Appendix A. CUDA-ENABLED GPUS

http://developer.nvidia.com/cuda-gpus lists all CUDA-enabled devices with their compute capability.

The compute capability, number of multiprocessors, clock frequency, total amount of device memory, and other properties can be queried using the runtime (see reference manual).

Appendix B. C++ LANGUAGE EXTENSIONS

B.1. Function Execution Space Specifiers

Function execution space specifiers denote whether a function executes on the host or on the device and whether it is callable from the host or from the device.

B.1.1. \_\_global\_\_ The \_\_global\_\_ exection space specifier declares a function as being a kernel. Such a function is: ‣ Executed on the device, ‣ Callable from the host, ‣ Callable from the device for devices of compute capability 3.2 or higher (see CUDA Dynamic Parallelism for more details). A \_\_global\_\_ function must have void return type, and cannot be a member of a class. Any call to a \_\_global\_\_ function must specify its execution configuration as described in Execution Configuration. A call to a \_\_global\_\_ function is asynchronous, meaning it returns before the device has completed its execution.

B.1.2. \_\_device\_\_ The \_\_device\_\_ execution space specifier declares a function that is: ‣ Executed on the device, ‣ Callable from the device only. The \_\_global\_\_ and \_\_device\_\_ execution space specifiers cannot be used together.

B.1.3. \_\_host\_\_ The \_\_host\_\_ execution space specifier declares a function that is: ‣ Executed on the host, ‣ Callable from the host only. It is equivalent to declare a function with only the \_\_host\_\_ execution space specifier or to declare it without any of the \_\_host\_\_, \_\_device\_\_, or \_\_global\_\_ execution space specifier; in either case the function is compiled for the host only. The \_\_global\_\_ and \_\_host\_\_ execution space specifiers cannot be used together. The \_\_device\_\_ and \_\_host\_\_ execution space specifiers can be used together however, in which case the function is compiled for both the host and the device. The \_\_CUDA\_ARCH\_\_ macro introduced in Application Compatibility can be used to differentiate code paths between host and device: \_\_host\_\_ \_\_device\_\_ func() { #if \_\_CUDA\_ARCH\_\_ >= 700 // Device code path for compute capability 7.x #elif \_\_CUDA\_ARCH\_\_ >= 600 // Device code path for compute capability 6.x #elif \_\_CUDA\_ARCH\_\_ >= 500 // Device code path for compute capability 5.x #elif \_\_CUDA\_ARCH\_\_ >= 300 // Device code path for compute capability 3.x #elif !defined(\_\_CUDA\_ARCH\_\_) // Host code path #endif }

B.1.4. \_\_noinline\_\_ and \_\_forceinline\_\_ The compiler inlines any \_\_device\_\_ function when deemed appropriate. The \_\_noinline\_\_ function qualifier can be used as a hint for the compiler not to inline the function if possible. The \_\_forceinline\_\_ function qualifier can be used to force the compiler to inline the function. The \_\_noinline\_\_ and \_\_forceinline\_\_ function qualifiers cannot be used together, and neither function qualifier can be applied to an inline function. B.2. Variable Memory Space Specifiers Variable memory space specifiers denote the memory location on the device of a variable. An automatic variable declared in device code without any of the \_\_device\_\_, \_\_shared\_\_ and \_\_constant\_\_ memory space specifiers described in this section generally resides in a register. However in some cases the compiler might choose to place it in local memory, which can have adverse performance consequences as detailed in Device Memory Accesses.

B.2.1. \_\_device\_\_

The \_\_device\_\_ memory space specifier declares a variable that resides on the device. At most one of the other memory space specifiers defined in the next three sections may be used together with \_\_device\_\_ to further denote which memory space the variable belongs to. If none of them is present, the variable: ‣ Resides in global memory space, ‣ Has the lifetime of the CUDA context in which it is created, ‣ Has a distinct object per device, ‣ Is accessible from all the threads within the grid and from the host through the runtime library (cudaGetSymbolAddress() / cudaGetSymbolSize() / cudaMemcpyToSymbol() / cudaMemcpyFromSymbol()).

B.2.2. \_\_constant\_\_ The \_\_constant\_\_ memory space specifier, optionally used together with \_\_device\_\_, declares a variable that: ‣ Resides in constant memory space, ‣ Has the lifetime of the CUDA context in which it is created, ‣ Has a distinct object per device, ‣ Is accessible from all the threads within the grid and from the host through the runtime library (cudaGetSymbolAddress() / cudaGetSymbolSize() / cudaMemcpyToSymbol() / cudaMemcpyFromSymbol()).

B.2.3. \_\_shared\_\_ The \_\_shared\_\_ memory space specifier, optionally used together with \_\_device\_\_, declares a variable that: ‣ Resides in the shared memory space of a thread block, ‣ Has the lifetime of the block, ‣ Has a distinct object per block, ‣ Is only accessible from all the threads within the block, ‣ Does not have a constant address. When declaring a variable in shared memory as an external array such as extern \_\_shared\_\_ float shared[]; the size of the array is determined at launch time (see Execution Configuration). All variables declared in this fashion, start at the same address in memory, so that the layout of the variables in the array must be explicitly managed through offsets. For example, if one wants the equivalent of short array0[128]; float array1[64]; int array2[256]; in dynamically allocated shared memory, one could declare and initialize the arrays the following way: extern \_\_shared\_\_ float array[]; \_\_device\_\_ void func() // \_\_device\_\_ or \_\_global\_\_ function { short\* array0 = (short\*)array; float\* array1 = (float\*)&array0[128]; int\* array2 = (int\*)&array1[64]; } Note that pointers need to be aligned to the type they point to, so the following code, for example, does not work since array1 is not aligned to 4 bytes. extern \_\_shared\_\_ float array[]; \_\_device\_\_ void func() // \_\_device\_\_ or \_\_global\_\_ function { short\* array0 = (short\*)array; float\* array1 = (float\*)&array0[127]; } Alignment requirements for the built-in vector types are listed in Table 4.

B.2.4. \_\_managed\_\_ The \_\_managed\_\_ memory space specifier, optionally used together with \_\_device\_\_, declares a variable that: ‣ Can be referenced from both device and host code, e.g., its address can be taken or it can be read or written directly from a device or host function. ‣ Has the lifetime of an application. See \_\_managed\_\_ Memory Space Specifier for more details.

B.2.5. \_\_restrict\_\_ nvcc supports restricted pointers via the \_\_restrict\_\_ keyword. Restricted pointers were introduced in C99 to alleviate the aliasing problem that exists in C-type languages, and which inhibits all kind of optimization from code re-ordering to common sub-expression elimination.

Here is an example subject to the aliasing issue, where use of restricted pointer can help the compiler to reduce the number of instructions: void foo(const float\* a, const float\* b, float\* c) { c[0] = a[0] \* b[0]; c[1] = a[0] \* b[0]; c[2] = a[0] \* b[0] \* a[1]; c[3] = a[0] \* a[1]; c[4] = a[0] \* b[0]; c[5] = b[0]; ... } In C-type languages, the pointers a, b, and c may be aliased, so any write through c could modify elements of a or b. This means that to guarantee functional correctness, the compiler cannot load a[0] and b[0] into registers, multiply them, and store the result to both c[0] and c[1], because the results would differ from the abstract execution model if, say, a[0] is really the same location as c[0]. So the compiler cannot take advantage of the common sub-expression. Likewise, the compiler cannot just reorder the computation of c[4] into the proximity of the computation of c[0] and c[1] because the preceding write to c[3] could change the inputs to the computation of c[4]. By making a, b, and c restricted pointers, the programmer asserts to the compiler that the pointers are in fact not aliased, which in this case means writes through c would never overwrite elements of a or b. This changes the function prototype as follows: void foo(const float\* \_\_restrict\_\_ a, const float\* \_\_restrict\_\_ b, float\* \_\_restrict\_\_ c); Note that all pointer arguments need to be made restricted for the compiler optimizer to derive any benefit. With the \_\_restrict\_\_ keywords added, the compiler can now reorder and do common sub-expression elimination at will, while retaining functionality identical with the abstract execution model: void foo(const float\* \_\_restrict\_\_ a, const float\* \_\_restrict\_\_ b, float\* \_\_restrict\_\_ c) { float t0 = a[0]; float t1 = b[0]; float t2 = t0 \* t2; float t3 = a[1]; c[0] = t2; c[1] = t2; c[4] = t2; c[2] = t2 \* t3; c[3] = t0 \* t3; c[5] = t1; ... } The effects here are a reduced number of memory accesses and reduced number of computations. This is balanced by an increase in register pressure due to "cached" loads and common sub-expressions.

Since register pressure is a critical issue in many CUDA codes, use of restricted pointers can have negative performance impact on CUDA code, due to reduced occupancy. B.3. Built-in Vector Types B.3.1. char, short, int, long, longlong, float, double These are vector types derived from the basic integer and floating-point types. They are structures and the 1st, 2nd, 3rd, and 4th components are accessible through the fields x, y, z, and w, respectively. They all come with a constructor function of the form make\_<type name>; for example, int2 make\_int2(int x, int y); which creates a vector of type int2 with value (x, y). The alignment requirements of the vector types are detailed in Table 4. Table 4 Alignment Requirements Type Alignment char1, uchar1 1 char2, uchar2 2 char3, uchar3 1 char4, uchar4 4 short1, ushort1 2 short2, ushort2 4 short3, ushort3 2 short4, ushort4 8 int1, uint1 4 int2, uint2 8 int3, uint3 4 int4, uint4 16 long1, ulong1 4 if sizeof(long) is equal to sizeof(int) 8, otherwise long2, ulong2 8 if sizeof(long) is equal to sizeof(int), 16, otherwise long3, ulong3 4 if sizeof(long) is equal to sizeof(int), 8, otherwise long4, ulong4 16 www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 131 ----------------------- Page 148----------------------- C++ Language Extensions Type Alignment longlong1, ulonglong1 8 longlong2, ulonglong2 16 longlong3, ulonglong3 8 longlong4, ulonglong4 16 float1 4 float2 8 float3 4 float4 16 double1 8 double2 16 double3 8 double4 16

B.3.2. dim3 This type is an integer vector type based on uint3 that is used to specify dimensions. When defining a variable of type dim3, any component left unspecified is initialized to 1.

B.4. Built-in Variables Built-in variables specify the grid and block dimensions and the block and thread indices. They are only valid within functions that are executed on the device.

B.4.1. gridDim This variable is of type dim3 (see dim3) and contains the dimensions of the grid. B.4.2. blockIdx This variable is of type uint3 (see char, short, int, long, longlong, float, double) and contains the block index within the grid. B.4.3. blockDim This variable is of type dim3 (see dim3) and contains the dimensions of the block.

B.4.4. threadIdx This variable is of type uint3 (see char, short, int, long, longlong, float, double ) and contains the thread index within the block.

B.4.5. warpSize This variable is of type int and contains the warp size in threads (see SIMT Architecture for the definition of a warp). B.5. Memory Fence Functions The CUDA programming model assumes a device with a weakly-ordered memory model, that is the order in which a CUDA thread writes data to shared memory, global memory, page-locked host memory, or the memory of a peer device is not necessarily the order in which the data is observed being written by another CUDA or host thread. For example, if thread 1 executes writeXY() and thread 2 executes readXY() as defined in the following code sample \_\_device\_\_ volatile int X = 1, Y = 2; \_\_device\_\_ void writeXY() { X = 10; Y = 20; } \_\_device\_\_ void readXY() { int A = X; int B = Y; } it is possible that B ends up equal to 20 and A equal to 1 for thread 2. In a strongly- ordered memory model, the only possibilities would be: ‣ A equal to 1 and B equal to 2, ‣ A equal to 10 and B equal to 2, ‣ A equal to 10 and B equal to 20, Memory fence functions can be used to enforce some ordering on memory accesses. The memory fence functions differ in the scope in which the orderings are enforced but they are independent of the accessed memory space (shared memory, global memory, page- locked host memory, and the memory of a peer device). void \_\_threadfence\_block(); ensures that:

All writes to all memory made by the calling thread before the call to \_\_threadfence\_block() are observed by all threads in the block of the calling thread as occurring before all writes to all memory made by the calling thread after the call to \_\_threadfence\_block(); ‣ All reads from all memory made by the calling thread before the call to \_\_threadfence\_block() are ordered before all reads from all memory made by the calling thread after the call to \_\_threadfence\_block(). void \_\_threadfence(); acts as \_\_threadfence\_block() for all threads in the block of the calling thread and also ensures that no writes to all memory made by the calling thread after the call to \_\_threadfence() are observed by any thread in the device as occurring before any write to all memory made by the calling thread before the call to \_\_threadfence(). Note that for this ordering guarantee to be true, the observing threads must truly observe the memory and not cached versions of it; this is ensured by using the volatile keyword as detailed in Volatile Qualifier. void \_\_threadfence\_system(); acts as \_\_threadfence\_block() for all threads in the block of the calling thread and also ensures that all writes to all memory made by the calling thread before the call to \_\_threadfence\_system() are observed by all threads in the device, host threads, and all threads in peer devices as occurring before all writes to all memory made by the calling thread after the call to \_\_threadfence\_system(). \_\_threadfence\_system() is only supported by devices of compute capability 2.x and higher. In the previous code sample, inserting a fence function call between X = 10; and Y = 20; and between int A = X; and int B = Y; would ensure that for thread 2, A will always be equal to 10 if B is equal to 20. If thread 1 and 2 belong to the same block, it is enough to use \_\_threadfence\_block(). If thread 1 and 2 do not belong to the same block, \_\_threadfence() must be used if they are CUDA threads from the same device and \_\_threadfence\_system() must be used if they are CUDA threads from two different devices. A common use case is when threads consume some data produced by other threads as illustrated by the following code sample of a kernel that computes the sum of an array of N numbers in one call. Each block first sums a subset of the array and stores the result in global memory. When all blocks are done, the last block done reads each of these partial sums from global memory and sums them to obtain the final result. In order to determine which block is finished last, each block atomically increments a counter to signal that it is done with computing and storing its partial sum (see Atomic Functions about atomic functions). The last block is the one that receives the counter value equal to gridDim.x-1. If no fence is placed between storing the partial sum and incrementing the counter, the counter might increment before the partial sum is stored and therefore,

might reach gridDim.x-1 and let the last block start reading partial sums before they have been actually updated in memory. Memory fence functions only affect the ordering of memory operations by a thread; they do not ensure that these memory operations are visible to other threads (like \_\_syncthreads() does for threads within a block (see Synchronization Functions)). In the code sample below, the visibility of memory operations on the result variable is ensured by declaring it as volatile (see Volatile Qualifier). \_\_device\_\_ unsigned int count = 0; \_\_shared\_\_ bool isLastBlockDone; \_\_global\_\_ void sum(const float\* array, unsigned int N, volatile float\* result) { // Each block sums a subset of the input array. float partialSum = calculatePartialSum(array, N); if (threadIdx.x == 0) { // Thread 0 of each block stores the partial sum // to global memory. The compiler will use // a store operation that bypasses the L1 cache // since the "result" variable is declared as // volatile. This ensures that the threads of // the last block will read the correct partial // sums computed by all other blocks. result[blockIdx .x] = partialSum; // Thread 0 makes sure that the incrementation // of the "count" variable is only performed after // the partial sum has been written to global memory. \_\_threadfence(); // Thread 0 signals that it is done. unsigned int value = atomicInc(&count, gridDim.x); // Thread 0 determines if its block is the last // block to be done. isLastBlockDone = (value == (gridDim.x - 1)); } // Synchronize to make sure that each thread reads // the correct value of isLastBlockDone. \_\_syncthreads(); if (isLastBlockDone) { // The last block sums the partial sums // stored in result[0 .. gridDim.x-1] float totalSum = calculateTotalSum(result); if (threadIdx.x == 0) { // Thread 0 of last block stores the total sum // to global memory and resets the count // varialble, so that the next kernel call // works properly. result[0] = totalSum; count = 0; } } }

