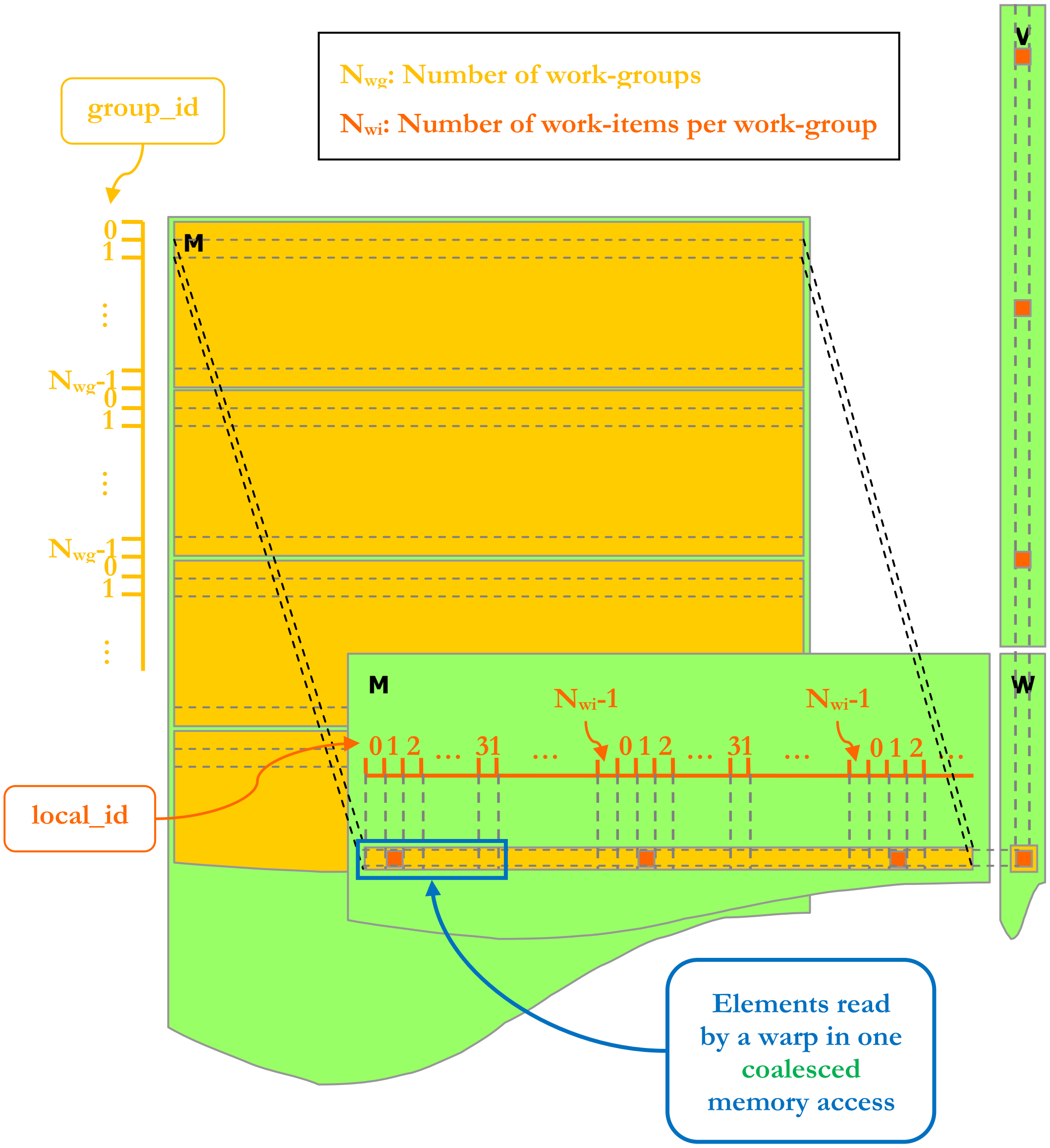

N_{wg}: Number of work-groups
N_{wi}: Number of work-items per work-group
group_id
0
1
…
N_{wg}-1
0
1
…
N_{wg}-1
0
1
…
V
M
M
W
N_{wi}-1
N_{wi}-1
0 1 2 … 31 … 0 1 2 … 31 … 0 1 2 …
local_id
Elements read by a warp in one coalesced memory access

Figure 3. A work-group computes multiple elements of W (one every N_{wg} elements). Global memory accesses are coalesced.

Using OpenCL Local Memory

OpenCL local memory corresponds to on-chip memory in the CUDA architecture and is therefore much faster than global memory.

The Programming and Best Practices Guides give examples of how to use OpenCL local memory to enable coalesced accesses to global memory (as in the kernel of Listing 4) or to reduce low-bandwidth memory accesses.

OpenCL local memory is also used to enable cooperation among the work-items of a work-group. In the kernel of Listing 4, for example, the last part where the first work-item of a work-group serially accumulates the partial dot products can be advantageously replaced with a parallel reduction.

Parallel reduction – reducing an array of values to a single value in parallel (sum of all values, maximum of all values, etc.) – is a fundamental parallel algorithm. It decreases time complexity without performing more operations than a sequential reduction.

There are several possible implementations of parallel reduction. The implementation illustrated in Figure 4 and used in the kernel of Listing 5 generates shared memory bank conflicts as illustrated in Figure 5 (see Programming and Best Practices Guides for a detailed description of shared memory bank conflicts). It nonetheless significantly improves performance over the implementation of Listing 4 as shown in Table 1.

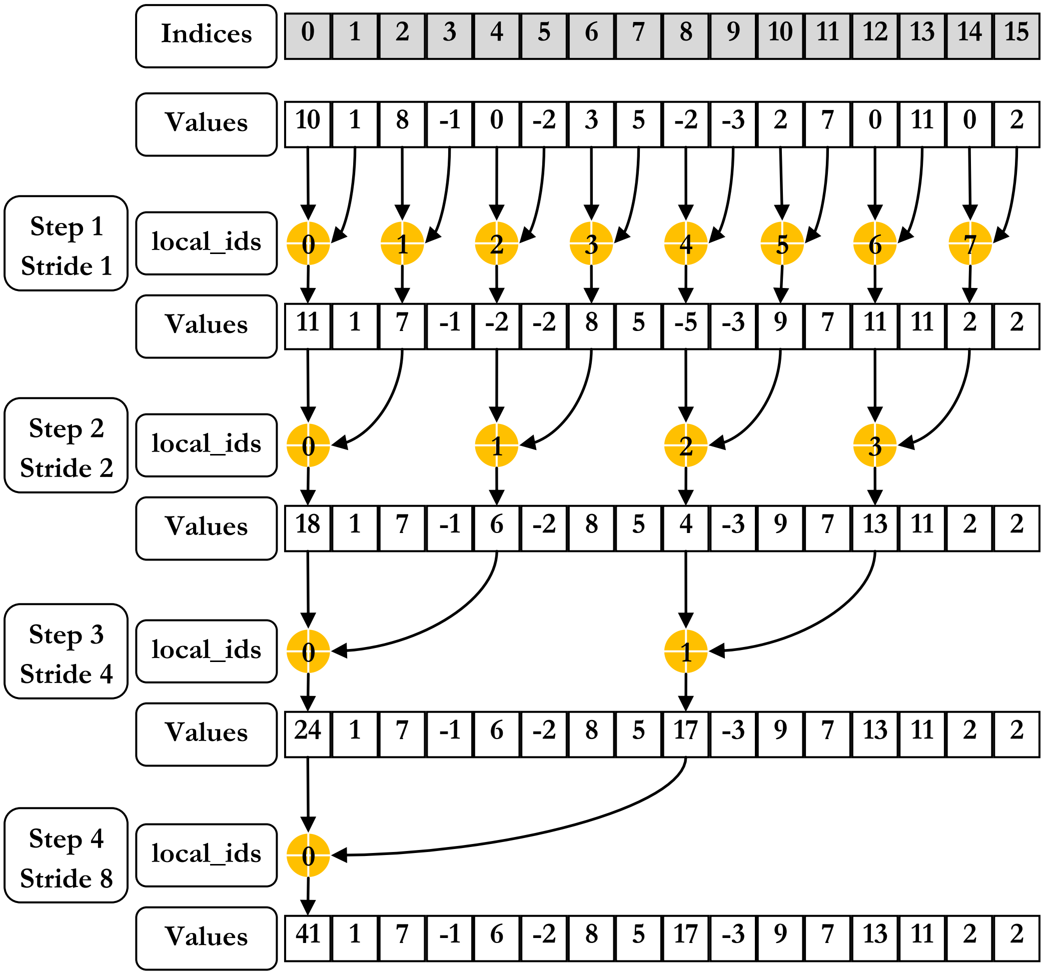
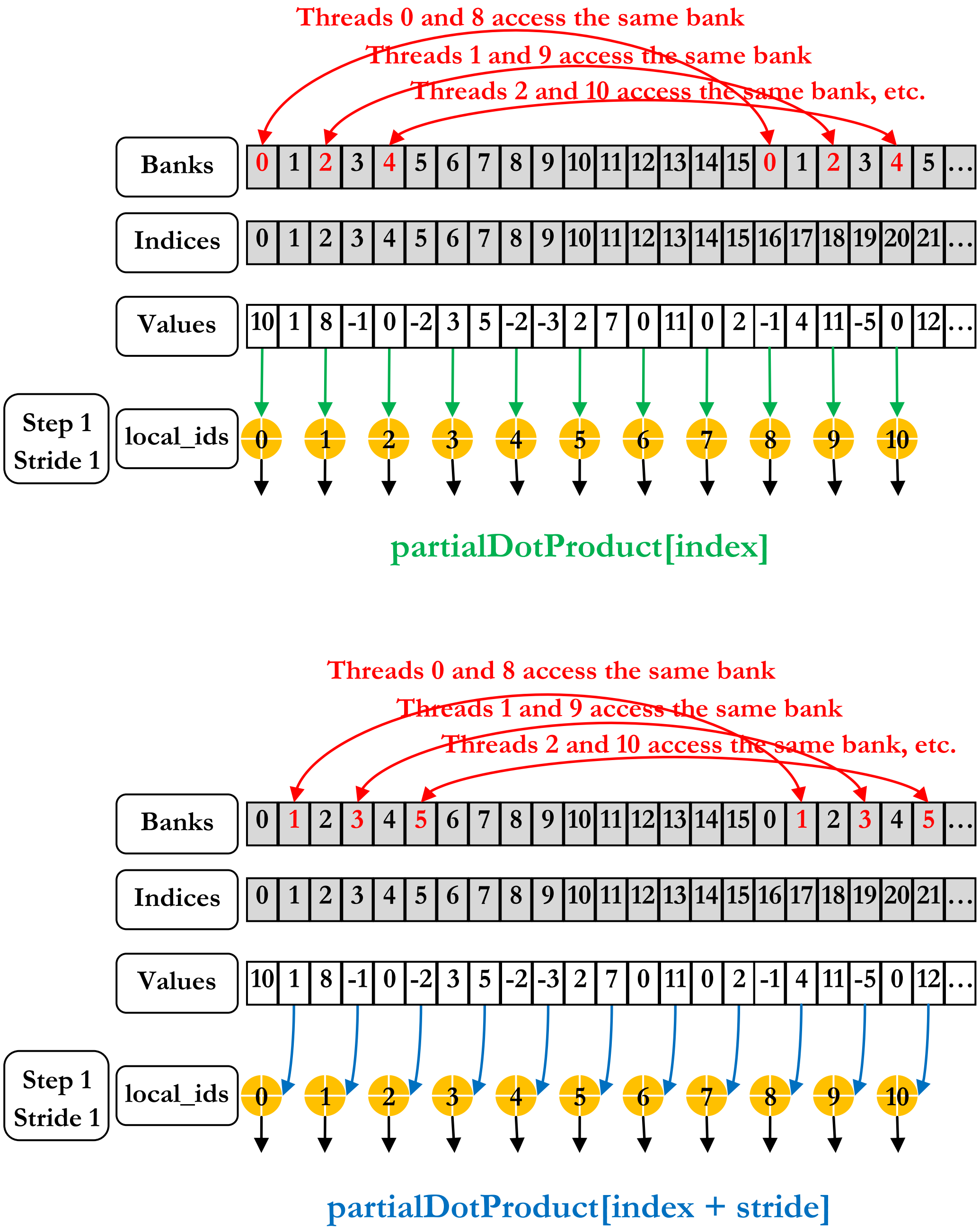


Figure 4. Parallel reduction with shared memory bank conflicts as illustrated in Figure 5.

```
__kernel void MatrixVectorMul(const __global float* M,
                              uint width, uint height,
                              const __global float* V,
                              __global float* W,
                              __local float* partialDotProduct)
{
   // Each work-group computes multiple elements of W
   for (uint y = get_group_id(0); y < height; y += get_num_groups(0)) {
      const __global float* row = M + y * width;

      // Each work-item accumulates as many products as necessary
      // into local variable "sum"
      float sum = 0;
      for (uint x = get_local_id(0); x < width; x += get_local_size(0))
         sum += row[x] * V[x];

      // Each partial dot product is stored in shared memory
      partialDotProduct[get_local_id(0)] = sum;

      // Perform parallel reduction to add each work-item's
      // partial dot product together
      for (uint stride = 1; stride < get_local_size(0); stride *= 2) {

         // Synchronize to make sure each work-item is done updating
         // shared memory; this is necessary because work-items read
         // results written by other work-items during each parallel
         // reduction step
         barrier(CLK_LOCAL_MEM_FENCE);

         // Index into the "partialDotProduct" array where
         // the work-item will write during this step
         uint index = 2 * stride * get_local_id(0);

         // Check for valid indices
         if (index < get_local_size(0)) {

            // Add two elements from the "partialDotProduct" array
            // and store the result in partialDotProduct[index]
            partialDotProduct[index] +=
                                      partialDotProduct[index + stride];
         }
      }

      // Write the result of the reduction to global memory
      if (get_local_id(0) == 0)
         W[y] = partialDotProduct[0];

      // Synchronize to make sure the first work-item is done reading
      // partialDotProduct
      barrier(CLK_LOCAL_MEM_FENCE);
   }
}
```

Listing 5. Alternative implementation of the kernel of Listing 4, using parallel reduction.


Threads 0 and 8 access the same bank
Threads 1 and 9 access the same bank
Threads 2 and 10 access the same bank, etc.
Step 1
Stride 1
local_ids
0
1
2
3
4
5
6
7
8
9
10
partialDotProduct[index]

Banks	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	…
Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	…
Values	10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2	-1	4	11	-5	0	12	…

Threads 0 and 8 access the same bank
Threads 1 and 9 access the same bank
Threads 2 and 10 access the same bank, etc.
Step 1
Stride 1
local_ids
0
1
2
3
4
5
6
7
8
9
10
partialDotProduct[index + stride]

Banks	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	…
Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	…
Values	10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2	-1	4	11	-5	0	12	…

Figure 5. partialDotProduct[index] and partialDotProduct[index + stride] from kernel of Listing 5 exhibit 2-way bank conflicts (showed for Step 1 in this figure).

A high degree of bank conflicts can affect performance, so OpenCL local memory accesses should be organized to minimize these conflicts as much as possible. In this particular case, the kernel is not significantly affected by bank conflicts because they are only 2-way bank conflicts and the loop has few iterations. A conflict-free implementation of parallel reduction exists however as illustrated in Figure 6 and used in the kernel of Listing 6. Table 1 shows some performance improvements.

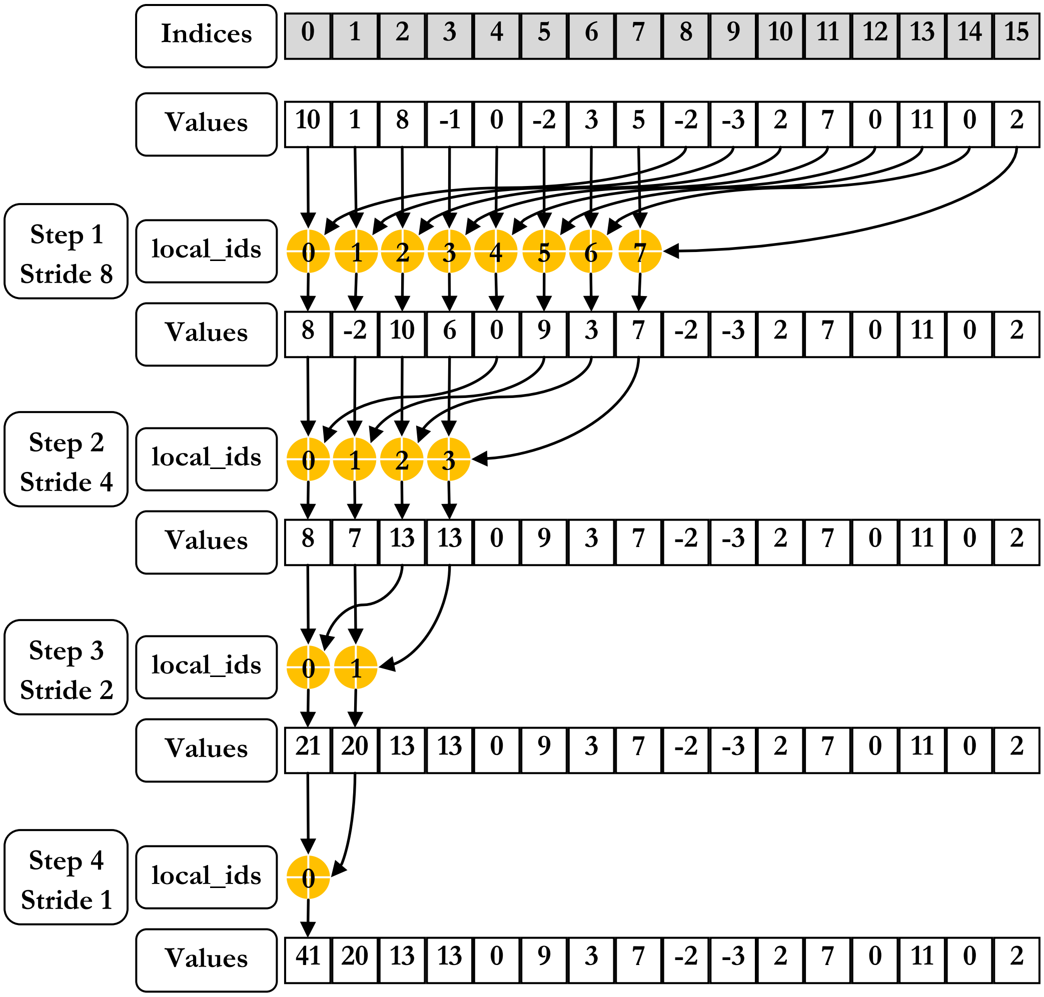

Indices 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Values 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1
Stride 8
local_ids 0 1 2 3 4 5 6 7
Values 8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 2
Stride 4
local_ids 0 1 2 3
Values 8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 3
Stride 2
local_ids 0 1
Values 21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 4
Stride 1
local_ids 0
Values 41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Figure 6. Parallel reduction without shared memory bank conflicts.

```
__kernel void MatrixVectorMul(const __global float* M,
                              uint width, uint height,
                              const __global float* V,
                              __global float* W,
                              __local float* partialDotProduct)
{
   // Each work-group computes multiple elements of W
   for (uint y = get_group_id(0); y < height  y += get_num_groups(0)) {
      const __global float* row = M + y * width;

      // Each work-item accumulates as many products as necessary
      // into local variable "sum"
      float sum = 0;
      for (uint x = get_local_id(0); x < width; x += get_local_size(0))
         sum += row[x] * V[x];

      // Each partial dot product is stored in shared memory
      partialDotProduct[get_local_id(0)] = sum;

      // Perform parallel reduction to add each work-item's
      // partial dot product together
      for (uint stride = get_local_size(0)/2; stride > 0; stride /= 2) {

         // Synchronize to make sure each work-item is done updating
         // shared memory; this is necessary because work-items read
         // results written by other work-items during each parallel
         // reduction step
         barrier(CLK_LOCAL_MEM_FENCE);

         // Only the first work-items in the work-group add elements
         // together
         if (get_local_id(0) < stride) {

            // Add two elements from the "partialDotProduct" array
            // and store the result in partialDotProduct[index]
            partialDotProduct[get_local_id(0)] +=
                            partialDotProduct[get_local_id(0) + stride];
         }
      }

      // Write the result of the reduction to global memory
      if (get_local_id(0) == 0)
         W[y] = partialDotProduct[0];

      // Synchronize to make sure the first work-item is done reading
      // partialDotProduct
      barrier(CLK_LOCAL_MEM_FENCE);
   }
}
```

Listing 6. Alternative implementation of the kernel of Listing 5, using parallel reduction without shared memory bank conflicts.

Using Constant Memory

Constant memory is fast when all threads of a warp access the same address. A typical use case for constant memory is for storing data common to all work-items of a work-group.

Instruction Optimizations

Scalar Architecture

The CUDA architecture is a scalar architecture. Therefore, there is no performance benefit from using vector types and instructions. These should only be used for convenience. It is also in general better to have more work-items than fewer using large vectors.

Trading Precision for Speed

The CUDA architecture offers ways to trade precision for speed:

- **`native_*`** and **`half_*`** mathematical functions have lower precision than their counterparts, but can be orders of magnitude faster.
- The **`-cl-mad-enable`** build option can provide large performance gains for kernels with many multiply-add operations.

Avoiding Low-Throughput Instructions

OpenCL applications should minimize use of low-throughput instructions as documented in the Programming Guide. Integer division and modulo, in particular, are expensive instructions that should be avoided or replaced with bitwise operations whenever possible.

Control Flow

Because of the SIMT execution model, threads within the same warp should follow the same execution path as much as possible. Otherwise, the different execution paths are serialized increasing overall execution time accordingly. Too many divergent warps during a kernel launch can drastically affect performance.

Note that threads in different warps are free to diverge without performance penalties.

Warp Synchronization

Since threads within a warp are inherently synchronized, synchronization at work-group level via a work-group barrier function call is only needed if there is communication between threads from different warps.

A trivial case is when only one warp is active like in the last loop iterations of the parallel reduction code of Listing 6, for example. The kernel could be rewritten by unrolling the loop and removing the barrier function call for the last iterations. It could also be rewritten to take advantage of warp synchronization as illustrated in Figure 7 and Listing 7. In a first step, each warp reduces its own 64-element portion of the array independendtly of each other, so there is no need to synchronize. The size of the array is equal to the size of the work-group, which is less or equal to 512 on NVIDIA GPUs. So, after the first step, a maximum of 8 elements remain to be reduced, which is done by a single warp. The performance gain shown in Table 1 is modest, but we expect it to increase as our OpenCL implementation improves.

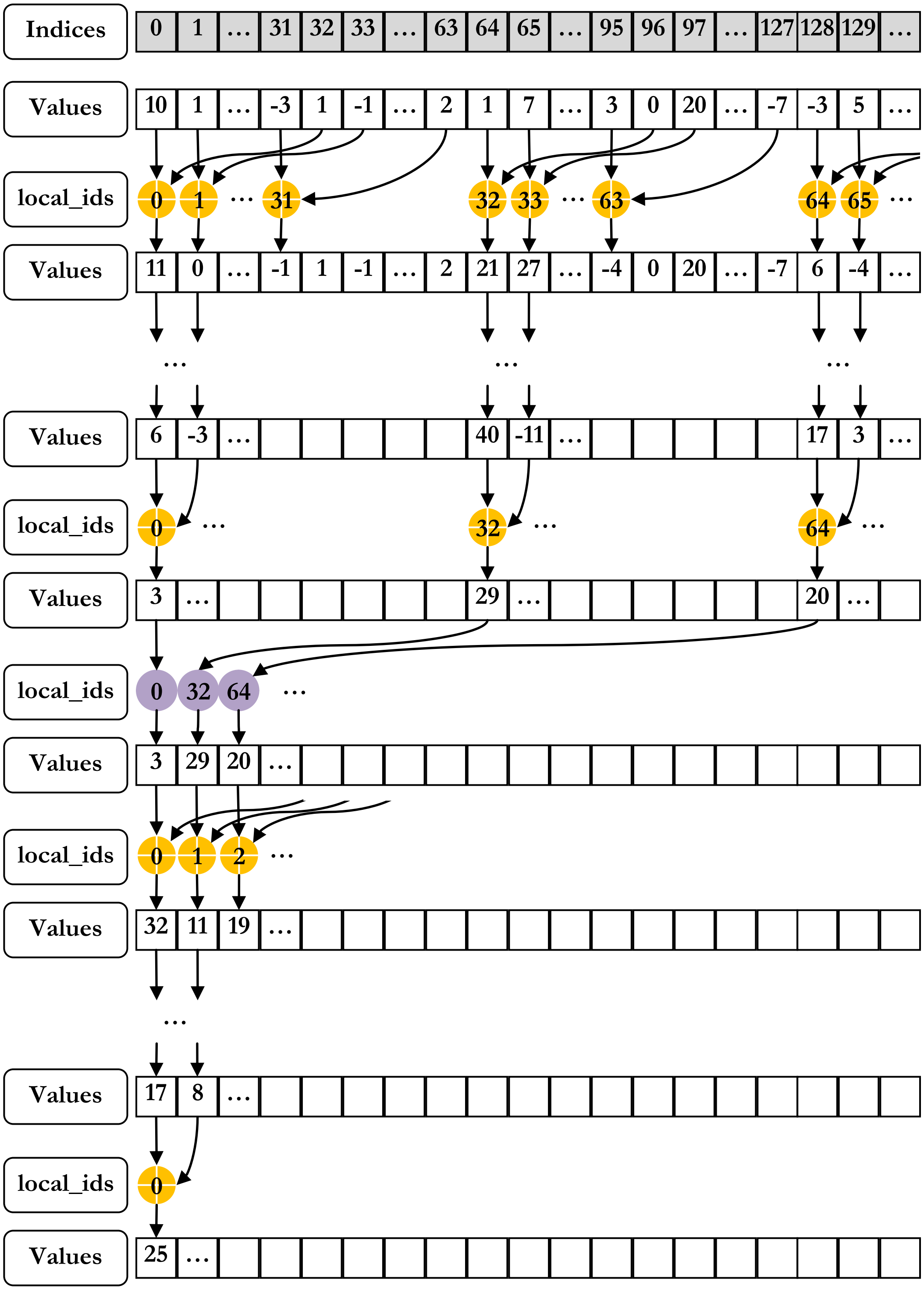


Figure 7. Warp-based parallel reduction.

```
#define WARP_SIZE 32

__kernel void MatrixVectorMul(const __global float* M,
                              uint width, uint height,
                              const __global float* V,
                              __global float* W,
                              __local float* partialDotProduct)
{
   // Each work-group computes multiple elements of W
   for (uint y = get_group_id(0); y < height; y += get_num_groups(0)) {
      const __global float* row = M + y * width;

      // Each work-item accumulates as many products as necessary
      // into local variable "sum"
      float sum = 0;
      for (uint x = get_local_id(0); x < width; x += get_local_size(0))
         sum += row[x] * V[x];

      // Each partial dot product is stored in shared memory
      partialDotProduct[get_local_id(0)] = sum;

      // Perform parallel reduction to add each work-item's
      // partial dot product together

      // Synchronize to make sure each work-item is done writing to
      // partialDotProduct
      barrier(CLK_LOCAL_MEM_FENCE);

      // Thread local ID within a warp
      uint id = get_local_id(0) & (WARP_SIZE - 1);

      // Each warp reduces 64 consecutive elements
      float warpResult = 0;
      if (get_local_id(0) < get_local_size(0) / 2) {
          volatile __local float* p = partialDotProduct
                                      + 2 * get_local_id(0) - id;
          p[0] += p[32];
          p[0] += p[16];
          p[0] += p[8];
          p[0] += p[4];6
          p[0] += p[2];
          p[0] += p[1];
          float warpResult = p[0];
      }

      // Synchronize to make sure each warp is done reading
      // partialDotProduct before it is overwritten in the next step
      barrier(CLK_LOCAL_MEM_FENCE);

      // The first thread of each warp stores the result of the reduction
      // at the beginning of partialDotProduct
      if (id == 0)
         partialDotProduct[get_local_id(0) / WARP_SIZE] = warpResult;

      // Synchronize to make sure each warp is done writing to
      // partialDotProduct before it is read in the next step
      barrier(CLK_LOCAL_MEM_FENCE);
      // Number of remaining elements after the first reduction
      uint size = get_local_size(0) / (2 * WARP_SIZE);

      // get_local_size(0) is less or equal to 512 on NVIDIA GPUs, so
      // only a single warp is needed for the following last reduction
      // step
      if (get_local_id(0) < size / 2) {
         __local float* p = partialDotProduct + get_local_id(0);
         if (size >= 8)
            p[0] += p[4];
         if (size >= 4)
            p[0] += p[2];
         if (size >= 2)
            p[0] += p[1];
      }

      // Write the result of the reduction to global memory
      if (get_local_id(0) == 0)
         W[y] = partialDotProduct[0];

      // Synchronize to make sure the first work-item is done reading
      // partialDotProduct
      barrier(CLK_LOCAL_MEM_FENCE);
   }
}
```

Listing 7. Alternative implementation of the kernel of Listing 6, using warp-based parallel reduction.

Example Execution Time

The table below gives kernel execution times measured on a GTX 285 for a matrix of width 1100 and height 60989, a work-group size of 256, 239 work groups for the first kernel, and 60 work-groups for the five last kernels.

Table 1. Kernel execution times for the various implementations of MatrixVectorMul.

MatrixVectorMul	**Execution times (ms)**
Listing 2	83.2
Listing 3	82.2
Listing 4	32.5
Listing 5	12.1

Listing 6	9.6
Listing 7	9.2

Next Step

This whitepaper provides an introduction to the main guidelines for optimizing OpenCL applications on NVIDIA GPUs. As a follow-up we recommend studying the NVIDIA OpenCL Programming Guide [1] and NVIDIA OpenCL Best Practices Guide [2], as well as the multiple code samples from the NVIDIA OpenCL SDK. The example used in this whitepaper can be found in the *oclMatVecMul* sample.

References

[1] NVIDIA OpenCL Programming Guide in NVIDIA OpenCL SDK

[2] NVIDIA OpenCL Best Practices Guide in NVIDIA OpenCL SDK

[3] Scalable Parallel Programming with CUDA, in ACM Queue, VOL 6, No. 2 (March/April 2008), © ACM, 2008. http://mags.acm.org/queue/20080304/?u1=texterity"

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

Copyright

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com