B.6. Synchronization Functions void \_\_syncthreads(); waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to \_\_syncthreads() are visible to all threads in the block. \_\_syncthreads() is used to coordinate communication between the threads of the same block. When some threads within a block access the same addresses in shared or global memory, there are potential read-after-write, write-after-read, or write-after- write hazards for some of these memory accesses. These data hazards can be avoided by synchronizing threads in-between these accesses. \_\_syncthreads() is allowed in conditional code but only if the conditional evaluates identically across the entire thread block, otherwise the code execution is likely to hang or produce unintended side effects. Devices of compute capability 2.x and higher support three variations of \_\_syncthreads() described below. int \_\_syncthreads\_count(int predicate); is identical to \_\_syncthreads() with the additional feature that it evaluates predicate for all threads of the block and returns the number of threads for which predicate evaluates to non-zero. int \_\_syncthreads\_and(int predicate); is identical to \_\_syncthreads() with the additional feature that it evaluates predicate for all threads of the block and returns non-zero if and only if predicate evaluates to non- zero for all of them. int \_\_syncthreads\_or(int predicate); is identical to \_\_syncthreads() with the additional feature that it evaluates predicate for all threads of the block and returns non-zero if and only if predicate evaluates to non- zero for any of them. void \_\_syncwarp(unsigned mask=0xffffffff); will cause the executing thread to wait until all warp lanes named in mask have executed a \_\_syncwarp() (with the same mask) before resuming execution. All non- exited threads named in mask must execute a corresponding \_\_syncwarp() with the same mask, or the result is undefined. Executing \_\_syncwarp() guarantees memory ordering among threads participating in the barrier. Thus, threads within a warp that wish to communicate via memory can store www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 136 ----------------------- Page 153----------------------- C++ Language Extensions to memory, execute \_\_syncwarp(), and then safely read values stored by other threads in the warp. For .target sm\_6x or below, all threads in mask must execute the same \_\_syncwarp() in convergence, and the union of all values in mask must be equal to the active mask. Otherwise, the behavior is undefined. B.7. Mathematical Functions The reference manual lists all C/C++ standard library mathematical functions that are supported in device code and all intrinsic functions that are only supported in device code. Mathematical Functions provides accuracy information for some of these functions when relevant. B.8. Texture Functions Texture objects are described in Texture Object API Texture references are described in Texture Reference API Texture fetching is described in Texture Fetching. B.8.1. Texture Object API B.8.1.1. tex1Dfetch() template<class T> T tex1Dfetch(cudaTextureObject\_t texObj, int x); fetches from the region of linear memory specified by the one-dimensional texture object texObj using integer texture coordinate x. tex1Dfetch() only works with non- normalized coordinates, so only the border and clamp addressing modes are supported. It does not perform any texture filtering. For integer types, it may optionally promote the integer to single-precision floating point. B.8.1.2. tex1D() template<class T> T tex1D(cudaTextureObject\_t texObj, float x); fetches from the CUDA array specified by the one-dimensional texture object texObj using texture coordinate x. B.8.1.3. tex1DLod() template<class T> T tex1DLod(cudaTextureObject\_t texObj, float x, float level); www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 137 ----------------------- Page 154----------------------- C++ Language Extensions fetches from the CUDA array specified by the one-dimensional texture object texObj using texture coordinate x at the level-of-detail level. B.8.1.4. tex1DGrad() template<class T> T tex1DGrad(cudaTextureObject\_t texObj, float x, float dx, float dy); fetches from the CUDA array specified by the one-dimensional texture object texObj using texture coordinate x. The level-of-detail is derived from the X-gradient dx and Y- gradient dy. B.8.1.5. tex2D() template<class T> T tex2D(cudaTextureObject\_t texObj, float x, float y); fetches from the CUDA array or the region of linear memory specified by the two- dimensional texture object texObj using texture coordinate (x,y). B.8.1.6. tex2DLod() template<class T> tex2DLod(cudaTextureObject\_t texObj, float x, float y, float level); fetches from the CUDA array or the region of linear memory specified by the two- dimensional texture object texObj using texture coordinate (x,y) at level-of-detail level. B.8.1.7. tex2DGrad() template<class T> T tex2DGrad(cudaTextureObject\_t texObj, float x, float y, float2 dx, float2 dy); fetches from the CUDA array specified by the two-dimensional texture object texObj using texture coordinate (x,y). The level-of-detail is derived from the dx and dy gradients. B.8.1.8. tex3D() template<class T> T tex3D(cudaTextureObject\_t texObj, float x, float y, float z); fetches from the CUDA array specified by the three-dimensional texture object texObj using texture coordinate (x,y,z). B.8.1.9. tex3DLod() template<class T> T tex3DLod(cudaTextureObject\_t texObj, float x, float y, float z, float level); fetches from the CUDA array or the region of linear memory specified by the three- dimensional texture object texObj using texture coordinate (x,y,z) at level-of-detail level. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 138 ----------------------- Page 155----------------------- C++ Language Extensions B.8.1.10. tex3DGrad() template<class T> T tex3DGrad(cudaTextureObject\_t texObj, float x, float y, float z, float4 dx, float4 dy); fetches from the CUDA array specified by the three-dimensional texture object texObj using texture coordinate (x,y,z) at a level-of-detail derived from the X and Y gradients dx and dy. B.8.1.11. tex1DLayered() template<class T> T tex1DLayered(cudaTextureObject\_t texObj, float x, int layer); fetches from the CUDA array specified by the one-dimensional texture object texObj using texture coordinate x and index layer, as described in Layered Textures B.8.1.12. tex1DLayeredLod() template<class T> T tex1DLayeredLod(cudaTextureObject\_t texObj, float x, int layer, float level); fetches from the CUDA array specified by the one-dimensional layered texture at layer layer using texture coordinate x and level-of-detail level. B.8.1.13. tex1DLayeredGrad() template<class T> T tex1DLayeredGrad(cudaTextureObject\_t texObj, float x, int layer, float dx, float dy); fetches from the CUDA array specified by the one-dimensional layered texture at layer layer using texture coordinate x and a level-of-detail derived from the dx and dy gradients. B.8.1.14. tex2DLayered() template<class T> T tex2DLayered(cudaTextureObject\_t texObj, float x, float y, int layer); fetches from the CUDA array specified by the two-dimensional texture object texObj using texture coordinate (x,y) and index layer, as described in Layered Textures. B.8.1.15. tex2DLayeredLod() template<class T> T tex2DLayeredLod(cudaTextureObject\_t texObj, float x, float y, int layer, float level); fetches from the CUDA array specified by the two-dimensional layered texture at layer layer using texture coordinate (x,y). www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 139 ----------------------- Page 156----------------------- C++ Language Extensions B.8.1.16. tex2DLayeredGrad() template<class T> T tex2DLayeredGrad(cudaTextureObject\_t texObj, float x, float y, int layer, float2 dx, float2 dy); fetches from the CUDA array specified by the two-dimensional layered texture at layer layer using texture coordinate (x,y) and a level-of-detail derived from the dx and dy X and Y gradients. B.8.1.17. texCubemap() template<class T> T texCubemap(cudaTextureObject\_t texObj, float x, float y, float z); fetches the CUDA array specified by the three-dimensional texture object texObj using texture coordinate (x,y,z), as described in Cubemap Textures. B.8.1.18. texCubemapLod() template<class T> T texCubemapLod(cudaTextureObject\_t texObj, float x, float, y, float z, float level); fetches from the CUDA array specified by the three-dimensional texture object texObj using texture coordinate (x,y,z) as described in Cubemap Textures. The level-of-detail used is given by level. B.8.1.19. texCubemapLayered() template<class T> T texCubemapLayered(cudaTextureObject\_t texObj, float x, float y, float z, int layer); fetches from the CUDA array specified by the cubemap layered texture object texObj using texture coordinates (x,y,z), and index layer, as described in Cubemap Layered Textures. B.8.1.20. texCubemapLayeredLod() template<class T> T texCubemapLayeredLod(cudaTextureObject\_t texObj, float x, float y, float z, int layer, float level); fetches from the CUDA array specified by the cubemap layered texture object texObj using texture coordinate (x,y,z) and index layer, as described in Cubemap Layered Textures, at level-of-detail level level. B.8.1.21. tex2Dgather() template<class T> T tex2Dgather(cudaTextureObject\_t texObj, float x, float y, int comp = 0); www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 140 ----------------------- Page 157----------------------- C++ Language Extensions fetches from the CUDA array specified by the 2D texture object texObj using texture coordinates x and y and the comp parameter as described in Texture Gather. B.8.2. Texture Reference API B.8.2.1. tex1Dfetch() template<class DataType> Type tex1Dfetch( texture<DataType, cudaTextureType1D, cudaReadModeElementType> texRef, int x); float tex1Dfetch( texture<unsigned char, cudaTextureType1D, cudaReadModeNormalizedFloat> texRef, int x); float tex1Dfetch( texture<signed char, cudaTextureType1D, cudaReadModeNormalizedFloat> texRef, int x); float tex1Dfetch( texture<unsigned short, cudaTextureType1D, cudaReadModeNormalizedFloat> texRef, int x); float tex1Dfetch( texture<signed short, cudaTextureType1D, cudaReadModeNormalizedFloat> texRef, int x); fetches from the region of linear memory bound to the one-dimensional texture reference texRef using integer texture coordinate x. tex1Dfetch() only works with non-normalized coordinates, so only the border and clamp addressing modes are supported. It does not perform any texture filtering. For integer types, it may optionally promote the integer to single-precision floating point. Besides the functions shown above, 2-, and 4-tuples are supported; for example: float4 tex1Dfetch( texture<uchar4, cudaTextureType1D, cudaReadModeNormalizedFloat> texRef, int x); fetches from the region of linear memory bound to texture reference texRef using texture coordinate x. B.8.2.2. tex1D() template<class DataType, enum cudaTextureReadMode readMode> Type tex1D(texture<DataType, cudaTextureType1D, readMode> texRef, float x); fetches from the CUDA array bound to the one-dimensional texture reference texRef using texture coordinate x. Type is equal to DataType except when readMode is equal to cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is equal to the matching floating-point type. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 141 ----------------------- Page 158----------------------- C++ Language Extensions B.8.2.3. tex1DLod() template<class DataType, enum cudaTextureReadMode readMode> Type tex1DLod(texture<DataType, cudaTextureType1D, readMode> texRef, float x, float level); fetches from the CUDA array bound to the one-dimensional texture reference texRef using texture coordinate x. The level-of-detail is given by level. Type is the same as DataType except when readMode is cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is the corresponding floating-point type. B.8.2.4. tex1DGrad() template<class DataType, enum cudaTextureReadMode readMode> Type tex1DGrad(texture<DataType, cudaTextureType1D, readMode> texRef, float x, float dx, float dy); fetches from the CUDA array bound to the one-dimensional texture reference texRef using texture coordinate x. The level-of-detail is derived from the dx and dy X- and Y-gradients. Type is the same as DataType except when readMode is cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is the corresponding floating-point type. B.8.2.5. tex2D() template<class DataType, enum cudaTextureReadMode readMode> Type tex2D(texture<DataType, cudaTextureType2D, readMode> texRef, float x, float y); fetches from the CUDA array or the region of linear memory bound to the two- dimensional texture reference texRef using texture coordinates x and y. Type is equal to DataType except when readMode is equal to cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is equal to the matching floating-point type. B.8.2.6. tex2DLod() template<class DataType, enum cudaTextureReadMode readMode> Type tex2DLod(texture<DataType, cudaTextureType2D, readMode> texRef, float x, float y, float level); fetches from the CUDA array bound to the two-dimensional texture reference texRef using texture coordinate (x,y). The level-of-detail is given by level. Type is the same as DataType except when readMode is cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is the corresponding floating-point type. B.8.2.7. tex2DGrad() template<class DataType, enum cudaTextureReadMode readMode> Type tex2DGrad(texture<DataType, cudaTextureType2D, readMode> texRef, float x, float y, float2 dx, float2 dy); www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 142 ----------------------- Page 159----------------------- C++ Language Extensions fetches from the CUDA array bound to the two-dimensional texture reference texRef using texture coordinate (x,y). The level-of-detail is derived from the dx and dy X- and Y-gradients. Type is the same as DataType except when readMode is cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is the corresponding floating-point type. B.8.2.8. tex3D() template<class DataType, enum cudaTextureReadMode readMode> Type tex3D(texture<DataType, cudaTextureType3D, readMode> texRef, float x, float y, float z); fetches from the CUDA array bound to the three-dimensional texture reference texRef using texture coordinates x, y, and z. Type is equal to DataType except when readMode is equal to cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is equal to the matching floating-point type. B.8.2.9. tex3DLod() template<class DataType, enum cudaTextureReadMode readMode> Type tex3DLod(texture<DataType, cudaTextureType3D, readMode> texRef, float x, float y, float z, float level); fetches from the CUDA array bound to the two-dimensional texture reference texRef using texture coordinate (x,y,z). The level-of-detail is given by level. Type is the same as DataType except when readMode is cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is the corresponding floating-point type. B.8.2.10. tex3DGrad() template<class DataType, enum cudaTextureReadMode readMode> Type tex3DGrad(texture<DataType, cudaTextureType3D, readMode> texRef, float x, float y, float z, float4 dx, float4 dy); fetches from the CUDA array bound to the two-dimensional texture reference texRef using texture coordinate (x,y,z). The level-of-detail is derived from the dx and dy X- and Y-gradients. Type is the same as DataType except when readMode is cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is the corresponding floating-point type. B.8.2.11. tex1DLayered() template<class DataType, enum cudaTextureReadMode readMode> Type tex1DLayered( texture<DataType, cudaTextureType1DLayered, readMode> texRef, float x, int layer); fetches from the CUDA array bound to the one-dimensional layered texture reference texRef using texture coordinate x and index layer, as described in Layered Textures. Type is equal to DataType except when readMode is equal to www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 143 ----------------------- Page 160----------------------- C++ Language Extensions cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is equal to the matching floating-point type. B.8.2.12. tex1DLayeredLod() template<class DataType, enum cudaTextureReadMode readMode> Type tex1DLayeredLod(texture<DataType, cudaTextureType1D, readMode> texRef, float x, int layer, float level); fetches from the CUDA array bound to the one-dimensional texture reference texRef using texture coordinate x and index layer as described in Layered Textures. The level- of-detail is given by level. Type is the same as DataType except when readMode is cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is the corresponding floating-point type. B.8.2.13. tex1DLayeredGrad() template<class DataType, enum cudaTextureReadMode readMode> Type tex1DLayeredGrad(texture<DataType, cudaTextureType1D, readMode> texRef, float x, int layer, float dx, float dy); fetches from the CUDA array bound to the one-dimensional texture reference texRef using texture coordinate x and index layer as described in Layered Textures. The level-of-detail is derived from the dx and dy X- and Y-gradients. Type is the same as DataType except when readMode is cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is the corresponding floating-point type. B.8.2.14. tex2DLayered() template<class DataType, enum cudaTextureReadMode readMode> Type tex2DLayered( texture<DataType, cudaTextureType2DLayered, readMode> texRef, float x, float y, int layer); fetches from the CUDA array bound to the two-dimensional layered texture reference texRef using texture coordinates x and y, and index layer, as described in Texture Memory. Type is equal to DataType except when readMode is equal to cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is equal to the matching floating-point type. B.8.2.15. tex2DLayeredLod() template<class DataType, enum cudaTextureReadMode readMode> Type tex2DLayeredLod(texture<DataType, cudaTextureType2D, readMode> texRef, float x, float y, int layer, float level); fetches from the CUDA array bound to the two-dimensional texture reference texRef using texture coordinate (x,y) and index layer as described in Layered Textures. The level-of-detail is given by level. Type is the same as DataType except when readMode is cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is the corresponding floating-point type. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 144 ----------------------- Page 161----------------------- C++ Language Extensions B.8.2.16. tex2DLayeredGrad() template<class DataType, enum cudaTextureReadMode readMode> Type tex2DLayeredGrad(texture<DataType, cudaTextureType2D, readMode> texRef, float x, float y, int layer, float2 dx, float2 dy); fetches from the CUDA array bound to the two-dimensional texture reference texRef using texture coordinate (x,y) and index layer as described in Layered Textures. The level-of-detail is derived from the dx and dy X- and Y-gradients. Type is the same as DataType except when readMode is cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is the corresponding floating-point type. B.8.2.17. texCubemap() template<class DataType, enum cudaTextureReadMode readMode> Type texCubemap( texture<DataType, cudaTextureTypeCubemap, readMode> texRef, float x, float y, float z); fetches from the CUDA array bound to the cubemap texture reference texRef using texture coordinates x, y, and z, as described in Cubemap Textures. Type is equal to DataType except when readMode is equal to cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is equal to the matching floating-point type. B.8.2.18. texCubemapLod() template<class DataType, enum cudaTextureReadMode readMode> Type texCubemapLod(texture<DataType, cudaTextureType3D, readMode> texRef, float x, float y, float z, float level); fetches from the CUDA array bound to the two-dimensional texture reference texRef using texture coordinate (x,y,z). The level-of-detail is given by level. Type is the same as DataType except when readMode is cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is the corresponding floating-point type. B.8.2.19. texCubemapLayered() template<class DataType, enum cudaTextureReadMode readMode> Type texCubemapLayered( texture<DataType, cudaTextureTypeCubemapLayered, readMode> texRef, float x, float y, float z, int layer); fetches from the CUDA array bound to the cubemap layered texture reference texRef using texture coordinates x, y, and z, and index layer, as described in Cubemap Layered Textures. Type is equal to DataType except when readMode is equal to cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is equal to the matching floating-point type. B.8.2.20. texCubemapLayeredLod() template<class DataType, enum cudaTextureReadMode readMode> Type texCubemapLayeredLod(texture<DataType, cudaTextureType3D, readMode> texRef, float x, float y, float z, int layer, float level); www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 145 ----------------------- Page 162----------------------- C++ Language Extensions fetches from the CUDA array bound to the two-dimensional texture reference texRef using texture coordinate (x,y,z) and index layer as described in Layered Textures. The level-of-detail is given by level. Type is the same as DataType except when readMode is cudaReadModeNormalizedFloat (see Texture Reference API), in which case Type is the corresponding floating-point type. B.8.2.21. tex2Dgather() template<class DataType, enum cudaTextureReadMode readMode> Type tex2Dgather( texture<DataType, cudaTextureType2D, readMode> texRef, float x, float y, int comp = 0); fetches from the CUDA array bound to the 2D texture reference texRef using texture coordinates x and y and the comp parameter as described in Texture Gather. Type is a 4- component vector type. It is based on the base type of DataType except when readMode is equal to cudaReadModeNormalizedFloat (see Texture Reference API), in which case it is always float4. B.9. Surface Functions Surface functions are only supported by devices of compute capability 2.0 and higher. Surface objects are described in described in Surface Object API Surface references are described in Surface Reference API. In the sections below, boundaryMode specifies the boundary mode, that is how out-of- range surface coordinates are handled; it is equal to either cudaBoundaryModeClamp, in which case out-of-range coordinates are clamped to the valid range, or cudaBoundaryModeZero, in which case out-of-range reads return zero and out-of-range writes are ignored, or cudaBoundaryModeTrap, in which case out-of-range accesses cause the kernel execution to fail. B.9.1. Surface Object API B.9.1.1. surf1Dread() template<class T> T surf1Dread(cudaSurfaceObject\_t surfObj, int x, boundaryMode = cudaBoundaryModeTrap); reads the CUDA array specified by the one-dimensional surface object surfObj using coordinate x. B.9.1.2. surf1Dwrite template<class T> void surf1Dwrite(T data, cudaSurfaceObject\_t surfObj, int x, boundaryMode = cudaBoundaryModeTrap); www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 146 ----------------------- Page 163----------------------- C++ Language Extensions writes value data to the CUDA array specified by the one-dimensional surface object surfObj at coordinate x. B.9.1.3. surf2Dread() template<class T> T surf2Dread(cudaSurfaceObject\_t surfObj, int x, int y, boundaryMode = cudaBoundaryModeTrap); template<class T> void surf2Dread(T\* data, cudaSurfaceObject\_t surfObj, int x, int y, boundaryMode = cudaBoundaryModeTrap); reads the CUDA array specified by the two-dimensional surface object surfObj using coordinates x and y. B.9.1.4. surf2Dwrite() template<class T> void surf2Dwrite(T data, cudaSurfaceObject\_t surfObj, int x, int y, boundaryMode = cudaBoundaryModeTrap); writes value data to the CUDA array specified by the two-dimensional surface object surfObj at coordinate x and y. B.9.1.5. surf3Dread() template<class T> T surf3Dread(cudaSurfaceObject\_t surfObj, int x, int y, int z, boundaryMode = cudaBoundaryModeTrap); template<class T> void surf3Dread(T\* data, cudaSurfaceObject\_t surfObj, int x, int y, int z, boundaryMode = cudaBoundaryModeTrap); reads the CUDA array specified by the three-dimensional surface object surfObj using coordinates x, y, and z. B.9.1.6. surf3Dwrite() template<class T> void surf3Dwrite(T data, cudaSurfaceObject\_t surfObj, int x, int y, int z, boundaryMode = cudaBoundaryModeTrap); writes value data to the CUDA array specified by the three-dimensional object surfObj at coordinate x, y, and z. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 147 ----------------------- Page 164----------------------- C++ Language Extensions B.9.1.7. surf1DLayeredread() template<class T> T surf1DLayeredread( cudaSurfaceObject\_t surfObj, int x, int layer, boundaryMode = cudaBoundaryModeTrap); template<class T> void surf1DLayeredread(T data, cudaSurfaceObject\_t surfObj, int x, int layer, boundaryMode = cudaBoundaryModeTrap); reads the CUDA array specified by the one-dimensional layered surface object surfObj using coordinate x and index layer. B.9.1.8. surf1DLayeredwrite() template<class Type> void surf1DLayeredwrite(T data, cudaSurfaceObject\_t surfObj, int x, int layer, boundaryMode = cudaBoundaryModeTrap); writes value data to the CUDA array specified by the two-dimensional layered surface object surfObj at coordinate x and index layer. B.9.1.9. surf2DLayeredread() template<class T> T surf2DLayeredread( cudaSurfaceObject\_t surfObj, int x, int y, int layer, boundaryMode = cudaBoundaryModeTrap); template<class T> void surf2DLayeredread(T data, cudaSurfaceObject\_t surfObj, int x, int y, int layer, boundaryMode = cudaBoundaryModeTrap); reads the CUDA array specified by the two-dimensional layered surface object surfObj using coordinate x and y, and index layer. B.9.1.10. surf2DLayeredwrite() template<class T> void surf2DLayeredwrite(T data, cudaSurfaceObject\_t surfObj, int x, int y, int layer, boundaryMode = cudaBoundaryModeTrap); writes value data to the CUDA array specified by the one-dimensional layered surface object surfObj at coordinate x and y, and index layer. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 148 ----------------------- Page 165----------------------- C++ Language Extensions B.9.1.11. surfCubemapread() template<class T> T surfCubemapread( cudaSurfaceObject\_t surfObj, int x, int y, int face, boundaryMode = cudaBoundaryModeTrap); template<class T> void surfCubemapread(T data, cudaSurfaceObject\_t surfObj, int x, int y, int face, boundaryMode = cudaBoundaryModeTrap); reads the CUDA array specified by the cubemap surface object surfObj using coordinate x and y, and face index face. B.9.1.12. surfCubemapwrite() template<class T> void surfCubemapwrite(T data, cudaSurfaceObject\_t surfObj, int x, int y, int face, boundaryMode = cudaBoundaryModeTrap); writes value data to the CUDA array specified by the cubemap object surfObj at coordinate x and y, and face index face. B.9.1.13. surfCubemapLayeredread() template<class T> T surfCubemapLayeredread( cudaSurfaceObject\_t surfObj, int x, int y, int layerFace, boundaryMode = cudaBoundaryModeTrap); template<class T> void surfCubemapLayeredread(T data, cudaSurfaceObject\_t surfObj, int x, int y, int layerFace, boundaryMode = cudaBoundaryModeTrap); reads the CUDA array specified by the cubemap layered surface object surfObj using coordinate x and y, and index layerFace. B.9.1.14. surfCubemapLayeredwrite() template<class T> void surfCubemapLayeredwrite(T data, cudaSurfaceObject\_t surfObj, int x, int y, int layerFace, boundaryMode = cudaBoundaryModeTrap); writes value data to the CUDA array specified by the cubemap layered object surfObj at coordinate x and y, and index layerFace. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 149 ----------------------- Page 166----------------------- C++ Language Extensions B.9.2. Surface Reference API B.9.2.1. surf1Dread() template<class Type> Type surf1Dread(surface<void, cudaSurfaceType1D> surfRef, int x, boundaryMode = cudaBoundaryModeTrap); template<class Type> void surf1Dread(Type data, surface<void, cudaSurfaceType1D> surfRef, int x, boundaryMode = cudaBoundaryModeTrap); reads the CUDA array bound to the one-dimensional surface reference surfRef using coordinate x. B.9.2.2. surf1Dwrite template<class Type> void surf1Dwrite(Type data, surface<void, cudaSurfaceType1D> surfRef, int x, boundaryMode = cudaBoundaryModeTrap); writes value data to the CUDA array bound to the one-dimensional surface reference surfRef at coordinate x. B.9.2.3. surf2Dread() template<class Type> Type surf2Dread(surface<void, cudaSurfaceType2D> surfRef, int x, int y, boundaryMode = cudaBoundaryModeTrap); template<class Type> void surf2Dread(Type\* data, surface<void, cudaSurfaceType2D> surfRef, int x, int y, boundaryMode = cudaBoundaryModeTrap); reads the CUDA array bound to the two-dimensional surface reference surfRef using coordinates x and y. B.9.2.4. surf2Dwrite() template<class Type> void surf3Dwrite(Type data, surface<void, cudaSurfaceType3D> surfRef, int x, int y, int z, boundaryMode = cudaBoundaryModeTrap); writes value data to the CUDA array bound to the two-dimensional surface reference surfRef at coordinate x and y. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 150 ----------------------- Page 167----------------------- C++ Language Extensions B.9.2.5. surf3Dread() template<class Type> Type surf3Dread(surface<void, cudaSurfaceType3D> surfRef, int x, int y, int z, boundaryMode = cudaBoundaryModeTrap); template<class Type> void surf3Dread(Type\* data, surface<void, cudaSurfaceType3D> surfRef, int x, int y, int z, boundaryMode = cudaBoundaryModeTrap); reads the CUDA array bound to the three-dimensional surface reference surfRef using coordinates x, y, and z. B.9.2.6. surf3Dwrite() template<class Type> void surf3Dwrite(Type data, surface<void, cudaSurfaceType3D> surfRef, int x, int y, int z, boundaryMode = cudaBoundaryModeTrap); writes value data to the CUDA array bound to the three-dimensional surface reference surfRef at coordinate x, y, and z. B.9.2.7. surf1DLayeredread() template<class Type> Type surf1DLayeredread( surface<void, cudaSurfaceType1DLayered> surfRef, int x, int layer, boundaryMode = cudaBoundaryModeTrap); template<class Type> void surf1DLayeredread(Type data, surface<void, cudaSurfaceType1DLayered> surfRef, int x, int layer, boundaryMode = cudaBoundaryModeTrap); reads the CUDA array bound to the one-dimensional layered surface reference surfRef using coordinate x and index layer. B.9.2.8. surf1DLayeredwrite() template<class Type> void surf1DLayeredwrite(Type data, surface<void, cudaSurfaceType1DLayered> surfRef, int x, int layer, boundaryMode = cudaBoundaryModeTrap); writes value data to the CUDA array bound to the two-dimensional layered surface reference surfRef at coordinate x and index layer. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 151 ----------------------- Page 168----------------------- C++ Language Extensions B.9.2.9. surf2DLayeredread() template<class Type> Type surf2DLayeredread( surface<void, cudaSurfaceType2DLayered> surfRef, int x, int y, int layer, boundaryMode = cudaBoundaryModeTrap); template<class Type> void surf2DLayeredread(Type data, surface<void, cudaSurfaceType2DLayered> surfRef, int x, int y, int layer, boundaryMode = cudaBoundaryModeTrap); reads the CUDA array bound to the two-dimensional layered surface reference surfRef using coordinate x and y, and index layer. B.9.2.10. surf2DLayeredwrite() template<class Type> void surf2DLayeredwrite(Type data, surface<void, cudaSurfaceType2DLayered> surfRef, int x, int y, int layer, boundaryMode = cudaBoundaryModeTrap); writes value data to the CUDA array bound to the one-dimensional layered surface reference surfRef at coordinate x and y, and index layer. B.9.2.11. surfCubemapread() template<class Type> Type surfCubemapread( surface<void, cudaSurfaceTypeCubemap> surfRef, int x, int y, int face, boundaryMode = cudaBoundaryModeTrap); template<class Type> void surfCubemapread(Type data, surface<void, cudaSurfaceTypeCubemap> surfRef, int x, int y, int face, boundaryMode = cudaBoundaryModeTrap); reads the CUDA array bound to the cubemap surface reference surfRef using coordinate x and y, and face index face. B.9.2.12. surfCubemapwrite() template<class Type> void surfCubemapwrite(Type data, surface<void, cudaSurfaceTypeCubemap> surfRef, int x, int y, int face, boundaryMode = cudaBoundaryModeTrap); writes value data to the CUDA array bound to the cubemap reference surfRef at coordinate x and y, and face index face. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 152 ----------------------- Page 169----------------------- C++ Language Extensions B.9.2.13. surfCubemapLayeredread() template<class Type> Type surfCubemapLayeredread( surface<void, cudaSurfaceTypeCubemapLayered> surfRef, int x, int y, int layerFace, boundaryMode = cudaBoundaryModeTrap); template<class Type> void surfCubemapLayeredread(Type data, surface<void, cudaSurfaceTypeCubemapLayered> surfRef, int x, int y, int layerFace, boundaryMode = cudaBoundaryModeTrap); reads the CUDA array bound to the cubemap layered surface reference surfRef using coordinate x and y, and index layerFace. B.9.2.14. surfCubemapLayeredwrite() template<class Type> void surfCubemapLayeredwrite(Type data, surface<void, cudaSurfaceTypeCubemapLayered> surfRef, int x, int y, int layerFace, boundaryMode = cudaBoundaryModeTrap); writes value data to the CUDA array bound to the cubemap layered reference surfRef at coordinate x and y, and index layerFace. B.10. Read-Only Data Cache Load Function The read-only data cache load function is only supported by devices of compute capability 3.5 and higher. T \_\_ldg(const T\* address); returns the data of type T located at address address, where T is char, signed char, short, int, long, long long unsigned char, unsigned short, unsigned int, unsigned long, unsigned long long, char2, char4, short2, short4, int2, int4, longlong2 uchar2, uchar4, ushort2, ushort4, uint2, uint4, ulonglong2 float, float2, float4, double, or double2. The operation is cached in the read-only data cache (see Global Memory). B.11. Time Function clock\_t clock(); long long int clock64(); when executed in device code, returns the value of a per-multiprocessor counter that is incremented every clock cycle. Sampling this counter at the beginning and at the end of a kernel, taking the difference of the two samples, and recording the result per thread provides a measure for each thread of the number of clock cycles taken by the device to completely execute the thread, but not of the number of clock cycles the device actually www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 153 ----------------------- Page 170----------------------- C++ Language Extensions spent executing thread instructions. The former number is greater than the latter since threads are time sliced. B.12. Atomic Functions An atomic function performs a read-modify-write atomic operation on one 32-bit or 64- bit word residing in global or shared memory. For example, atomicAdd() reads a word at some address in global or shared memory, adds a number to it, and writes the result back to the same address. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete. Atomic functions do not act as memory fences and do not imply synchronization or ordering constraints for memory operations (see Memory Fence Functions for more details on memory fences). Atomic functions can only be used in device functions. On GPU architectures with compute capability lower than 6.x, atomics operations done from the GPU are atomic only with respect to that GPU. If the GPU attempts an atomic operation to a peer GPU’s memory, the operation appears as a regular read followed by a write to the peer GPU, and the two operations are not done as one single atomic operation. Similarly, atomic operations from the GPU to CPU memory will not be atomic with respect to CPU initiated atomic operations. Compute capability 6.x introduces new type of atomics which allows developers to widen or narrow the scope of an atomic operation. For example, atomicAdd\_system guarantees that the instruction is atomic with respect to other CPUs and GPUs in the system. atomicAdd\_block implies that the instruction is atomic only with respect atomics from other threads in the same thread block. In the following example both CPU and GPU can atomically update integer value at address addr: \_\_global\_\_ void mykernel(int \*addr) { atomicAdd\_system(addr, 10); // only available on devices with compute capability 6.x } void foo() { int \*addr; cudaMallocManaged(&addr, 4); \*addr = 0; mykernel<<<...>>> (addr); \_\_sync\_fetch\_and\_add(addr, 10); // CPU atomic operation } System wide atomics are not supported on Tegra devices with compute capability less than 7.2. The new scoped versions of atomics are available for all atomics listed below only for compute capabilities 6.x and later. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 154 ----------------------- Page 171----------------------- C++ Language Extensions Note that any atomic operation can be implemented based on atomicCAS() (Compare And Swap). For example, atomicAdd() for double-precision floating-point numbers is not available on devices with compute capability lower than 6.0 but it can be implemented as follows: #if \_\_CUDA\_ARCH\_\_ < 600 \_\_device\_\_ double atomicAdd(double\* address, double val) { unsigned long long int\* address\_as\_ull = (unsigned long long int\*)address; unsigned long long int old = \*address\_as\_ull, assumed; do { assumed = old; old = atomicCAS(address\_as\_ull, assumed, \_\_double\_as\_longlong(val + \_\_longlong\_as\_double(assumed))); // Note: uses integer comparison to avoid hang in case of NaN (since NaN != NaN) } while (assumed != old); return \_\_longlong\_as\_double(old); } #endif B.12.1. Arithmetic Functions B.12.1.1. atomicAdd() int atomicAdd(int\* address, int val); unsigned int atomicAdd(unsigned int\* address, unsigned int val); unsigned long long int atomicAdd(unsigned long long int\* address, unsigned long long int val); float atomicAdd(float\* address, float val); double atomicAdd(double\* address, double val); \_\_half2 atomicAdd(\_\_half2 \*address, \_\_half2 val); \_\_half atomicAdd(\_\_half \*address, \_\_half val); reads the 16-bit, 32-bit or 64-bit word old located at the address address in global or shared memory, computes (old + val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old. The 32-bit floating-point version of atomicAdd() is only supported by devices of compute capability 2.x and higher. The 64-bit floating-point version of atomicAdd() is only supported by devices of compute capability 6.x and higher. The 32-bit \_\_half2 floating-point version of atomicAdd() is only supported by devices of compute capability 6.x and higher. The atomicity of the \_\_half2 add operation is guaranteed separately for each of the two \_\_half elements; the entire \_\_half2 is not guaranteed to be atomic as a single 32-bit access. The 16-bit \_\_half floating-point version of atomicAdd() is only supported by devices of compute capability 7.x and higher. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 155 ----------------------- Page 172----------------------- C++ Language Extensions B.12.1.2. atomicSub() int atomicSub(int\* address, int val); unsigned int atomicSub(unsigned int\* address, unsigned int val); reads the 32-bit word old located at the address address in global or shared memory, computes (old - val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old. B.12.1.3. atomicExch() int atomicExch(int\* address, int val); unsigned int atomicExch(unsigned int\* address, unsigned int val); unsigned long long int atomicExch(unsigned long long int\* address, unsigned long long int val); float atomicExch(float\* address, float val); reads the 32-bit or 64-bit word old located at the address address in global or shared memory and stores val back to memory at the same address. These two operations are performed in one atomic transaction. The function returns old. B.12.1.4. atomicMin() int atomicMin(int\* address, int val); unsigned int atomicMin(unsigned int\* address, unsigned int val); unsigned long long int atomicMin(unsigned long long int\* address, unsigned long long int val); reads the 32-bit or 64-bit word old located at the address address in global or shared memory, computes the minimum of old and val, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old. The 64-bit version of atomicMin() is only supported by devices of compute capability 3.5 and higher. B.12.1.5. atomicMax() int atomicMax(int\* address, int val); unsigned int atomicMax(unsigned int\* address, unsigned int val); unsigned long long int atomicMax(unsigned long long int\* address, unsigned long long int val); reads the 32-bit or 64-bit word old located at the address address in global or shared memory, computes the maximum of old and val, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old. The 64-bit version of atomicMax() is only supported by devices of compute capability 3.5 and higher. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 156 ----------------------- Page 173----------------------- C++ Language Extensions B.12.1.6. atomicInc() unsigned int atomicInc(unsigned int\* address, unsigned int val); reads the 32-bit word old located at the address address in global or shared memory, computes ((old >= val) ? 0 : (old+1)), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old. B.12.1.7. atomicDec() unsigned int atomicDec(unsigned int\* address, unsigned int val); reads the 32-bit word old located at the address address in global or shared memory, computes (((old == 0) || (old > val)) ? val : (old-1) ), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old. B.12.1.8. atomicCAS() int atomicCAS(int\* address, int compare, int val); unsigned int atomicCAS(unsigned int\* address, unsigned int compare, unsigned int val); unsigned long long int atomicCAS(unsigned long long int\* address, unsigned long long int compare, unsigned long long int val); unsigned short int atomicCAS(unsigned short int \*address, unsigned short int compare, unsigned short int val); reads the 16-bit, 32-bit or 64-bit word old located at the address address in global or shared memory, computes (old == compare ? val : old) , and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old (Compare And Swap). B.12.2. Bitwise Functions B.12.2.1. atomicAnd() int atomicAnd(int\* address, int val); unsigned int atomicAnd(unsigned int\* address, unsigned int val); unsigned long long int atomicAnd(unsigned long long int\* address, unsigned long long int val); reads the 32-bit or 64-bit word old located at the address address in global or shared memory, computes (old & val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 157 ----------------------- Page 174----------------------- C++ Language Extensions The 64-bit version of atomicAnd() is only supported by devices of compute capability 3.5 and higher. B.12.2.2. atomicOr() int atomicOr(int\* address, int val); unsigned int atomicOr(unsigned int\* address, unsigned int val); unsigned long long int atomicOr(unsigned long long int\* address, unsigned long long int val); reads the 32-bit or 64-bit word old located at the address address in global or shared memory, computes (old | val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old. The 64-bit version of atomicOr() is only supported by devices of compute capability 3.5 and higher. B.12.2.3. atomicXor() int atomicXor(int\* address, int val); unsigned int atomicXor(unsigned int\* address, unsigned int val); unsigned long long int atomicXor(unsigned long long int\* address, unsigned long long int val); reads the 32-bit or 64-bit word old located at the address address in global or shared memory, computes (old ^ val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old. The 64-bit version of atomicXor() is only supported by devices of compute capability 3.5 and higher. B.13. Address Space Predicate Functions B.13.1. \_\_isGlobal() unsigned int \_\_isGlobal(const void \*ptr); Returns 1 if ptr contains the generic address of an object in global memory space, otherwise returns 0. B.13.2. \_\_isShared() unsigned int \_\_isShared(const void \*ptr); Returns 1 if ptr contains the generic address of an object in shared memory space, otherwise returns 0. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 158 ----------------------- Page 175----------------------- C++ Language Extensions B.13.3. \_\_isConstant() unsigned int \_\_isConstant(const void \*ptr); Returns 1 if ptr contains the generic address of an object in constant memory space, otherwise returns 0. B.13.4. \_\_isLocal() unsigned int \_\_isLocal(const void \*ptr); Returns 1 if ptr contains the generic address of an object in local memory space, otherwise returns 0. B.14. Warp Vote Functions int \_\_all\_sync(unsigned mask, int predicate); int \_\_any\_sync(unsigned mask, int predicate); unsigned \_\_ballot\_sync(unsigned mask, int predicate); unsigned \_\_activemask(); Deprecation notice: \_\_any, \_\_all, and \_\_ballot have been deprecated in CUDA 9.0 for all devices. Removal notice: When targeting devices with compute capability 7.x or higher, \_\_any, \_\_all, and \_\_ballot are no longer available and their sync variants should be used instead. The warp vote functions allow the threads of a given warp to perform a reduction-and- broadcast operation. These functions take as input an integer predicate from each thread in the warp and compare those values with zero. The results of the comparisons are combined (reduced) across the active threads of the warp in one of the following ways, broadcasting a single return value to each participating thread: \_\_all\_sync(unsigned mask, predicate): Evaluate predicate for all non-exited threads in mask and return non-zero if and only if predicate evaluates to non-zero for all of them. \_\_any\_sync(unsigned mask, predicate): Evaluate predicate for all non-exited threads in mask and return non-zero if and only if predicate evaluates to non-zero for any of them. \_\_ballot\_sync(unsigned mask, predicate): Evaluate predicate for all non-exited threads in mask and return an integer whose Nth bit is set if and only if predicate evaluates to non-zero for the Nth thread of the warp and the Nth thread is active. \_\_activemask(): Returns a 32-bit integer mask of all currently active threads in the calling warp. The Nth bit is set if the Nth lane in the warp is active when \_\_activemask() is called. Inactive threads are represented by 0 bits in the returned mask. Threads which have exited the program are always marked as inactive. Note that threads that are convergent at an \_\_activemask() call are not guaranteed to be convergent at www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 159 ----------------------- Page 176----------------------- C++ Language Extensions subsequent instructions unless those instructions are synchronizing warp-builtin functions. Notes For \_\_all\_sync, \_\_any\_sync, and \_\_ballot\_sync, a mask must be passed that specifies the threads participating in the call. A bit, representing the thread's lane ID, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware. All active threads named in mask must execute the same intrinsic with the same mask, or the result is undefined. B.15. Warp Match Functions \_\_match\_any\_sync and \_\_match\_all\_sync perform a broadcast-and-compare operation of a variable between threads within a warp. Supported by devices of compute capability 7.x or higher. B.15.1. Synopsys unsigned int \_\_match\_any\_sync(unsigned mask, T value); unsigned int \_\_match\_all\_sync(unsigned mask, T value, int \*pred); T can be int, unsigned int, long, unsigned long, long long, unsigned long long, float or double. B.15.2. Description The \_\_match\_sync() intrinsics permit a broadcast-and-compare of a value value across threads in a warp after synchronizing threads named in mask. \_\_match\_any\_sync Returns mask of threads that have same value of value in mask \_\_match\_all\_sync Returns mask if all threads in mask have the same value for value; otherwise 0 is returned. Predicate pred is set to true if all threads in mask have the same value of value; otherwise the predicate is set to false. The new \*\_sync match intrinsics take in a mask indicating the threads participating in the call. A bit, representing the thread's lane id, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware. All non-exited threads named in mask must execute the same intrinsic with the same mask, or the result is undefined. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 160 ----------------------- Page 177----------------------- C++ Language Extensions B.16. Warp Shuffle Functions \_\_shfl\_sync, \_\_shfl\_up\_sync, \_\_shfl\_down\_sync, and \_\_shfl\_xor\_sync exchange a variable between threads within a warp. Supported by devices of compute capability 3.x or higher. Deprecation Notice: \_\_shfl, \_\_shfl\_up, \_\_shfl\_down, and \_\_shfl\_xor have been deprecated in CUDA 9.0 for all devices. Removal Notice: When targeting devices with compute capability 7.x or higher, \_\_shfl, \_\_shfl\_up, \_\_shfl\_down, and \_\_shfl\_xor are no longer available and their sync variants should be used instead. B.16.1. Synopsis T \_\_shfl\_sync(unsigned mask, T var, int srcLane, int width=warpSize); T \_\_shfl\_up\_sync(unsigned mask, T var, unsigned int delta, int width=warpSize); T \_\_shfl\_down\_sync(unsigned mask, T var, unsigned int delta, int width=warpSize); T \_\_shfl\_xor\_sync(unsigned mask, T var, int laneMask, int width=warpSize); T can be int, unsigned int, long, unsigned long, long long, unsigned long long, float or double. With the cuda\_fp16.h header included, T can also be \_\_half or \_\_half2. B.16.2. Description The \_\_shfl\_sync() intrinsics permit exchanging of a variable between threads within a warp without use of shared memory. The exchange occurs simultaneously for all active threads within the warp (and named in mask), moving 4 or 8 bytes of data per thread depending on the type. Threads within a warp are referred to as lanes, and may have an index between 0 and warpSize-1 (inclusive). Four source-lane addressing modes are supported: \_\_shfl\_sync() Direct copy from indexed lane \_\_shfl\_up\_sync() Copy from a lane with lower ID relative to caller \_\_shfl\_down\_sync() Copy from a lane with higher ID relative to caller \_\_shfl\_xor\_sync() Copy from a lane based on bitwise XOR of own lane ID www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 161 ----------------------- Page 178----------------------- C++ Language Extensions Threads may only read data from another thread which is actively participating in the \_\_shfl\_sync() command. If the target thread is inactive, the retrieved value is undefined. All of the \_\_shfl\_sync() intrinsics take an optional width parameter which alters the behavior of the intrinsic. width must have a value which is a power of 2; results are undefined if width is not a power of 2, or is a number greater than warpSize. \_\_shfl\_sync() returns the value of var held by the thread whose ID is given by srcLane. If width is less than warpSize then each subsection of the warp behaves as a separate entity with a starting logical lane ID of 0. If srcLane is outside the range [0:width-1], the value returned corresponds to the value of var held by the srcLane modulo width (i.e. within the same subsection). \_\_shfl\_up\_sync() calculates a source lane ID by subtracting delta from the caller's lane ID. The value of var held by the resulting lane ID is returned: in effect, var is shifted up the warp by delta lanes. If width is less than warpSize then each subsection of the warp behaves as a separate entity with a starting logical lane ID of 0. The source lane index will not wrap around the value of width, so effectively the lower delta lanes will be unchanged. \_\_shfl\_down\_sync() calculates a source lane ID by adding delta to the caller's lane ID. The value of var held by the resulting lane ID is returned: this has the effect of shifting var down the warp by delta lanes. If width is less than warpSize then each subsection of the warp behaves as a separate entity with a starting logical lane ID of 0. As for \_\_shfl\_up\_sync(), the ID number of the source lane will not wrap around the value of width and so the upper delta lanes will remain unchanged. \_\_shfl\_xor\_sync() calculates a source line ID by performing a bitwise XOR of the caller's lane ID with laneMask: the value of var held by the resulting lane ID is returned. If width is less than warpSize then each group of width consecutive threads are able to access elements from earlier groups of threads, however if they attempt to access elements from later groups of threads their own value of var will be returned. This mode implements a butterfly addressing pattern such as is used in tree reduction and broadcast. The new \*\_sync shfl intrinsics take in a mask indicating the threads participating in the call. A bit, representing the thread's lane id, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware. All non-exited threads named in mask must execute the same intrinsic with the same mask, or the result is undefined. B.16.3. Notes Threads may only read data from another thread which is actively participating in the \_\_shfl\_sync() command. If the target thread is inactive, the retrieved value is undefined. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 162 ----------------------- Page 179----------------------- C++ Language Extensions width must be a power-of-2 (i.e., 2, 4, 8, 16 or 32). Results are unspecified for other values. B.16.4. Examples B.16.4.1. Broadcast of a single value across a warp #include <stdio.h> \_\_global\_\_ void bcast(int arg) { int laneId = threadIdx.x & 0x1f; int value; if (laneId == 0) // Note unused variable for value = arg; // all threads except lane 0 value = \_\_shfl\_sync(0xffffffff, value, 0); // Synchronize all threads in warp, and get "value" from lane 0 if (value != arg) printf("Thread %d failed.\n", threadIdx.x); } int main() { bcast<<< 1, 32 >>> (1234); cudaDeviceSynchronize(); return 0; } B.16.4.2. Inclusive plus-scan across sub-partitions of 8 threads #include <stdio.h> \_\_global\_\_ void scan4() { int laneId = threadIdx.x & 0x1f; // Seed sample starting value (inverse of lane ID) int value = 31 - laneId; // Loop to accumulate scan within my partition. // Scan requires log2(n) == 3 steps for 8 threads // It works by an accumulated sum up the warp // by 1, 2, 4, 8 etc. steps. for (int i=1; i<=4; i\*=2) { // We do the \_\_shfl\_sync unconditionally so that we // can read even from threads which won't do a // sum, and then conditionally assign the result. int n = \_\_shfl\_up\_sync(0xffffffff, value, i, 8); if ((laneId & 7) >= i) value += n; } printf("Thread %d final value = %d\n", threadIdx.x, value); } int main() { scan4<<< 1, 32 >>> (); cudaDeviceSynchronize(); return 0; } www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 163 ----------------------- Page 180----------------------- C++ Language Extensions B.16.4.3. Reduction across a warp #include <stdio.h> \_\_global\_\_ void warpReduce() { int laneId = threadIdx.x & 0x1f; // Seed starting value as inverse lane ID int value = 31 - laneId; // Use XOR mode to perform butterfly reduction for (int i=16; i>=1; i/=2) value += \_\_shfl\_xor\_sync(0xffffffff, value, i, 32); // "value" now contains the sum across all threads printf("Thread %d final value = %d\n", threadIdx.x, value); } int main() { warpReduce<<< 1, 32 >>> (); cudaDeviceSynchronize(); return 0; } B.17. Warp matrix functions C++ warp matrix operations leverage Tensor Cores to accelerate matrix problems of the form D=A\*B+C. These operations are supported on mixed-precision floating point data for devices of compute capability 7.0 or higher. This requires co-operation from all threads in a warp. In addition, these operations are allowed in conditional code only if the condition evaluates identically across the entire warp, otherwise the code execution is likely to hang. B.17.1. Description All following functions and types are defined in the namespace nvcuda::wmma. Sub- byte operations are considered preview, i.e. the data structures and APIs for them are subject to change and may not be compatible with future releases. This extra functionality is defined in the nvcuda::wmma::experimental namespace. template<typename Use, int m, int n, int k, typename T, typename Layout=void> class fragment; void load\_matrix\_sync(fragment<...> &a, const T\* mptr, unsigned ldm); void load\_matrix\_sync(fragment<...> &a, const T\* mptr, unsigned ldm, layout\_t layout); void store\_matrix\_sync(T\* mptr, const fragment<...> &a, unsigned ldm, layout\_t layout); void fill\_fragment(fragment<...> &a, const T& v); void mma\_sync(fragment<...> &d, const fragment<...> &a, const fragment<...> &b, const fragment<...> &c, bool satf=false); www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 164 ----------------------- Page 181----------------------- C++ Language Extensions fragment An overloaded class containing a section of a matrix distributed across all threads in the warp. The mapping of matrix elements into fragment internal storage is unspecified and subject to change in future architectures. Only certain combinations of template arguments are allowed. The first template parameter specifies how the fragment will participate in the matrix operation. Acceptable values for Use are: ‣ matrix\_a when the fragment is used as the first multiplicand, A, ‣ matrix\_b when the fragment is used as the second multiplicand, B, or ‣ accumulator when the fragment is used as the source or destination accumulators (C or D, respectively). The m, n and k sizes describe the shape of the warp-wide matrix tiles participating in the multiply-accumulate operation. The dimension of each tile depends on its role. For matrix\_a the tile takes dimension m x k; for matrix\_b the dimension is k x n, and accumulator tiles are m x n. The data type, T, may be \_\_half, char, or unsigned char for multiplicands and \_\_half, float, or int for accumulators. As documented in Element Types & Matrix Sizes, limited combinations of accumulator and multiplicand types are supported. The Layout parameter must be specified for matrix\_a and matrix\_b fragments. row\_major or col\_major indicate that elements within a matrix row or column are contiguous in memory, respectively. The Layout parameter for an accumulator matrix should retain the default value of void. A row or column layout is specified only when the accumulator is loaded or stored as described below. load\_matrix\_sync Waits until all warp lanes have arrived at load\_matrix\_sync and then loads the matrix fragment a from memory. mptr must be a 256-bit aligned pointer pointing to the first element of the matrix in memory. ldm describes the stride in elements between consecutive rows (for row major layout) or columns (for column major layout) and must be a multiple of 16 bytes (i.e., 8 \_\_half elements or 4 float elements). If the fragment is an accumulator, the layout argument must be specified as either mem\_row\_major or mem\_col\_major. For matrix\_a and matrix\_b fragments, the layout is inferred from the fragment's layout parameter. The values of mptr, ldm, layout and all template parameters for a must be the same for all threads in the warp. This function must be called by all threads in the warp, or the result is undefined. store\_matrix\_sync Waits until all warp lanes have arrived at store\_matrix\_sync and then stores the matrix fragment a to memory. mptr must be a 256-bit aligned pointer pointing to the first element of the matrix in memory. ldm describes the stride in elements between consecutive rows (for row major layout) or columns (for column major layout) and www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 165 ----------------------- Page 182----------------------- C++ Language Extensions must be a multiple of 16 bytes. The layout of the output matrix must be specified as either mem\_row\_major or mem\_col\_major. The values of mptr, ldm, layout and all template parameters for a must be the same for all threads in the warp. fill\_fragment Fill a matrix fragment with a constant value v. Because the mapping of matrix elements to each fragment is unspecified, this function is ordinarily called by all threads in the warp with a common value for v. mma\_sync Waits until all warp lanes have arrived at mma\_sync, and then performs the warp- synchronous matrix multiply-accumulate operation D=A\*B+C. The in-place operation, C=A\*B+C, is also supported. The value of satf and template parameters for each matrix fragment must be the same for all threads in the warp. Also, the template parameters m, n and k must match between fragments A, B, C and D. This function must be called by all threads in the warp, or the result is undefined. Saturation on integer accumulators will clamp the output to the maximum (or minimum) 32-bit signed integer value. Otherwise, if the accumulation would overflow, the value wraps. For floating point accumulators, the satf (saturate-to-finite value) mode parameter is deprecated. Using it can lead to unexpected results. For floating point accumulators, if satf (saturate to finite value) mode is true, the following additional numerical properties apply for the destination accumulator: ‣ If an element result is +Infinity, the corresponding accumulator will contain +MAX\_NORM ‣ If an element result is -Infinity, the corresponding accumulator will contain - MAX\_NORM ‣ If an element result is NaN, the corresponding accumulator will contain +0 Because the map of matrix elements into each thread's fragment is unspecified, individual matrix elements must be accessed from memory (shared or global) after calling store\_matrix\_sync. In the special case where all threads in the warp will apply an element-wise operation uniformly to all fragment elements, direct element access can be implemented using the following fragment class members. enum fragment<Use, m, n, k, T, Layout>::num\_elements; T fragment<Use, m, n, k, T, Layout>::x[num\_elements]; As an example, the following code scales an accumulator matrix tile by half. wmma::fragment<wmma::accumulator, 16, 16, 16, float> frag; float alpha = 0.5f; // Same value for all threads in warp ... for(int t=0; t<frag.num\_elements; t++) frag.x[t] \*= alpha; www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 166 ----------------------- Page 183----------------------- C++ Language Extensions B.17.2. Sub-byte Operations Sub-byte WMMA operations provide a way to access the low-precision capabilities of Tensor Cores. They are considered a preview feature i.e. the data structures and APIs for them are subject to change and may not be compatible with future releases. This functionality is available via the nvcuda::wmma::experimental namespace: namespace experimental { namespace precision { struct u4; // 4-bit unsigned struct s4; // 4-bit signed struct b1; // 1-bit } enum bmmaBitOp { bmmaBitOpXOR = 1 }; enum bmmaAccumulateOp { bmmaAccumulateOpPOPC = 1 }; } For 4 bit precision, the APIs available remain the same, but you must specify experimental::precision::u4 or experimental::precision::s4 as the fragment data type. Since the elements of the fragment are packed together, num\_storage\_elements will be smaller than num\_elements for that fragment. This is true for single bit precision as well, in which case, the mapping from element\_type<T> to storage\_element\_type<T> is as follows: experimental::precision::u4 -> unsigned (8 elements in 1 storage element) experimental::precision::s4 -> int (8 elements in 1 storage element) experimental::precision::b1 -> unsigned (32 elements in 1 storage element) all other types T -> T The allowed layouts for sub-byte fragments is always row\_major for matrix\_a and col\_major for matrix\_b. bmma\_sync Waits until all warp lanes have executed bmma\_sync, and then performs the warp- synchronous bit matrix multiply-accumulate operation D = (A op B) + C, where op consists of a logical operation bmmaBitOp followed by the accumulation defined by bmmaAccumulateOp. The only available op is a bmmaBitOpXOR, a 128- bit XOR of a row in matrix\_a with the 128-bit column of matrix\_b, followed by a bmmaAccumulateOpPOPC which counts the number of set bits. B.17.3. Restrictions The special format required by tensor cores may be different for each major and minor device architecture. This is further complicated by threads holding only a fragment (opaque architecture-specific ABI data structure) of the overall matrix, with the developer not allowed to make assumptions on how the individual parameters are mapped to the registers participating in the matrix multiply-accumulate. Since fragments are architecture-specific, it is unsafe to pass them from function A to function B if the functions have been compiled for different link-compatible architectures and linked together into the same device executable. In this case, the size www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 167 ----------------------- Page 184----------------------- C++ Language Extensions and layout of the fragment will be specific to one architecture and using WMMA APIs in the other will lead to incorrect results or potentially, corruption. An example of two link-compatible architectures, where the layout of the fragment differs, is sm\_70 and sm\_75. fragA.cu: void foo() { wmma::fragment<...> mat\_a; bar(&mat\_a); } fragB.cu: void bar(wmma::fragment<...> \*mat\_a) { // operate on mat\_a } // sm\_70 fragment layout $> nvcc -dc -arch=compute\_70 -code=sm\_70 fragA.cu -o fragA.o // sm\_75 fragment layout $> nvcc -dc -arch=compute\_75 -code=sm\_75 fragB.cu -o fragB.o // Linking the two together $> nvcc -dlink -arch=sm\_75 fragA.o fragB.o -o frag.o This undefined behavior might also be undetectable at compilation time and by tools at runtime, so extra care is needed to make sure the layout of the fragments is consistent. This linking hazard is most likely to appear when linking with a legacy library that is both built for a different link-compatible architecture and expecting to be passed a WMMA fragment. Note that in the case of weak linkages (for example, a CUDA C++ inline function), the linker may choose any available function definition which may result in implicit passes between compilation units. To avoid these sorts of problems, the matrix should always be stored out to memory for transit through external interfaces (e.g. wmma::store\_matrix\_sync(dst, …);) and then it can be safely passed to bar() as a pointer type [e.g. float \*dst]. Note that since sm\_70 can run on sm\_75, the above example sm\_75 code can be changed to sm\_70 and correctly work on sm\_75. However, it is recommended to have sm\_75 native code in your application when linking with other sm\_75 separately compiled binaries. B.17.4. Element Types & Matrix Sizes Tensor Cores support a variety of element types and matrix sizes. The following table presents the various combinations of matrix\_a, matrix\_b and accumulator matrix supported: Matrix A Matrix B Accumulator Matrix Size (m-n-k) \_\_half \_\_half float 16x16x16 \_\_half \_\_half float 32x8x16 \_\_half \_\_half float 8x32x16 \_\_half \_\_half \_\_half 16x16x16 www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 168 ----------------------- Page 185----------------------- C++ Language Extensions Matrix A Matrix B Accumulator Matrix Size (m-n-k) \_\_half \_\_half \_\_half 32x8x16 \_\_half \_\_half \_\_half 8x32x16 unsigned char unsigned char int 16x16x16 unsigned char unsigned char int 32x8x16 unsigned char unsigned char int 8x32x16 signed char signed char int 16x16x16 signed char signed char int 32x8x16 signed char signed char int 8x32x16 In addition, Tensor Cores have experimental support for the following sub-byte operations: Matrix A Matrix B Accumulator Matrix Size (m-n-k) precision::u4 precision::u4 int 8x8x32 precision::s4 precision::s4 int 8x8x32 precision::b1 precision::b1 int 8x8x128 B.17.5. Example The following code implements a 16x16x16 matrix multiplication in a single warp. #include <mma.h> using namespace nvcuda; \_\_global\_\_ void wmma\_ker(half \*a, half \*b, float \*c) { // Declare the fragments wmma::fragment<wmma::matrix\_a, 16, 16, 16, half, wmma::col\_major> a\_frag; wmma::fragment<wmma::matrix\_b, 16, 16, 16, half, wmma::row\_major> b\_frag; wmma::fragment<wmma::accumulator, 16, 16, 16, float> c\_frag; // Initialize the output to zero wmma::fill\_fragment(c\_frag, 0.0f); // Load the inputs wmma::load\_matrix\_sync(a\_frag, a, 16); wmma::load\_matrix\_sync(b\_frag, b, 16); // Perform the matrix multiplication wmma::mma\_sync(c\_frag, a\_frag, b\_frag, c\_frag); // Store the output wmma::store\_matrix\_sync(c, c\_frag, 16, wmma::mem\_row\_major); } www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 169 ----------------------- Page 186----------------------- C++ Language Extensions B.18. Profiler Counter Function Each multiprocessor has a set of sixteen hardware counters that an application can increment with a single instruction by calling the \_\_prof\_trigger() function. void \_\_prof\_trigger(int counter); increments by one per warp the per-multiprocessor hardware counter of index counter. Counters 8 to 15 are reserved and should not be used by applications. The value of counters 0, 1, ..., 7 can be obtained via nvprof by nvprof --events prof\_trigger\_0x where x is 0, 1, ..., 7. All counters are reset before each kernel launch (note that when collecting counters, kernel launches are synchronous as mentioned in Concurrent Execution between Host and Device). B.19. Assertion Assertion is only supported by devices of compute capability 2.x and higher. It is not supported on MacOS, regardless of the device, and loading a module that references the assert function on Mac OS will fail. void assert(int expression); stops the kernel execution if expression is equal to zero. If the program is run within a debugger, this triggers a breakpoint and the debugger can be used to inspect the current state of the device. Otherwise, each thread for which expression is equal to zero prints a message to stderr after synchronization with the host via cudaDeviceSynchronize(), cudaStreamSynchronize(), or cudaEventSynchronize(). The format of this message is as follows: <filename>:<line number>:<function>: block: [blockId.x,blockId.x,blockIdx .z], thread: [threadIdx.x,threadIdx.y,threadIdx.z] Assertion `<expression>` failed. Any subsequent host-side synchronization calls made for the same device will return cudaErrorAssert. No more commands can be sent to this device until cudaDeviceReset() is called to reinitialize the device. If expression is different from zero, the kernel execution is unaffected. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 170 ----------------------- Page 187----------------------- C++ Language Extensions For example, the following program from source file test.cu #include <assert.h> \_\_global\_\_ void testAssert(void) { int is\_one = 1; int should\_be\_one = 0; // This will have no effect assert(is\_one); // This will halt kernel execution assert(should\_be\_one); } int main(int argc, char\* argv[]) { testAssert<<<1,1>>> (); cudaDeviceSynchronize(); return 0; } will output: test.cu:19: void testAssert(): block: [0,0,0], thread: [0,0,0] Assertion `should\_be\_one` failed. Assertions are for debugging purposes. They can affect performance and it is therefore recommended to disable them in production code. They can be disabled at compile time by defining the NDEBUG preprocessor macro before including assert.h. Note that expression should not be an expression with side effects (something like (++i > 0), for example), otherwise disabling the assertion will affect the functionality of the code. B.20. Formatted Output Formatted output is only supported by devices of compute capability 2.x and higher. int printf(const char \*format[, arg, ...]); prints formatted output from a kernel to a host-side output stream. The in-kernel printf() function behaves in a similar way to the standard C-library printf() function, and the user is referred to the host system's manual pages for a complete description of printf() behavior. In essence, the string passed in as format is output to a stream on the host, with substitutions made from the argument list wherever a format specifier is encountered. Supported format specifiers are listed below. The printf() command is executed as any other device-side function: per-thread, and in the context of the calling thread. From a multi-threaded kernel, this means that a straightforward call to printf() will be executed by every thread, using that thread's data as specified. Multiple versions of the output string will then appear at the host stream, once for each thread which encountered the printf(). It is up to the programmer to limit the output to a single thread if only a single output string is desired (see Examples for an illustrative example). www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 171 ----------------------- Page 188----------------------- C++ Language Extensions Unlike the C-standard printf(), which returns the number of characters printed, CUDA's printf() returns the number of arguments parsed. If no arguments follow the format string, 0 is returned. If the format string is NULL, -1 is returned. If an internal error occurs, -2 is returned. B.20.1. Format Specifiers As for standard printf(), format specifiers take the form: %[flags][width] [.precision][size]type The following fields are supported (see widely-available documentation for a complete description of all behaviors): ‣ Flags: '#' ' ' '0' '+' '-' ‣ Width: '\*' '0-9' ‣ Precision: '0-9' ‣ Size: 'h' 'l' 'll' ‣ Type: "%cdiouxXpeEfgGaAs" Note that CUDA's printf()will accept any combination of flag, width, precision, size and type, whether or not overall they form a valid format specifier. In other words, "%hd" will be accepted and printf will expect a double-precision variable in the corresponding location in the argument list. B.20.2. Limitations Final formatting of the printf() output takes place on the host system. This means that the format string must be understood by the host-system's compiler and C library. Every effort has been made to ensure that the format specifiers supported by CUDA's printf function form a universal subset from the most common host compilers, but exact behavior will be host-OS-dependent. As described in Format Specifiers, printf() will accept all combinations of valid flags and types. This is because it cannot determine what will and will not be valid on the host system where the final output is formatted. The effect of this is that output may be undefined if the program emits a format string which contains invalid combinations. The printf() command can accept at most 32 arguments in addition to the format string. Additional arguments beyond this will be ignored, and the format specifier output as-is. Owing to the differing size of the long type on 64-bit Windows platforms (four bytes on 64-bit Windows platforms, eight bytes on other 64-bit platforms), a kernel which is compiled on a non-Windows 64-bit machine but then run on a win64 machine will see corrupted output for all format strings which include "%ld". It is recommended that the compilation platform matches the execution platform to ensure safety. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 172 ----------------------- Page 189----------------------- C++ Language Extensions The output buffer for printf() is set to a fixed size before kernel launch (see Associated Host-Side API). It is circular and if more output is produced during kernel execution than can fit in the buffer, older output is overwritten. It is flushed only when one of these actions is performed: ‣ Kernel launch via <<<>>> or cuLaunchKernel() (at the start of the launch, and if the CUDA\_LAUNCH\_BLOCKING environment variable is set to 1, at the end of the launch as well), ‣ Synchronization via cudaDeviceSynchronize(), cuCtxSynchronize(), cudaStreamSynchronize(), cuStreamSynchronize(), cudaEventSynchronize(), or cuEventSynchronize(), ‣ Memory copies via any blocking version of cudaMemcpy\*() or cuMemcpy\*(), ‣ Module loading/unloading via cuModuleLoad() or cuModuleUnload(), ‣ Context destruction via cudaDeviceReset() or cuCtxDestroy(). ‣ Prior to executing a stream callback added by cudaStreamAddCallback or cuStreamAddCallback. Note that the buffer is not flushed automatically when the program exits. The user must call cudaDeviceReset() or cuCtxDestroy() explicitly, as shown in the examples below. Internally printf() uses a shared data structure and so it is possible that calling printf() might change the order of execution of threads. In particular, a thread which calls printf() might take a longer execution path than one which does not call printf(), and that path length is dependent upon the parameters of the printf(). Note, however, that CUDA makes no guarantees of thread execution order except at explicit \_\_syncthreads() barriers, so it is impossible to tell whether execution order has been modified by printf() or by other scheduling behaviour in the hardware. B.20.3. Associated Host-Side API The following API functions get and set the size of the buffer used to transfer the printf() arguments and internal metadata to the host (default is 1 megabyte): ‣ cudaDeviceGetLimit(size\_t\* size,cudaLimitPrintfFifoSize) ‣ cudaDeviceSetLimit(cudaLimitPrintfFifoSize, size\_t size) www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 173 ----------------------- Page 190----------------------- C++ Language Extensions B.20.4. Examples The following code sample: #include <stdio.h> \_\_global\_\_ void helloCUDA(float f) { printf("Hello thread %d, f=%f\n", threadIdx.x, f); } int main() { helloCUDA<<<1, 5>>> (1.2345f); cudaDeviceSynchronize(); return 0; } will output: Hello thread 2, f=1.2345 Hello thread 1, f=1.2345 Hello thread 4, f=1.2345 Hello thread 0, f=1.2345 Hello thread 3, f=1.2345 Notice how each thread encounters the printf() command, so there are as many lines of output as there were threads launched in the grid. As expected, global values (i.e., float f) are common between all threads, and local values (i.e., threadIdx.x) are distinct per-thread. The following code sample: #include <stdio.h> \_\_global\_\_ void helloCUDA(float f) { if (threadIdx.x == 0) printf("Hello thread %d, f=%f\n", threadIdx.x, f) ; } int main() { helloCUDA<<<1, 5>>> (1.2345f); cudaDeviceSynchronize(); return 0; } will output: Hello thread 0, f=1.2345 Self-evidently, the if() statement limits which threads will call printf, so that only a single line of output is seen. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 174 ----------------------- Page 191----------------------- C++ Language Extensions B.21. Dynamic Global Memory Allocation and Operations Dynamic global memory allocation and operations are only supported by devices of compute capability 2.x and higher. \_\_host\_\_ \_\_device\_\_ void\* malloc(size\_t size); \_\_device\_\_ void \*\_\_nv\_aligned\_device\_malloc(size\_t size, size\_t align); \_\_host\_\_ \_\_device\_\_ void free(void\* ptr); allocate and free memory dynamically from a fixed-size heap in global memory. \_\_host\_\_ \_\_device\_\_ void\* memcpy(void\* dest, const void\* src, size\_t size); copy size bytes from the memory location pointed by src to the memory location pointed by dest. \_\_host\_\_ \_\_device\_\_ void\* memset(void\* ptr, int value, size\_t size); set size bytes of memory block pointed by ptr to value (interpreted as an unsigned char). The CUDA in-kernel malloc() function allocates at least size bytes from the device heap and returns a pointer to the allocated memory or NULL if insufficient memory exists to fulfill the request. The returned pointer is guaranteed to be aligned to a 16-byte boundary. The CUDA in-kernel \_\_nv\_aligned\_device\_malloc() function allocates at least size bytes from the device heap and returns a pointer to the allocated memory or NULL if insufficient memory exists to fulfill the requested size or alignment. The address of the allocated memory will be a multiple of align. align must be a non-zero power of 2. The CUDA in-kernel free() function deallocates the memory pointed to by ptr, which must have been returned by a previous call to malloc() or \_\_nv\_aligned\_device\_malloc(). If ptr is NULL, the call to free() is ignored. Repeated calls to free() with the same ptr has undefined behavior. The memory allocated by a given CUDA thread via malloc() or \_\_nv\_aligned\_device\_malloc() remains allocated for the lifetime of the CUDA context, or until it is explicitly released by a call to free(). It can be used by any other CUDA threads even from subsequent kernel launches. Any CUDA thread may free memory allocated by another thread, but care should be taken to ensure that the same pointer is not freed more than once. B.21.1. Heap Memory Allocation The device memory heap has a fixed size that must be specified before any program using malloc(), \_\_nv\_aligned\_device\_malloc() or free() is loaded into the www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 175 ----------------------- Page 192----------------------- C++ Language Extensions context. A default heap of eight megabytes is allocated if any program uses malloc() or \_\_nv\_aligned\_device\_malloc() without explicitly specifying the heap size. The following API functions get and set the heap size: ‣ cudaDeviceGetLimit(size\_t\* size, cudaLimitMallocHeapSize) ‣ cudaDeviceSetLimit(cudaLimitMallocHeapSize, size\_t size) The heap size granted will be at least size bytes. cuCtxGetLimit()and cudaDeviceGetLimit() return the currently requested heap size. The actual memory allocation for the heap occurs when a module is loaded into the context, either explicitly via the CUDA driver API (see Module), or implicitly via the CUDA runtime API (see CUDA Runtime). If the memory allocation fails, the module load will generate a CUDA\_ERROR\_SHARED\_OBJECT\_INIT\_FAILED error. Heap size cannot be changed once a module load has occurred and it does not resize dynamically according to need. Memory reserved for the device heap is in addition to memory allocated through host- side CUDA API calls such as cudaMalloc(). B.21.2. Interoperability with Host Memory API Memory allocated via device malloc() or \_\_nv\_aligned\_device\_malloc() cannot be freed using the runtime (i.e., by calling any of the free memory functions from Device Memory). Similarly, memory allocated via the runtime (i.e., by calling any of the memory allocation functions from Device Memory) cannot be freed via free(). In addition, memory allocated by a call to malloc() or \_\_nv\_aligned\_device\_malloc() in device code cannot be used in any runtime or driver API calls (i.e. cudaMemcpy, cudaMemset, etc). B.21.3. Examples www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 176 ----------------------- Page 193----------------------- C++ Language Extensions B.21.3.1. Per Thread Allocation The following code sample: #include <stdlib.h> #include <stdio.h> \_\_global\_\_ void mallocTest() { size\_t size = 123; char\* ptr = (char\*)malloc(size); memset(ptr, 0, size); printf("Thread %d got pointer: %p\n", threadIdx.x, ptr); free(ptr); } int main() { // Set a heap size of 128 megabytes. Note that this must // be done before any kernel is launched. cudaDeviceSetLimit(cudaLimitMallocHeapSize, 128\*1024\*1024); mallocTest<<<1, 5>>> (); cudaDeviceSynchronize(); return 0; } will output: Thread 0 got pointer: 00057020 Thread 1 got pointer: 0005708c Thread 2 got pointer: 000570f8 Thread 3 got pointer: 00057164 Thread 4 got pointer: 000571d0 Notice how each thread encounters the malloc() and memset() commands and so receives and initializes its own allocation. (Exact pointer values will vary: these are illustrative.) www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 177 ----------------------- Page 194----------------------- C++ Language Extensions B.21.3.2. Per Thread Block Allocation #include <stdlib.h> \_\_global\_\_ void mallocTest() { \_\_shared\_\_ int\* data; // The first thread in the block does the allocation and then // shares the pointer with all other threads through shared memory, // so that access can easily be coalesced. // 64 bytes per thread are allocated. if (threadIdx.x == 0) { size\_t size = blockDim .x \* 64; data = (int\*)malloc(size); } \_\_syncthreads(); // Check for failure if (data == NULL) return; // Threads index into the memory, ensuring coalescence int\* ptr = data; for (int i = 0; i < 64; ++i) ptr[i \* blockDim .x + threadIdx.x] = threadIdx.x; // Ensure all threads complete before freeing \_\_syncthreads(); // Only one thread may free the memory! if (threadIdx.x == 0) free(data); } int main() { cudaDeviceSetLimit(cudaLimitMallocHeapSize, 128\*1024\*1024); mallocTest<<<10, 128>>> (); cudaDeviceSynchronize(); return 0; } www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 178 ----------------------- Page 195----------------------- C++ Language Extensions B.21.3.3. Allocation Persisting Between Kernel Launches #include <stdlib.h> #include <stdio.h> #define NUM\_BLOCKS 20 \_\_device\_\_ int\* dataptr[NUM\_BLOCKS]; // Per-block pointer \_\_global\_\_ void allocmem() { // Only the first thread in the block does the allocation // since we want only one allocation per block. if (threadIdx.x == 0) dataptr[blockIdx .x] = (int\*)malloc(blockDim .x \* 4); \_\_syncthreads(); // Check for failure if (dataptr[blockIdx .x] == NULL) return; // Zero the data with all threads in parallel dataptr[blockIdx .x][threadIdx.x] = 0; } // Simple example: store thread ID into each element \_\_global\_\_ void usemem() { int\* ptr = dataptr[blockIdx .x]; if (ptr != NULL) ptr[threadIdx.x] += threadIdx.x; } // Print the content of the buffer before freeing it \_\_global\_\_ void freemem() { int\* ptr = dataptr[blockIdx .x]; if (ptr != NULL) printf("Block %d, Thread %d: final value = %d\n", blockIdx .x, threadIdx.x, ptr[threadIdx.x]); // Only free from one thread! if (threadIdx.x == 0) free(ptr); } int main() { cudaDeviceSetLimit(cudaLimitMallocHeapSize, 128\*1024\*1024); // Allocate memory allocmem<<< NUM\_BLOCKS, 10 >>> (); // Use memory usemem<<< NUM\_BLOCKS, 10 >>> (); usemem<<< NUM\_BLOCKS, 10 >>> (); usemem<<< NUM\_BLOCKS, 10 >>> (); // Free memory freemem<<< NUM\_BLOCKS, 10 >>> (); cudaDeviceSynchronize(); return 0; } www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 179 ----------------------- Page 196----------------------- C++ Language Extensions B.22. Execution Configuration Any call to a \_\_global\_\_ function must specify the execution configuration for that call. The execution configuration defines the dimension of the grid and blocks that will be used to execute the function on the device, as well as the associated stream (see CUDA Runtime for a description of streams). The execution configuration is specified by inserting an expression of the form <<< Dg, Db, Ns, S >>> between the function name and the parenthesized argument list, where: ‣ Dg is of type dim3 (see dim3) and specifies the dimension and size of the grid, such that Dg.x \* Dg.y \* Dg.z equals the number of blocks being launched; ‣ Db is of type dim3 (see dim3) and specifies the dimension and size of each block, such that Db.x \* Db.y \* Db.z equals the number of threads per block; ‣ Ns is of type size\_t and specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory; this dynamically allocated memory is used by any of the variables declared as an external array as mentioned in \_\_shared\_\_; Ns is an optional argument which defaults to 0; ‣ S is of type cudaStream\_t and specifies the associated stream; S is an optional argument which defaults to 0. As an example, a function declared as \_\_global\_\_ void Func(float\* parameter); must be called like this: Func<<< Dg, Db, Ns >>> (parameter); The arguments to the execution configuration are evaluated before the actual function arguments. The function call will fail if Dg or Db are greater than the maximum sizes allowed for the device as specified in Compute Capabilities, or if Ns is greater than the maximum amount of shared memory available on the device, minus the amount of shared memory required for static allocation. B.23. Launch Bounds As discussed in detail in Multiprocessor Level, the fewer registers a kernel uses, the more threads and thread blocks are likely to reside on a multiprocessor, which can improve performance. Therefore, the compiler uses heuristics to minimize register usage while keeping register spilling (see Device Memory Accesses) and instruction count to a minimum. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 180 ----------------------- Page 197----------------------- C++ Language Extensions An application can optionally aid these heuristics by providing additional information to the compiler in the form of launch bounds that are specified using the \_\_launch\_bounds\_\_() qualifier in the definition of a \_\_global\_\_ function: \_\_global\_\_ void \_\_launch\_bounds\_\_(maxThreadsPerBlock, minBlocksPerMultiprocessor) MyKernel(...) { ... } ‣ maxThreadsPerBlock specifies the maximum number of threads per block with which the application will ever launch MyKernel(); it compiles to the .maxntid PTX directive; ‣ minBlocksPerMultiprocessor is optional and specifies the desired minimum number of resident blocks per multiprocessor; it compiles to the .minnctapersm PTX directive. If launch bounds are specified, the compiler first derives from them the upper limit L on the number of registers the kernel should use to ensure that minBlocksPerMultiprocessor blocks (or a single block if minBlocksPerMultiprocessor is not specified) of maxThreadsPerBlock threads can reside on the multiprocessor (see Hardware Multithreading for the relationship between the number of registers used by a kernel and the number of registers allocated per block). The compiler then optimizes register usage in the following way: ‣ If the initial register usage is higher than L, the compiler reduces it further until it becomes less or equal to L, usually at the expense of more local memory usage and/ or higher number of instructions; ‣ If the initial register usage is lower than L ‣ If maxThreadsPerBlock is specified and minBlocksPerMultiprocessor is not, the compiler uses maxThreadsPerBlock to determine the register usage thresholds for the transitions between n and n+1 resident blocks (i.e., when using one less register makes room for an additional resident block as in the example of Multiprocessor Level) and then applies similar heuristics as when no launch bounds are specified; ‣ If both minBlocksPerMultiprocessor and maxThreadsPerBlock are specified, the compiler may increase register usage as high as L to reduce the number of instructions and better hide single thread instruction latency. A kernel will fail to launch if it is executed with more threads per block than its launch bound maxThreadsPerBlock. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 181 ----------------------- Page 198----------------------- C++ Language Extensions Optimal launch bounds for a given kernel will usually differ across major architecture revisions. The sample code below shows how this is typically handled in device code using the \_\_CUDA\_ARCH\_\_ macro introduced in Application Compatibility #define THREADS\_PER\_BLOCK 256 #if \_\_CUDA\_ARCH\_\_ >= 200 #define MY\_KERNEL\_MAX\_THREADS (2 \* THREADS\_PER\_BLOCK) #define MY\_KERNEL\_MIN\_BLOCKS 3 #else #define MY\_KERNEL\_MAX\_THREADS THREADS\_PER\_BLOCK #define MY\_KERNEL\_MIN\_BLOCKS 2 #endif // Device code \_\_global\_\_ void \_\_launch\_bounds\_\_(MY\_KERNEL\_MAX\_THREADS, MY\_KERNEL\_MIN\_BLOCKS) MyKernel(...) { ... } In the common case where MyKernel is invoked with the maximum number of threads per block (specified as the first parameter of \_\_launch\_bounds\_\_()), it is tempting to use MY\_KERNEL\_MAX\_THREADS as the number of threads per block in the execution configuration: // Host code MyKernel<<<blocksPerGrid, MY\_KERNEL\_MAX\_THREADS>>> (...); This will not work however since \_\_CUDA\_ARCH\_\_ is undefined in host code as mentioned in Application Compatibility, so MyKernel will launch with 256 threads per block even when \_\_CUDA\_ARCH\_\_ is greater or equal to 200. Instead the number of threads per block should be determined: ‣ Either at compile time using a macro that does not depend on \_\_CUDA\_ARCH\_\_, for example // Host code MyKernel<<<blocksPerGrid, THREADS\_PER\_BLOCK>>> (...); ‣ Or at runtime based on the compute capability // Host code cudaGetDeviceProperties(&deviceProp, device); int threadsPerBlock = (deviceProp.major >= 2 ? 2 \* THREADS\_PER\_BLOCK : THREADS\_PER\_BLOCK); MyKernel<<<blocksPerGrid, threadsPerBlock>>> (...); Register usage is reported by the --ptxas options=-v compiler option. The number of resident blocks can be derived from the occupancy reported by the CUDA profiler (see Device Memory Accessesfor a definition of occupancy). Register usage can also be controlled for all \_\_global\_\_ functions in a file using the maxrregcount compiler option. The value of maxrregcount is ignored for functions with launch bounds. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 182 ----------------------- Page 199----------------------- C++ Language Extensions B.24. #pragma unroll By default, the compiler unrolls small loops with a known trip count. The #pragma unroll directive however can be used to control unrolling of any given loop. It must be placed immediately before the loop and only applies to that loop. It is optionally 9 followed by an integral constant expression (ICE) . If the ICE is absent, the loop will be completely unrolled if its trip count is constant. If the ICE evaluates to 1, the compiler will not unroll the loop. The pragma will be ignored if the ICE evaluates to a non- positive integer or to an integer greater than the maximum value representable by the int data type. Examples: struct S1\_t { static const int value = 4; }; template <int X, typename T2> \_\_device\_\_ void foo(int \*p1, int \*p2) { // no argument specified, loop will be completely unrolled #pragma unroll for (int i = 0; i < 12; ++i) p1[i] += p2[i]\*2; // unroll value = 8 #pragma unroll (X+1) for (int i = 0; i < 12; ++i) p1[i] += p2[i]\*4; // unroll value = 1, loop unrolling disabled #pragma unroll 1 for (int i = 0; i < 12; ++i) p1[i] += p2[i]\*8; // unroll value = 4 #pragma unroll (T2::value) for (int i = 0; i < 12; ++i) p1[i] += p2[i]\*16; } \_\_global\_\_ void bar(int \*p1, int \*p2) { foo<7, S1\_t>(p1, p2); } B.25. SIMD Video Instructions PTX ISA version 3.0 includes SIMD (Single Instruction, Multiple Data) video instructions which operate on pairs of 16-bit values and quads of 8-bit values. These are available on devices of compute capability 3.0. The SIMD video instructions are: ‣ vadd2, vadd4 ‣ vsub2, vsub4 9 See the C++ Standard for definition of integral constant expression. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 183 ----------------------- Page 200----------------------- C++ Language Extensions ‣ vavrg2, vavrg4 ‣ vabsdiff2, vabsdiff4 ‣ vmin2, vmin4 ‣ vmax2, vmax4 ‣ vset2, vset4 PTX instructions, such as the SIMD video instructions, can be included in CUDA programs by way of the assembler, asm(), statement. The basic syntax of an asm() statement is: asm("template-string" : "constraint"(output) : "constraint"(input)")); An example of using the vabsdiff4 PTX instruction is: asm("vabsdiff4.u32.u32.u32.add" " %0, %1, %2, %3;": "=r" (result):"r" (A), "r" (B), "r" (C)); This uses the vabsdiff4 instruction to compute an integer quad byte SIMD sum of absolute differences. The absolute difference value is computed for each byte of the unsigned integers A and B in SIMD fashion. The optional accumulate operation ( .add) is specified to sum these differences. Refer to the document "Using Inline PTX Assembly in CUDA" for details on using the assembly statement in your code. Refer to the PTX ISA documentation ("Parallel Thread Execution ISA Version 3.0" for example) for details on the PTX instructions for the version of PTX that you are using. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 184 ----------------------- Page 201----------------------- Appendix C. COOPERATIVE GROUPS C.1. Introduction Cooperative Groups is an extension to the CUDA programming model, introduced in CUDA 9, for organizing groups of communicating threads. Cooperative Groups allows developers to express the granularity at which threads are communicating, helping them to express richer, more efficient parallel decompositions. Historically, the CUDA programming model has provided a single, simple construct for synchronizing cooperating threads: a barrier across all threads of a thread block, as implemented with the \_\_syncthreads() intrinsic function. However, programmers would like to define and synchronize groups of threads at other granularities to enable greater performance, design flexibility, and software reuse in the form of “collective” group-wide function interfaces. In an effort to express broader patterns of parallel interaction, many performance-oriented programmers have resorted to writing their own ad hoc and unsafe primitives for synchronizing threads within a single warp, or across sets of thread blocks running on a single GPU. Whilst the performance improvements achieved have often been valuable, this has resulted in an ever-growing collection of brittle code that is expensive to write, tune, and maintain over time and across GPU generations. Cooperative Groups addresses this by providing a safe and future-proof mechanism to enable performant code. The Cooperative Groups programming model extension describes synchronization patterns both within and across CUDA thread blocks. It provides both the means for applications to define their own groups of threads, and the interfaces to synchronize them. It also provides new launch APIs that enforce certain restrictions and therefore can guarantee the synchronization will work. These primitives enable new patterns of cooperative parallelism within CUDA, including producer-consumer parallelism, opportunistic parallelism, and global synchronization across the entire Grid. The expression of groups as first-class program objects improves software composition, since collective functions can receive an explicit object representing the group of participating threads. This object also makes programmer intent explicit, which eliminates unsound architectural assumptions that result in brittle code, undesirable www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 185 ----------------------- Page 202----------------------- Cooperative Groups restrictions upon compiler optimizations, and better compatibility with new GPU generations. The Cooperative Groups programming model consists of the following elements: ‣ data types for representing groups of cooperating threads; ‣ operations to obtain intrinsic groups defined by the CUDA launch API (e.g., thread blocks); ‣ operations for partitioning existing groups into new groups; ‣ a barrier operation to synchronize a given group; ‣ and operations to inspect the group properties as well as group-specific collectives. C.2. Intra-block Groups In this section we describe the functionality available to create groups of threads within a thread block that can synchronize and collaborate. Note that the use of Cooperative Groups for synchronization across thread blocks or devices requires some additional considerations, as described later. Cooperative Groups requires CUDA 9.0 or later. To use Cooperative Groups, include the header file: #include <cooperative\_groups.h> and use the Cooperative Groups namespace: using namespace cooperative\_groups; Then code containing any intra-block Cooperative Groups functionality can be compiled in the normal way using nvcc. C.2.1. Thread Groups and Thread Blocks Any CUDA programmer is already familiar with a certain group of threads: the thread block. The Cooperative Groups extension introduces a new datatype, thread\_block, to explicitly represent this concept within the kernel. The group can be initialized as follows: thread\_block g = this\_thread\_block(); The thread\_block datatype is derived from the more generic thread\_group datatype, which can be used to represent a wider class of groups. thread\_group provides the following functionality: void sync(); // Synchronize the threads in the group unsigned size(); // Total number of threads in the group unsigned thread\_rank(); // Rank of the calling thread within [0, size] bool is\_valid(); // Whether the group violated any APIconstraints whereas thread\_block provides the following additional block-specific functionality: dim3 group\_index(); // 3-dimensional block index within the grid dim3 thread\_index(); // 3-dimensional thread index within the block www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 186 ----------------------- Page 203----------------------- Cooperative Groups For example, if the group g is initialized as above, then g.sync(); will synchronize all threads in the block (i.e. equivalent to \_\_syncthreads();). Note that all threads in the group must participate in collective operations, or the behavior is undefined. C.2.2. Tiled Partitions The tiled\_partition() function can be used to decompose the thread block into multiple smaller groups of cooperative threads. For example, if we first create a group containing all the threads in the block: thread\_block wholeBlock = this\_thread\_block(); then we can partition this into smaller groups, each of size 32 threads: thread\_group tile32 = tiled\_partition(wholeBlock, 32); and, furthermore, we can partition each of these groups into even smaller groups, each of size 4 threads: thread\_group tile4 = tiled\_partition(tile32, 4); If, for instance, if we were to then include the following line of code: if (tile4.thread\_rank()==0) printf(“Hello from tile4 rank 0\n”); then the statement would be printed by every fourth thread in the block: the threads of rank 0 in each tile4 group, which correspond to those threads with ranks 0,4,8,12… in the wholeBlock group. Note that, currently, only supported are tile sizes which are a power of 2 and no larger than 32. C.2.3. Thread Block Tiles An alternative templated version of the tiled\_partition function is available, where a template parameter is used to specify the size of the tile: with this known at compile time there is the potential for more optimal execution. Analogous to that in the previous section, the following code will create two sets of tiled groups, of size 32 and 4 respectively: thread\_block\_tile<32> tile32 = tiled\_partition<32>(this\_thread\_block()); thread\_block\_tile<4> tile4 = tiled\_partition<4>(this\_thread\_block()); Note that the thread\_block\_tile templated data structure is being used here, and that the size of the group is passed to the tiled\_partition call as a template parameter rather than an argument. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 187 ----------------------- Page 204----------------------- Cooperative Groups Thread Block Tiles also expose additional functionality as follows: .shfl() .shfl\_down() .shfl\_up() .shfl\_xor() .any() .all() .ballot() .match\_any() .match\_all() where these cooperative synchronous operations are analogous to those described in Warp Shuffle Functions and Warp Vote Functions. However their use here, in the context of these user-defined Cooperative Groups, offers enhanced flexibility and productivity. This functionality will be demonstrated later in this appendix. As mentioned above, only supported are tile sizes which are a power of 2 and no larger than 32. C.2.4. Coalesced Groups In CUDA’s SIMT architecture, at the hardware level the multiprocessor executes threads in groups of 32 called warps. If there exists a data-dependent conditional branch in the application code such that threads within a warp diverge, then the warp serially executes each branch disabling threads not on that path. The threads that remain active on the path are referred to as coalesced. Cooperative Groups has functionality to discover, and create, a group containing all coalesced threads as follows: coalesced\_group active = coalesced\_threads(); For example, consider a situation whereby there is a branch in the code in which only the 2nd, 4th and 8th threads in each warp are active. The above call, placed in that branch, will create (for each warp) a group, active, that has three threads (with ranks 0-2 inclusive). C.2.5. Uses of Intra-block Cooperative Groups In this section, Cooperative Group functionality is illustrated through some usage examples. C.2.5.1. Discovery Pattern Commonly developers need to work with the current active set of threads. No assumption is made about the threads that are present, and instead developers work with the threads that happen to be there. This is seen in the following “aggregating www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 188 ----------------------- Page 205----------------------- Cooperative Groups atomic increment across threads in a warp” example (written using the correct CUDA 9.0 set of intrinsics): { unsigned int writemask = \_\_activemask(); unsigned int total = \_\_popc(writemask); unsigned int prefix = \_\_popc(writemask & \_\_lanemask\_lt()); // Find the lowest-numbered active lane int elected\_lane = \_\_ffs(writemask) - 1; int base\_offset = 0; if (prefix == 0) { base\_offset = atomicAdd(p, total); } base\_offset = \_\_shfl\_sync(writemask, base\_offset, elected\_lane); int thread\_offset = prefix + base\_offset; return thread\_offset; } This can be re-written with Cooperative Groups as follows: { cg::coalesced\_group g = cg::coalesced\_threads(); int prev; if (g.thread\_rank() == 0) { prev = atomicAdd(p, g.size()); } prev = g.thread\_rank() + g.shfl(prev, 0); return prev; } C.2.5.2. Warp-Synchronous Code Pattern Developers might have had warp-synchronous codes that they previously made implicit assumptions about the warp size and would code around that number. Now this needs to be specified explicitly. // If the size is known statically auto g = tiled\_partition<16>(this\_thread\_block()); // Can use g.shfl and all other warp-synchronous builtins However, the user might want to better partition his algorithm, but without needing the advantage of warp-synchronous builtins. auto g = tiled\_partition(this\_thread\_block(), 8); www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 189 ----------------------- Page 206----------------------- Cooperative Groups In this case, the group g can still synchronize and you can still build varied parallel algorithms on top, but shfl() etc. are not accessible. \_\_global\_\_ void cooperative\_kernel(...) { // obtain default "current thread block" group thread\_group my\_block = this\_thread\_block(); // subdivide into 32-thread, tiled subgroups // Tiled subgroups evenly partition a parent group into // adjacent sets of threads - in this case each one warp in size thread\_group my\_tile = tiled\_partition(my\_block, 32); // This operation will be performed by only the // first 32-thread tile of each block if (my\_block.thread\_rank() < 32) { // ... my\_tile.sync(); } } C.2.5.3. Composition Previously, there were hidden constraints on the implementation when writing certain code. Take this example: \_\_device\_\_ int sum(int \*x, int n) { // ... \_\_syncthreads(); return total; } \_\_global\_\_ void parallel\_kernel(float \*x){ // ... // Entire thread block must call sum sum(x, n); } All threads in the thread block must arrive at the \_\_syncthreads() barrier, however, this constraint is hidden from the developer who might want to use sum(…). With Cooperative Groups, a better way of writing this would be: \_\_device\_\_ int sum(const thread\_group& g, int \*x, int n) { // ... g.sync() return total; } \_\_global\_\_ void parallel\_kernel(...) { // ... // Entire thread block must call sum sum(this\_thread\_block(), x, n); // ... } C.3. Grid Synchronization Prior to the introduction of Cooperative Groups, the CUDA programming model only allowed synchronization between thread blocks at a kernel completion boundary. The www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 190 ----------------------- Page 207----------------------- Cooperative Groups kernel boundary carries with it an implicit invalidation of state, and with it, potential performance implications. For example, in certain use cases, applications have a large number of small kernels, with each kernel representing a stage in a processing pipeline. The presence of these kernels is required by the current CUDA programming model to ensure that the thread blocks operating on one pipeline stage have produced data before the thread block operating on the next pipeline stage is ready to consume it. In such cases, the ability to provide global inter thread block synchronization would allow the application to be restructured to have persistent thread blocks, which are able to synchronize on the device when a given stage is complete. To synchronize across the grid, from within a kernel, you would simply use the group: grid\_group grid = this\_grid(); and call: grid.sync(); To enable grid synchronization, when launching the kernel it is necessary to use, instead of the <<<...>>> execution configuration syntax, the cudaLaunchCooperativeKernel CUDA runtime launch API: cudaLaunchCooperativeKernel( const T \*func, dim3 gridDim, dim3 blockDim, void \*\*args, size\_t sharedMem = 0, cudaStream\_t stream = 0 ) (or the CUDA driver equivalent). To guarantee co-residency of the thread blocks on the GPU, the number of blocks launched needs to be carefully considered. For example, as many blocks as there are SMs can be launched as follows: cudaDeviceProp deviceProp; cudaGetDeviceProperties(&deviceProp, dev); // initialize, then launch cudaLaunchCooperativeKernel((void\*)my\_kernel, deviceProp.multiProcessorCount, numThreads, args); Alternatively, you can maximize the exposed parallelism by calculating how many blocks can fit simultaneously per-SM using the occupancy calculator as follows: cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocksPerSm, my\_kernel, numThreads, 0); // initialize, then launch cudaLaunchCooperativeKernel((void\*)my\_kernel, deviceProp.multiProcessorCount\*numBlocksPerSm, numThreads, args); Note also that to use grid synchronization, the device code must be compiled in separate compilation (see the "Using Separate Compilation in CUDA" section in the CUDA www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 191 ----------------------- Page 208----------------------- Cooperative Groups Compiler Driver NVCC documentation) and the device runtime linked in. The simplest example is: nvcc -arch=sm\_61 -rdc=true mytestfile.cu -o mytest You should also ensure the device supports the cooperative launch property, as can be determined by usage of the cuDeviceGetAttribute CUDA driver API : int pi=0; cuDevice dev; cuDeviceGet(&dev,0) // get handle to device 0 cuDeviceGetAttribute(&pi, CU\_DEVICE\_ATTRIBUTE\_COOPERATIVE\_LAUNCH, dev); which will set pi to 1 if the property is supported on device 0. Only devices with compute capability of 6.0 and higher are supported. In addition, you need to be running on either of these: ‣ The Linux platform without MPS, or ‣ The Linux platform with MPS and on a device with compute capability 7.0 or higher, or ‣ On current versions of Windows with the device in TCC mode. C.4. Multi-Device Synchronization In order to enable synchronization across multiple devices with Cooperative Groups, use of the cuLaunchCooperativeKernelMultiDevice CUDA API is required. This, a significant departure from existing CUDA APIs, will allow a single host thread to launch a kernel across multiple devices. In addition to the constraints and guarantees made by cuLaunchCooperativeKernel, this API has the additional semantics: ‣ This API will ensure that a launch is atomic, i.e. if the API call succeeds, then the provided number of thread blocks will launch on all specified devices. ‣ The functions launched via this API must be identical. No explicit checks are done by the driver in this regard because it is largely not feasible. It is up to the application to ensure this. ‣ No two entries in the provided launchParamsList may map to the same device. ‣ All devices being targeted by this launch must be of the same compute capability - major and minor versions. ‣ The block size, grid size and amount of shared memory per grid must be the same across all devices. Note that this means the maximum number of blocks that can be launched per device will be limited by the device with the least number of SMs. ‣ Any user defined \_\_device\_\_, \_\_constant\_\_ or \_\_managed\_\_ device global variables present in the module that owns the CUfunction being launched are independently instantiated on every device. The user is responsible for initializing such device global variables appropriately. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 192 ----------------------- Page 209----------------------- Cooperative Groups The launch parameters should be defined using an array of structs (one per device): typedef struct CUDA\_LAUNCH\_PARAMS\_st { CUfunction function; unsigned int gridDimX; unsigned int gridDimY; unsigned int gridDimZ; unsigned int blockDimX; unsigned int blockDimY; unsigned int blockDimZ; unsigned int sharedMemBytes; CUstream hStream; void \*\*kernelParams; } CUDA\_LAUNCH\_PARAMS; and passed into the launch API: cudaLaunchCooperativeKernelMultiDevice( CUDA\_LAUNCH\_PARAMS \*launchParamsList, unsigned int numDevices, unsigned int flags = 0); in a similar fashion to that for grid-wide synchronization described above. Also, as with grid-wide synchronization, the resulting device code looks very similar: multi\_grid\_group multi\_grid = this\_multi\_grid(); multi\_grid.sync(); and needs to be compiled in separate compilation. You should also ensure the device supports the cooperative multi device launch property in a similar way to that described in the previous section, but with use of CU\_DEVICE\_ATTRIBUTE\_COOPERATIVE\_MULTI\_DEVICE\_LAUNCH instead of CU\_DEVICE\_ATTRIBUTE\_COOPERATIVE\_LAUNCH. Only devices with compute capability of 6.0 and higher are supported. In addition, you need to be running on the Linux platform (without MPS) or on current versions of Windows with the device in TCC mode. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 193 ----------------------- Page 210----------------------- Appendix D. CUDA DYNAMIC PARALLELISM D.1. Introduction D.1.1. Overview Dynamic Parallelism is an extension to the CUDA programming model enabling a CUDA kernel to create and synchronize with new work directly on the GPU. The creation of parallelism dynamically at whichever point in a program that it is needed offers exciting new capabilities. The ability to create work directly from the GPU can reduce the need to transfer execution control and data between host and device, as launch configuration decisions can now be made at runtime by threads executing on the device. Additionally, data-dependent parallel work can be generated inline within a kernel at run-time, taking advantage of the GPU's hardware schedulers and load balancers dynamically and adapting in response to data-driven decisions or workloads. Algorithms and programming patterns that had previously required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism may more transparently be expressed. This document describes the extended capabilities of CUDA which enable Dynamic Parallelism, including the modifications and additions to the CUDA programming model necessary to take advantage of these, as well as guidelines and best practices for exploiting this added capacity. Dynamic Parallelism is only supported by devices of compute capability 3.5 and higher. D.1.2. Glossary Definitions for terms used in this guide. Grid A Grid is a collection of Threads. Threads in a Grid execute a Kernel Function and are divided into Thread Blocks. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 194 ----------------------- Page 211----------------------- CUDA Dynamic Parallelism Thread Block A Thread Block is a group of threads which execute on the same multiprocessor (SM). Threads within a Thread Block have access to shared memory and can be explicitly synchronized. Kernel Function A Kernel Function is an implicitly parallel subroutine that executes under the CUDA execution and memory model for every Thread in a Grid. Host The Host refers to the execution environment that initially invoked CUDA. Typically the thread running on a system's CPU processor. Parent A Parent Thread, Thread Block, or Grid is one that has launched new grid(s), the Child Grid(s). The Parent is not considered completed until all of its launched Child Grids have also completed. Child A Child thread, block, or grid is one that has been launched by a Parent grid. A Child grid must complete before the Parent Thread, Thread Block, or Grid are considered complete. Thread Block Scope Objects with Thread Block Scope have the lifetime of a single Thread Block. They only have defined behavior when operated on by Threads in the Thread Block that created the object and are destroyed when the Thread Block that created them is complete. Device Runtime The Device Runtime refers to the runtime system and APIs available to enable Kernel Functions to use Dynamic Parallelism. D.2. Execution Environment and Memory Model D.2.1. Execution Environment The CUDA execution model is based on primitives of threads, thread blocks, and grids, with kernel functions defining the program executed by individual threads within a thread block and grid. When a kernel function is invoked the grid's properties are described by an execution configuration, which has a special syntax in CUDA. Support for dynamic parallelism in CUDA extends the ability to configure, launch, and synchronize upon new grids to threads that are running on the device. D.2.1.1. Parent and Child Grids A device thread that configures and launches a new grid belongs to the parent grid, and the grid created by the invocation is a child grid. The invocation and completion of child grids is properly nested, meaning that the parent grid is not considered complete until all child grids created by its threads have completed. Even if the invoking threads do not explicitly synchronize on the child grids www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 195 ----------------------- Page 212----------------------- CUDA Dynamic Parallelism launched, the runtime guarantees an implicit synchronization between the parent and child. Tim e CPU Thread Grid A Launch Grid A Com plete Grid A Threads Grid A - Parent Grid B Launch Grid B Com plete Grid B - Child Grid B Threads Figure 14 Parent-Child Launch Nesting D.2.1.2. Scope of CUDA Primitives On both host and device, the CUDA runtime offers an API for launching kernels, for waiting for launched work to complete, and for tracking dependencies between launches via streams and events. On the host system, the state of launches and the CUDA primitives referencing streams and events are shared by all threads within a process; however processes execute independently and may not share CUDA objects. A similar hierarchy exists on the device: launched kernels and CUDA objects are visible to all threads in a thread block, but are independent between thread blocks. This means for example that a stream may be created by one thread and used by any other thread in the same thread block, but may not be shared with threads in any other thread block. D.2.1.3. Synchronization CUDA runtime operations from any thread, including kernel launches, are visible across a thread block. This means that an invoking thread in the parent grid may perform synchronization on the grids launched by that thread, by other threads in the thread block, or on streams created within the same thread block. Execution of a thread block is not considered complete until all launches by all threads in the block have completed. If all threads in a block exit before all child launches have completed, a synchronization operation will automatically be triggered. D.2.1.4. Streams and Events CUDA Streams and Events allow control over dependencies between grid launches: grids launched into the same stream execute in-order, and events may be used to create dependencies between streams. Streams and events created on the device serve this exact same purpose. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 196 ----------------------- Page 213----------------------- CUDA Dynamic Parallelism Streams and events created within a grid exist within thread block scope but have undefined behavior when used outside of the thread block where they were created. As described above, all work launched by a thread block is implicitly synchronized when the block exits; work launched into streams is included in this, with all dependencies resolved appropriately. The behavior of operations on a stream that has been modified outside of thread block scope is undefined. Streams and events created on the host have undefined behavior when used within any kernel, just as streams and events created by a parent grid have undefined behavior if used within a child grid. D.2.1.5. Ordering and Concurrency The ordering of kernel launches from the device runtime follows CUDA Stream ordering semantics. Within a thread block, all kernel launches into the same stream are executed in-order. With multiple threads in the same thread block launching into the same stream, the ordering within the stream is dependent on the thread scheduling within the block, which may be controlled with synchronization primitives such as \_\_syncthreads(). Note that because streams are shared by all threads within a thread block, the implicit NULL stream is also shared. If multiple threads in a thread block launch into the implicit stream, then these launches will be executed in-order. If concurrency is desired, explicit named streams should be used. Dynamic Parallelism enables concurrency to be expressed more easily within a program; however, the device runtime introduces no new concurrency guarantees within the CUDA execution model. There is no guarantee of concurrent execution between any number of different thread blocks on a device. The lack of concurrency guarantee extends to parent thread blocks and their child grids. When a parent thread block launches a child grid, the child is not guaranteed to begin execution until the parent thread block reaches an explicit synchronization point (e.g. cudaDeviceSynchronize()). While concurrency will often easily be achieved, it may vary as a function of deviceconfiguration, application workload, and runtime scheduling. It is therefore unsafe to depend upon any concurrency between different thread blocks. D.2.1.6. Device Management There is no multi-GPU support from the device runtime; the device runtime is only capable of operating on the device upon which it is currently executing. It is permitted, however, to query properties for any CUDA capable device in the system. D.2.2. Memory Model Parent and child grids share the same global and constant memory storage, but have distinct local and shared memory. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 197 ----------------------- Page 214----------------------- CUDA Dynamic Parallelism D.2.2.1. Coherence and Consistency D.2.2.1.1. Global Memory Parent and child grids have coherent access to global memory, with weak consistency guarantees between child and parent. There are two points in the execution of a child grid when its view of memory is fully consistent with the parent thread: when the child grid is invoked by the parent, and when the child grid completes as signaled by a synchronization API invocation in the parent thread. All global memory operations in the parent thread prior to the child grid's invocation are visible to the child grid. All memory operations of the child grid are visible to the parent after the parent has synchronized on the child grid's completion. In the following example, the child grid executing child\_launch is only guaranteed to see the modifications to data made before the child grid was launched. Since thread 0 of the parent is performing the launch, the child will be consistent with the memory seen by thread 0 of the parent. Due to the first \_\_syncthreads() call, the child will see data[0]=0, data[1]=1, ..., data[255]=255 (without the \_\_syncthreads() call, only data[0] would be guaranteed to be seen by the child). When the child grid returns, thread 0 is guaranteed to see modifications made by the threads in its child grid. Those modifications become available to the other threads of the parent grid only after the second \_\_syncthreads() call: \_\_global\_\_ void child\_launch(int \*data) { data[threadIdx.x] = data[threadIdx.x]+1; } \_\_global\_\_ void parent\_launch(int \*data) { data[threadIdx.x] = threadIdx.x; \_\_syncthreads(); if (threadIdx.x == 0) { child\_launch<<< 1, 256 >>> (data); cudaDeviceSynchronize(); } \_\_syncthreads(); } void host\_launch(int \*data) { parent\_launch<<< 1, 256 >>> (data); } D.2.2.1.2. Zero Copy Memory Zero-copy system memory has identical coherence and consistency guarantees to global memory, and follows the semantics detailed above. A kernel may not allocate or free zero-copy memory, but may use pointers to zero-copy passed in from the host program. D.2.2.1.3. Constant Memory Constants are immutable and may not be modified from the device, even between parent and child launches. That is to say, the value of all \_\_constant\_\_ variables must www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 198 ----------------------- Page 215----------------------- CUDA Dynamic Parallelism be set from the host prior to launch. Constant memory is inherited automatically by all child kernels from their respective parents. Taking the address of a constant memory object from within a kernel thread has the same semantics as for all CUDA programs, and passing that pointer from parent to child or from a child to parent is naturally supported. D.2.2.1.4. Shared and Local Memory Shared and Local memory is private to a thread block or thread, respectively, and is not visible or coherent between parent and child. Behavior is undefined when an object in one of these locations is referenced outside of the scope within which it belongs, and may cause an error. The NVIDIA compiler will attempt to warn if it can detect that a pointer to local or shared memory is being passed as an argument to a kernel launch. At runtime, the programmer may use the \_\_isGlobal() intrinsic to determine whether a pointer references global memory and so may safely be passed to a child launch. Note that calls to cudaMemcpy\*Async() or cudaMemset\*Async() may invoke new child kernels on the device in order to preserve stream semantics. As such, passing shared or local memory pointers to these APIs is illegal and will return an error. D.2.2.1.5. Local Memory Local memory is private storage for an executing thread, and is not visible outside of that thread. It is illegal to pass a pointer to local memory as a launch argument when launching a child kernel. The result of dereferencing such a local memory pointer from a child will be undefined. For example the following is illegal, with undefined behavior if x\_array is accessed by child\_launch: int x\_array[10]; // Creates x\_array in parent's local memory child\_launch<<< 1, 1 >>> (x\_array); It is sometimes difficult for a programmer to be aware of when a variable is placed into local memory by the compiler. As a general rule, all storage passed to a child kernel should be allocated explicitly from the global-memory heap, either with cudaMalloc(), new() or by declaring \_\_device\_\_ storage at global scope. For example: // Correct - "value" is global storage \_\_device\_\_ int value; \_\_device\_\_ void x() { value = 5; child<<< 1, 1 >>> (&value); } // Invalid - "value" is local storage \_\_device\_\_ void y() { int value = 5; child<<< 1, 1 >>> (&value); } D.2.2.1.6. Texture Memory Writes to the global memory region over which a texture is mapped are incoherent with respect to texture accesses. Coherence for texture memory is enforced at the invocation www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 199 ----------------------- Page 216----------------------- CUDA Dynamic Parallelism of a child grid and when a child grid completes. This means that writes to memory prior to a child kernel launch are reflected in texture memory accesses of the child. Similarly, writes to memory by a child will be reflected in the texture memory accesses by a parent, but only after the parent synchronizes on the child's completion. Concurrent accesses by parent and child may result in inconsistent data. D.3. Programming Interface D.3.1. CUDA C++ Reference This section describes changes and additions to the CUDA C++ language extensions for supporting Dynamic Parallelism. The language interface and API available to CUDA kernels using CUDA C++ for Dynamic Parallelism, referred to as the Device Runtime, is substantially like that of the CUDA Runtime API available on the host. Where possible the syntax and semantics of the CUDA Runtime API have been retained in order to facilitate ease of code reuse for routines that may run in either the host or device environments. As with all code in CUDA C++, the APIs and code outlined here is per-thread code. This enables each thread to make unique, dynamic decisions regarding what kernel or operation to execute next. There are no synchronization requirements between threads within a block to execute any of the provided device runtime APIs, which enables the device runtime API functions to be called in arbitrarily divergent kernel code without deadlock. D.3.1.1. Device-Side Kernel Launch Kernels may be launched from the device using the standard CUDA <<< >>> syntax: kernel\_name<<< Dg, Db, Ns, S >>> ([kernel arguments]); ‣ Dg is of type dim3 and specifies the dimensions and size of the grid ‣ Db is of type dim3 and specifies the dimensions and size of each thread block ‣ Ns is of type size\_t and specifies the number of bytes of shared memory that is dynamically allocated per thread block for this call and addition to statically allocated memory. Ns is an optional argument that defaults to 0. ‣ S is of type cudaStream\_t and specifies the stream associated with this call. The stream must have been allocated in the same thread block where the call is being made. S is an optional argument that defaults to 0. D.3.1.1.1. Launches are Asynchronous Identical to host-side launches, all device-side kernel launches are asynchronous with respect to the launching thread. That is to say, the <<<>>> launch command will return immediately and the launching thread will continue to execute until it hits an explicit launch-synchronization point such as cudaDeviceSynchronize(). The grid launch is posted to the device and will execute independently of the parent thread. The child grid www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 200 ----------------------- Page 217----------------------- CUDA Dynamic Parallelism may begin execution at any time after launch, but is not guaranteed to begin execution until the launching thread reaches an explicit launch-synchronization point. D.3.1.1.2. Launch Environment Configuration All global device configuration settings (e.g., shared memory and L1 cache size as returned from cudaDeviceGetCacheConfig(), and device limits returned from cudaDeviceGetLimit()) will be inherited from the parent. Likewise, device limits such as stack size will remain as-configured. For host-launched kernels, per-kernel configurations set from the host will take precedence over the global setting. These configurations will be used when the kernel is launched from the device as well. It is not possible to reconfigure a kernel's environment from the device. D.3.1.2. Streams Both named and unnamed (NULL) streams are available from the device runtime. Named streams may be used by any thread within a thread-block, but stream handles may not be passed to other blocks or child/parent kernels. In other words, a stream should be treated as private to the block in which it is created. Stream handles are not guaranteed to be unique between blocks, so using a stream handle within a block that did not allocate it will result in undefined behavior. Similar to host-side launch, work launched into separate streams may run concurrently, but actual concurrency is not guaranteed. Programs that depend upon concurrency between child kernels are not supported by the CUDA programming model and will have undefined behavior. The host-side NULL stream's cross-stream barrier semantic is not supported on the device (see below for details). In order to retain semantic compatibility with the host runtime, all device streams must be created using the cudaStreamCreateWithFlags() API, passing the cudaStreamNonBlocking flag. The cudaStreamCreate() call is a host-runtime- only API and will fail to compile for the device. As cudaStreamSynchronize() and cudaStreamQuery() are unsupported by the device runtime, cudaDeviceSynchronize() should be used instead when the application needs to know that stream-launched child kernels have completed. D.3.1.2.1. The Implicit (NULL) Stream Within a host program, the unnamed (NULL) stream has additional barrier synchronization semantics with other streams (see Default Stream for details). The device runtime offers a single implicit, unnamed stream shared between all threads in a block, but as all named streams must be created with the cudaStreamNonBlocking flag, work launched into the NULL stream will not insert an implicit dependency on pending work in any other streams (including NULL streams of other thread blocks). D.3.1.3. Events Only the inter-stream synchronization capabilities of CUDA events are supported. This means that cudaStreamWaitEvent() is supported, but www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 201 ----------------------- Page 218----------------------- CUDA Dynamic Parallelism cudaEventSynchronize(), cudaEventElapsedTime(), and cudaEventQuery() are not. As cudaEventElapsedTime() is not supported, cudaEvents must be created via cudaEventCreateWithFlags(), passing the cudaEventDisableTiming flag. As for all device runtime objects, event objects may be shared between all threads withinthe thread-block which created them but are local to that block and may not be passed to other kernels, or between blocks within the same kernel. Event handles are not guaranteed to be unique between blocks, so using an event handle within a block that did not create it will result in undefined behavior. D.3.1.4. Synchronization The cudaDeviceSynchronize() function will synchronize on all work launched by any thread in the thread-block up to the point where cudaDeviceSynchronize() was called. Note that cudaDeviceSynchronize() may be called from within divergent code (see Block Wide Synchronization). It is up to the program to perform sufficient additional inter-thread synchronization, for example via a call to \_\_syncthreads(), if the calling thread is intended to synchronize with child grids invoked from other threads. D.3.1.4.1. Block Wide Synchronization The cudaDeviceSynchronize() function does not imply intra-block synchronization. In particular, without explicit synchronization via a \_\_syncthreads() directive the calling thread can make no assumptions about what work has been launched by any thread other than itself. For example if multiple threads within a block are each launching work and synchronization is desired for all this work at once (perhaps because of event-based dependencies), it is up to the program to guarantee that this work is submitted by all threads before calling cudaDeviceSynchronize(). Because the implementation is permitted to synchronize on launches from any thread in the block, it is quite possible that simultaneous calls to cudaDeviceSynchronize() by multiple threads will drain all work in the first call and then have no effect for the later calls. D.3.1.5. Device Management Only the device on which a kernel is running will be controllable from that kernel. This means that device APIs such as cudaSetDevice() are not supported by the device runtime. The active device as seen from the GPU (returned from cudaGetDevice()) will have the same device number as seen from the host system. The cudaDeviceGetAttribute() call may request information about another device as this API allows specification of a device ID as a parameter of the call. Note that the catch-all cudaGetDeviceProperties() API is not offered by the device runtime - properties must be queried individually. D.3.1.6. Memory Declarations www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 202 ----------------------- Page 219----------------------- CUDA Dynamic Parallelism D.3.1.6.1. Device and Constant Memory Memory declared at file scope with \_\_device\_\_ or \_\_constant\_\_ memory space specifiers behaves identically when using the device runtime. All kernels may read or write device variables, whether the kernel was initially launched by the host or device runtime. Equivalently, all kernels will have the same view of \_\_constant\_\_s as declared at the module scope. D.3.1.6.2. Textures & Surfaces 1 CUDA supports dynamically created texture and surface objects , where a texture reference may be created on the host, passed to a kernel, used by that kernel, and then destroyed from the host. The device runtime does not allow creation or destruction of texture or surface objects from within device code, but texture and surface objects created from the host may be used and passed around freely on the device. Regardless of where they are created, dynamically created texture objects are always valid and may be passed to child kernels from a parent. The device runtime does not support legacy module-scope (i.e., Fermi-style) textures and surfaces within a kernel launched from the device. Module-scope (legacy) textures may be created from the host and used in device code as for any kernel, but may only be used by a top-level kernel (i.e., the one which is launched from the host). D.3.1.6.3. Shared Memory Variable Declarations In CUDA C++ shared memory can be declared either as a statically sized file-scope or function-scoped variable, or as an extern variable with the size determined at runtime by the kernel's caller via a launch configuration argument. Both types of declarations are valid under the device runtime. 1 Dynamically created texture and surface objects are an addition to the CUDA memory model introduced with CUDA 5.0. Please see the CUDA Programming Guide for details. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 203 ----------------------- Page 220----------------------- CUDA Dynamic Parallelism \_\_global\_\_ void permute(int n, int \*data) { extern \_\_shared\_\_ int smem[]; if (n <= 1) return; smem[threadIdx.x] = data[threadIdx.x]; \_\_syncthreads(); permute\_data(smem, n); \_\_syncthreads(); // Write back to GMEM since we can't pass SMEM to children. data[threadIdx.x] = smem[threadIdx.x]; \_\_syncthreads(); if (threadIdx.x == 0) { permute<<< 1, 256, n/2\*sizeof(int) >>> (n/2, data); permute<<< 1, 256, n/2\*sizeof(int) >>> (n/2, data+n/2); } } void host\_launch(int \*data) { permute<<< 1, 256, 256\*sizeof(int) >>> (256, data); } D.3.1.6.4. Symbol Addresses Device-side symbols (i.e., those marked \_\_device\_\_) may be referenced from within a kernel simply via the & operator, as all global-scope device variables are in the kernel's visible address space. This also applies to \_\_constant\_\_ symbols, although in this case the pointer will reference read-only data. Given that device-side symbols can be referenced directly, those CUDA runtime APIs which reference symbols (e.g., cudaMemcpyToSymbol() or cudaGetSymbolAddress()) are redundant and hence not supported by the device runtime. Note this implies that constant data cannot be altered from within a running kernel, even ahead of a child kernel launch, as references to \_\_constant\_\_ space are read-only. D.3.1.7. API Errors and Launch Failures As usual for the CUDA runtime, any function may return an error code. The last error code returned is recorded and may be retrieved via the cudaGetLastError() call. Errors are recorded per-thread, so that each thread can identify the most recent error that it has generated. The error code is of type cudaError\_t. Similar to a host-side launch, device-side launches may fail for many reasons (invalid arguments, etc). The user must call cudaGetLastError() to determine if a launch generated an error, however lack of an error after launch does not imply the child kernel completed successfully. For device-side exceptions, e.g., access to an invalid address, an error in a child grid will be returned to the host instead of being returned by the parent's call to cudaDeviceSynchronize(). www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 204 ----------------------- Page 221----------------------- CUDA Dynamic Parallelism D.3.1.7.1. Launch Setup APIs Kernel launch is a system-level mechanism exposed through the device runtime library, and as such is available directly from PTX via the underlying cudaGetParameterBuffer() and cudaLaunchDevice() APIs. It is permitted for a CUDA application to call these APIs itself, with the same requirements as for PTX. In both cases, the user is then responsible for correctly populating all necessary data structures in the correct format according to specification. Backwards compatibility is guaranteed in these data structures. As with host-side launch, the device-side operator <<<>>> maps to underlying kernel launch APIs. This is so that users targeting PTX will be able to enact a launch, and so that the compiler front-end can translate <<<>>> into these calls. Table 5 New Device-only Launch Implementation Functions Description of Difference From Host Runtime Behaviour (behaviour is identical if Runtime API Launch Functions no description) cudaGetParameterBuffer Generated automatically from <<<>>>. Note different API to host equivalent. cudaLaunchDevice Generated automatically from <<<>>>. Note different API to host equivalent. The APIs for these launch functions are different to those of the CUDA Runtime API, and are defined as follows: extern device cudaError\_t cudaGetParameterBuffer(void \*\*params); extern \_\_device\_\_ cudaError\_t cudaLaunchDevice(void \*kernel, void \*params, dim3 gridDim, dim3 blockDim, unsigned int sharedMemSize = 0, cudaStream\_t stream = 0); D.3.1.8. API Reference The portions of the CUDA Runtime API supported in the device runtime are detailed here. Host and device runtime APIs have identical syntax; semantics are the same except where indicated. The table below provides an overview of the API relative to the version available from the host. Table 6 Supported API Functions Runtime API Functions Details cudaDeviceSynchronize Synchronizes on work launched from thread's own block only cudaDeviceGetCacheConfig cudaDeviceGetLimit cudaGetLastError Last error is per-thread state, not per-block state www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 205 ----------------------- Page 222----------------------- CUDA Dynamic Parallelism Runtime API Functions Details cudaPeekAtLastError cudaGetErrorString cudaGetDeviceCount cudaDeviceGetAttribute Will return attributes for any device cudaGetDevice Always returns current device ID as would be seen from host cudaStreamCreateWithFlags Must pass cudaStreamNonBlocking flag cudaStreamDestroy cudaStreamWaitEvent cudaEventCreateWithFlags Must pass cudaEventDisableTiming flag cudaEventRecord cudaEventDestroy cudaFuncGetAttributes cudaMemcpyAsync Notes about all memcpy/memset functions: cudaMemcpy2DAsync ‣ Only async memcpy/set functions are supported cudaMemcpy3DAsync ‣ Only device-to-device memcpy is permitted cudaMemsetAsync ‣ May not pass in local or shared memory pointers cudaMemset2DAsync cudaMemset3DAsync cudaRuntimeGetVersion cudaMalloc May not call cudaFree on the device on a pointer created on the host, and vice-versa cudaFree cudaOccupancyMaxActiveBlocksPerMultiprocessor cudaOccupancyMaxPotentialBlockSize cudaOccupancyMaxPotentialBlockSizeVariableSMem D.3.2. Device-side Launch from PTX This section is for the programming language and compiler implementers who target Parallel Thread Execution (PTX) and plan to support Dynamic Parallelism in their language. It provides the low-level details related to supporting kernel launches at the PTX level. D.3.2.1. Kernel Launch APIs Device-side kernel launches can be implemented using the following two APIs accessible from PTX: cudaLaunchDevice() and cudaGetParameterBuffer(). cudaLaunchDevice() launches the specified kernel with the parameter buffer that is obtained by calling cudaGetParameterBuffer() and filled with the parameters www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 206 ----------------------- Page 223----------------------- CUDA Dynamic Parallelism to the launched kernel. The parameter buffer can be NULL, i.e., no need to invoke cudaGetParameterBuffer(), if the launched kernel does not take any parameters. D.3.2.1.1. cudaLaunchDevice At the PTX level, cudaLaunchDevice()needs to be declared in one of the two forms shown below before it is used. // PTX-level Declaration of cudaLaunchDevice() when .address\_size is 64 .extern .func(.param .b32 func\_retval0) cudaLaunchDevice ( .param .b64 func, .param .b64 parameterBuffer, .param .align 4 .b8 gridDimension[12], .param .align 4 .b8 blockDimension[12], .param .b32 sharedMemSize, .param .b64 stream ) ; // PTX-level Declaration of cudaLaunchDevice() when .address\_size is 32 .extern .func(.param .b32 func\_retval0) cudaLaunchDevice ( .param .b32 func, .param .b32 parameterBuffer, .param .align 4 .b8 gridDimension[12], .param .align 4 .b8 blockDimension[12], .param .b32 sharedMemSize, .param .b32 stream ) ; The CUDA-level declaration below is mapped to one of the aforementioned PTX-level declarations and is found in the system header file cuda\_device\_runtime\_api.h. The function is defined in the cudadevrt system library, which must be linked with a program in order to use device-side kernel launch functionality. // CUDA-level declaration of cudaLaunchDevice() extern "C" \_\_device\_\_ cudaError\_t cudaLaunchDevice(void \*func, void \*parameterBuffer, dim3 gridDimension, dim3 blockDimension, unsigned int sharedMemSize, cudaStream\_t stream); The first parameter is a pointer to the kernel to be is launched, and the second parameter is the parameter buffer that holds the actual parameters to the launched kernel. The layout of the parameter buffer is explained in Parameter Buffer Layout, below. Other parameters specify the launch configuration, i.e., as grid dimension, block dimension, shared memory size, and the stream associated with the launch (please refer to Execution Configuration for the detailed description of launch configuration. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 207 ----------------------- Page 224----------------------- CUDA Dynamic Parallelism D.3.2.1.2. cudaGetParameterBuffer cudaGetParameterBuffer() needs to be declared at the PTX level before it's used. The PTX-level declaration must be in one of the two forms given below, depending on address size: // PTX-level Declaration of cudaGetParameterBuffer() when .address\_size is 64 // When .address\_size is 64 .extern .func(.param .b64 func\_retval0) cudaGetParameterBuffer ( .param .b64 alignment, .param .b64 size ) ; // PTX-level Declaration of cudaGetParameterBuffer() when .address\_size is 32 .extern .func(.param .b32 func\_retval0) cudaGetParameterBuffer ( .param .b32 alignment, .param .b32 size ) ; The following CUDA-level declaration of cudaGetParameterBuffer() is mapped to the aforementioned PTX-level declaration: // CUDA-level Declaration of cudaGetParameterBuffer() extern "C" \_\_device\_\_ void \*cudaGetParameterBuffer(size\_t alignment, size\_t size); The first parameter specifies the alignment requirement of the parameter buffer and the second parameter the size requirement in bytes. In the current implementation, the parameter buffer returned by cudaGetParameterBuffer() is always guaranteed to be 64- byte aligned, and the alignment requirement parameter is ignored. However, it is recommended to pass the correct alignment requirement value - which is the largest alignment of any parameter to be placed in the parameter buffer - to cudaGetParameterBuffer() to ensure portability in the future. D.3.2.2. Parameter Buffer Layout Parameter reordering in the parameter buffer is prohibited, and each individual parameter placed in the parameter buffer is required to be aligned. That is, each th parameter must be placed at the n byte in the parameter buffer, where n is the smallest multiple of the parameter size that is greater than the offset of the last byte taken by the preceding parameter. The maximum size of the parameter buffer is 4KB. For a more detailed description of PTX code generated by the CUDA compiler, please refer to the PTX-3.5 specification. D.3.3. Toolkit Support for Dynamic Parallelism D.3.3.1. Including Device Runtime API in CUDA Code Similar to the host-side runtime API, prototypes for the CUDA device runtime API are included automatically during program compilation. There is no need to include cuda\_device\_runtime\_api.h explicitly. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 208 ----------------------- Page 225----------------------- CUDA Dynamic Parallelism D.3.3.2. Compiling and Linking CUDA programs are automatically linked with the host runtime library when compiled with nvcc, but the device runtime is shipped as a static library which must explicitly be linked with a program which wishes to use it. The device runtime is offered as a static library (cudadevrt.lib on Windows, libcudadevrt.a under Linux and MacOS), against which a GPU application that uses the device runtime must be linked. Linking of device libraries can be accomplished through nvcc and/or nvlink. Two simple examples are shown below. A device runtime program may be compiled and linked in a single step, if all required source files can be specified from the command line: $ nvcc -arch=sm\_35 -rdc=true hello\_world.cu -o hello -lcudadevrt It is also possible to compile CUDA .cu source files first to object files, and then link these together in a two-stage process: $ nvcc -arch=sm\_35 -dc hello\_world.cu -o hello\_world.o $ nvcc -arch=sm\_35 -rdc=true hello\_world.o -o hello -lcudadevrt Please see the Using Separate Compilation section of The CUDA Driver Compiler NVCC guide for more details. D.4. Programming Guidelines D.4.1. Basics The device runtime is a functional subset of the host runtime. API level device management, kernel launching, device memcpy, stream management, and event management are exposed from the device runtime. Programming for the device runtime should be familiar to someone who already has experience with CUDA. Device runtime syntax and semantics are largely the same as that of the host API, with any exceptions detailed earlier in this document. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 209 ----------------------- Page 226----------------------- CUDA Dynamic Parallelism The following example shows a simple Hello World program incorporating dynamic parallelism: #include <stdio.h> \_\_global\_\_ void childKernel() { printf("Hello "); } \_\_global\_\_ void parentKernel() { // launch child childKernel<<<1,1>>> (); if (cudaSuccess != cudaGetLastError()) { return; } // wait for child to complete if (cudaSuccess != cudaDeviceSynchronize()) { return; } printf("World!\n"); } int main(int argc, char \*argv[]) { // launch parent parentKernel<<<1,1>>> (); if (cudaSuccess != cudaGetLastError()) { return 1; } // wait for parent to complete if (cudaSuccess != cudaDeviceSynchronize()) { return 2; } return 0; } This program may be built in a single step from the command line as follows: $ nvcc -arch=sm\_35 -rdc=true hello\_world.cu -o hello -lcudadevrt D.4.2. Performance D.4.2.1. Synchronization Synchronization by one thread may impact the performance of other threads in the same Thread Block, even when those other threads do not call cudaDeviceSynchronize() themselves. This impact will depend upon the underlying implementation. In general the implicit synchronization of child kernels done when a thread block ends is more efficient compared to calling cudaDeviceSynchronize() explicitly. It is therefore recommended to only call cudaDeviceSynchronize() if it is needed to synchronize with a child kernel before a thread block ends. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 210 ----------------------- Page 227----------------------- CUDA Dynamic Parallelism D.4.2.2. Dynamic-parallelism-enabled Kernel Overhead System software which is active when controlling dynamic launches may impose an overhead on any kernel which is running at the time, whether or not it invokes kernel launches of its own. This overhead arises from the device runtime's execution tracking and management software and may result in decreased performance for e.g., library calls when made from the device compared to from the host side. This overhead is, in general, incurred for applications that link against the device runtime library. D.4.3. Implementation Restrictions and Limitations Dynamic Parallelism guarantees all semantics described in this document, however, certain hardware and software resources are implementation-dependent and limit the scale, performance and other properties of a program which uses the device runtime. D.4.3.1. Runtime D.4.3.1.1. Memory Footprint The device runtime system software reserves memory for various management purposes, in particular one reservation which is used for saving parent-grid state during synchronization, and a second reservation for tracking pending grid launches. Configuration controls are available to reduce the size of these reservations in exchange for certain launch limitations. See Configuration Options, below, for details. The majority of reserved memory is allocated as backing-store for parent kernel state, for use when synchronizing on a child launch. Conservatively, this memory must support storing of state for the maximum number of live threads possible on the device. This means that each parent generation at which cudaDeviceSynchronize() is callable may require up to 860MB of device memory, depending on the device configuration, which will be unavailable for program use even if it is not all consumed. D.4.3.1.2. Nesting and Synchronization Depth Using the device runtime, one kernel may launch another kernel, and that kernel may launch another, and so on. Each subordinate launch is considered a new nesting level, and the total number of levels is the nesting depth of the program. The synchronization depth is defined as the deepest level at which the program will explicitly synchronize on a child launch. Typically this is one less than the nesting depth of the program, but if the program does not need to call cudaDeviceSynchronize() at all levels then the synchronization depth might be substantially different to the nesting depth. The overall maximum nesting depth is limited to 24, but practically speaking the real limit will be the amount of memory required by the system for each new level (see Memory Footprint above). Any launch which would result in a kernel at a deeper level than the maximum will fail. Note that this may also apply to cudaMemcpyAsync(), which might itself generate a kernel launch. See Configuration Options for details. By default, sufficient storage is reserved for two levels of synchronization. This maximum synchronization depth (and hence reserved storage) may be controlled by www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 211 ----------------------- Page 228----------------------- CUDA Dynamic Parallelism calling cudaDeviceSetLimit() and specifying cudaLimitDevRuntimeSyncDepth. The number of levels to be supported must be configured before the top-level kernel is launched from the host, in order to guarantee successful execution of a nested program. Calling cudaDeviceSynchronize() at a depth greater than the specified maximum synchronization depth will return an error. An optimization is permitted where the system detects that it need not reserve space for the parent's state in cases where the parent kernel never calls cudaDeviceSynchronize(). In this case, because explicit parent/child synchronization never occurs, the memory footprint required for a program will be much less than the conservative maximum. Such a program could specify a shallower maximum synchronization depth to avoid over-allocation of backing store. D.4.3.1.3. Pending Kernel Launches When a kernel is launched, all associated configuration and parameter data is tracked until the kernel completes. This data is stored within a system-managed launch pool. The launch pool is divided into a fixed-size pool and a virtualized pool with lower performance. The device runtime system software will try to track launch data in the fixed-size pool first. The virtualized pool will be used to track new launches when the fixed-size pool is full. The size of the fixed-size launch pool is configurable by calling cudaDeviceSetLimit() from the host and specifying cudaLimitDevRuntimePendingLaunchCount. D.4.3.1.4. Configuration Options Resource allocation for the device runtime system software is controlled via the cudaDeviceSetLimit() API from the host program. Limits must be set before any kernel is launched, and may not be changed while the GPU is actively running programs. The following named limits may be set: Limit Behavior cudaLimitDevRuntimeSyncDepth Sets the maximum depth at which cudaDeviceSynchronize() may be called. Launches may be performed deeper than this, but explicit synchronization deeper than this limit will return the cudaErrorLaunchMaxDepthExceeded. The default maximum sync depth is 2. cudaLimitDevRuntimePendingLaunchCount Controls the amount of memory set aside for buffering kernel launches which have not yet begun to execute, due either to unresolved dependencies or lack of execution resources. When the buffer is full, the device runtime system software will attempt to track new pending launches in a lower performance virtualized buffer. If the virtualized buffer is also full, i.e. when all available heap space is consumed, launches will not occur, www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 212 ----------------------- Page 229----------------------- CUDA Dynamic Parallelism Limit Behavior and the thread's last error will be set to cudaErrorLaunchPendingCountExceeded. The default pending launch count is 2048 launches. cudaLimitStackSize Controls the stack size in bytes of each GPU thread. The CUDA driver automatically increases the per-thread stack size for each kernel launch as needed. This size isn't reset back to the original value after each launch. To set the per-thread stack size to a different value, cudaDeviceSetLimit() can be called to set this limit. The stack will be immediately resized, and if necessary, the device will block until all preceding requested tasks are complete. cudaDeviceGetLimit() can be called to get the current per-thread stack size. D.4.3.1.5. Memory Allocation and Lifetime cudaMalloc() and cudaFree() have distinct semantics between the host and device environments. When invoked from the host, cudaMalloc() allocates a new region from unused device memory. When invoked from the device runtime these functions map to device-side malloc() and free(). This implies that within the device environment the total allocatable memory is limited to the device malloc() heap size, which may be smaller than the available unused device memory. Also, it is an error to invoke cudaFree() from the host program on a pointer which was allocated by cudaMalloc() on the device or vice-versa. cudaMalloc() on Host cudaMalloc() on Device cudaFree() on Host Supported Not Supported cudaFree() on Device Not Supported Supported Allocation limit Free device memory cudaLimitMallocHeapSize D.4.3.1.6. SM Id and Warp Id Note that in PTX %smid and %warpid are defined as volatile values. The device runtime may reschedule thread blocks onto different SMs in order to more efficiently manage resources. As such, it is unsafe to rely upon %smid or %warpid remaining unchanged across the lifetime of a thread or thread block. D.4.3.1.7. ECC Errors No notification of ECC errors is available to code within a CUDA kernel. ECC errors are reported at the host side once the entire launch tree has completed. Any ECC errors which arise during execution of a nested program will either generate an exception or continue execution (depending upon error and configuration). www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 213 ----------------------- Page 230----------------------- Appendix E. MATHEMATICAL FUNCTIONS The reference manual lists, along with their description, all the functions of the C/C++ standard library mathematical functions that are supported in device code, as well as all intrinsic functions (that are only supported in device code). This appendix provides accuracy information for some of these functions when applicable. E.1. Standard Functions The functions from this section can be used in both host and device code. This section specifies the error bounds of each function when executed on the device and also when executed on the host in the case where the host does not supply the function. The error bounds are generated from extensive but not exhaustive tests, so they are not guaranteed bounds. Single-Precision Floating-Point Functions Addition and multiplication are IEEE-compliant, so have a maximum error of 0.5 ulp. The recommended way to round a single-precision floating-point operand to an integer, with the result being a single-precision floating-point number is rintf(), not roundf(). The reason is that roundf() maps to a 4-instruction sequence on the device, whereas rintf() maps to a single instruction. truncf(), ceilf(), and floorf() each map to a single instruction as well. Table 7 Single-Precision Mathematical Standard Library Functions with Maximum ULP Error The maximum error is stated as the absolute value of the difference in ulps between a correctly rounded single-precision result and the result returned by the CUDA library function. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 214 ----------------------- Page 231----------------------- Mathematical Functions Function Maximum ulp error x+y 0 (IEEE-754 round-to-nearest-even) x\*y 0 (IEEE-754 round-to-nearest-even) x/y 0 for compute capability ≥ 2 when compiled with -prec- div=true 2 (full range), otherwise 1/x 0 for compute capability ≥ 2 when compiled with -prec- div=true 1 (full range), otherwise rsqrtf(x) 2 (full range) 1/sqrtf(x) Applies to 1/sqrtf(x) only when it is converted to rsqrtf(x) by the compiler. sqrtf(x) 0 when compiled with -prec-sqrt=true Otherwise 1 for compute capability ≥ 5.2 and 3 for older architectures cbrtf(x) 1 (full range) rcbrtf(x) 1 (full range) hypotf(x,y) 3 (full range) rhypotf(x,y) 2 (full range) norm3df(x,y,z) 3 (full range) rnorm3df(x,y,z) 2 (full range) norm4df(x,y,z,t) 3 (full range) rnorm4df(x,y,z,t) 2 (full range) normf(dim,arr) An error bound can't be provided because a fast algorithm is used with accuracy loss due to round-off rnormf(dim,arr) 3 (full range) expf(x) 2 (full range) exp2f(x) 2 (full range) exp10f(x) 2 (full range) expm1f(x) 1 (full range) www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 215 ----------------------- Page 232----------------------- Mathematical Functions Function Maximum ulp error logf(x) 1 (full range) log2f(x) 1 (full range) log10f(x) 2 (full range) log1pf(x) 1 (full range) sinf(x) 2 (full range) cosf(x) 2 (full range) tanf(x) 4 (full range) sincosf(x,sptr,cptr) 2 (full range) sinpif(x) 2 (full range) cospif(x) 2 (full range) sincospif(x,sptr,cptr) 2 (full range) asinf(x) 4 (full range) acosf(x) 3 (full range) atanf(x) 2 (full range) atan2f(y,x) 3 (full range) sinhf(x) 3 (full range) coshf(x) 2 (full range) tanhf(x) 2 (full range) asinhf(x) 3 (full range) acoshf(x) 4 (full range) atanhf(x) 3 (full range) powf(x,y) 8 (full range) erff(x) 2 (full range) erfcf(x) 4 (full range) erfinvf(x) 2 (full range) erfcinvf(x) 2 (full range) erfcxf(x) 4 (full range) normcdff(x) 5 (full range) www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 216 ----------------------- Page 233----------------------- Mathematical Functions Function Maximum ulp error normcdfinvf(x) 5 (full range) lgammaf(x) 6 (outside interval -10.001 ... -2.264; larger inside) tgammaf(x) 11 (full range) fmaf(x,y,z) 0 (full range) frexpf(x,exp) 0 (full range) ldexpf(x,exp) 0 (full range) scalbnf(x,n) 0 (full range) scalblnf(x,l) 0 (full range) logbf(x) 0 (full range) ilogbf(x) 0 (full range) j0f(x) 9 for |x| < 8 -6 otherwise, the maximum absolute error is 2.2 x 10 j1f(x) 9 for |x| < 8 -6 otherwise, the maximum absolute error is 2.2 x 10 -6 jnf(x) For n = 128, the maximum absolute error is 2.2 x 10 y0f(x) 9 for |x| < 8 -6 otherwise, the maximum absolute error is 2.2 x 10 y1f(x) 9 for |x| < 8 -6 otherwise, the maximum absolute error is 2.2 x 10 ynf(x) ceil(2 + 2.5n) for |x| < n -6 otherwise, the maximum absolute error is 2.2 x 10 cyl\_bessel\_i0f(x) 6 (full range) cyl\_bessel\_i1f(x) 6 (full range) fmodf(x,y) 0 (full range) remainderf(x,y) 0 (full range) remquof(x,y,iptr) 0 (full range) modff(x,iptr) 0 (full range) fdimf(x,y) 0 (full range) www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 217 ----------------------- Page 234----------------------- Mathematical Functions Function Maximum ulp error truncf(x) 0 (full range) roundf(x) 0 (full range) rintf(x) 0 (full range) nearbyintf(x) 0 (full range) ceilf(x) 0 (full range) floorf(x) 0 (full range) lrintf(x) 0 (full range) lroundf(x) 0 (full range) llrintf(x) 0 (full range) llroundf(x) 0 (full range) Double-Precision Floating-Point Functions The recommended way to round a double-precision floating-point operand to an integer, with the result being a double-precision floating-point number is rint(), not round(). The reason is that round() maps to a 4-instruction sequence on the device, whereas rint() maps to a single instruction. trunc(), ceil(), and floor() each map to a single instruction as well. Table 8 Double-Precision Mathematical Standard Library Functions with Maximum ULP Error The maximum error is stated as the absolute value of the difference in ulps between a correctly rounded double-precision result and the result returned by the CUDA library function. Function Maximum ulp error x+y 0 (IEEE-754 round-to-nearest-even) x\*y 0 (IEEE-754 round-to-nearest-even) x/y 0 (IEEE-754 round-to-nearest-even) 1/x 0 (IEEE-754 round-to-nearest-even) sqrt(x) 0 (IEEE-754 round-to-nearest-even) rsqrt(x) 1 (full range) cbrt(x) 1 (full range) www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 218 ----------------------- Page 235----------------------- Mathematical Functions Function Maximum ulp error rcbrt(x) 1 (full range) hypot(x,y) 2 (full range) rhypot(x,y) 1 (full range) norm3d(x,y,z) 2 (full range) rnorm3d(x,y,z) 1 (full range) norm4d(x,y,z,t) 2 (full range) rnorm4d(x,y,z,t) 1 (full range) norm(dim,arr) An error bound can't be provided because a fast algorithm is used with accuracy loss due to round-off rnorm(dim,arr) 2 (full range) exp(x) 1 (full range) exp2(x) 1 (full range) exp10(x) 1 (full range) expm1(x) 1 (full range) log(x) 1 (full range) log2(x) 1 (full range) log10(x) 1 (full range) log1p(x) 1 (full range) sin(x) 2 (full range) cos(x) 2 (full range) tan(x) 2 (full range) sincos(x,sptr,cptr) 2 (full range) sinpi(x) 2 (full range) cospi(x) 2 (full range) sincospi(x,sptr,cptr) 2 (full range) asin(x) 2 (full range) acos(x) 2 (full range) atan(x) 2 (full range) www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 219 ----------------------- Page 236----------------------- Mathematical Functions Function Maximum ulp error atan2(y,x) 2 (full range) sinh(x) 2 (full range) cosh(x) 1 (full range) tanh(x) 2 (full range) asinh(x) 2 (full range) acosh(x) 2 (full range) atanh(x) 2 (full range) pow(x,y) 2 (full range) erf(x) 2 (full range) erfc(x) 5 (full range) erfinv(x) 5 (full range) erfcinv(x) 6 (full range) erfcx(x) 4 (full range) normcdf(x) 5 (full range) normcdfinv(x) 8 (full range) lgamma(x) 4 (outside interval -11.0001 ... -2.2637; larger inside) tgamma(x) 8 (full range) fma(x,y,z) 0 (IEEE-754 round-to-nearest-even) frexp(x,exp) 0 (full range) ldexp(x,exp) 0 (full range) scalbn(x,n) 0 (full range) scalbln(x,l) 0 (full range) logb(x) 0 (full range) ilogb(x) 0 (full range) j0(x) 7 for |x| < 8 otherwise, the maximum absolute error is 5 x -12 10 j1(x) 7 for |x| < 8 www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 220 ----------------------- Page 237----------------------- Mathematical Functions Function Maximum ulp error otherwise, the maximum absolute error is 5 x -12 10 jn(x) For n = 128, the maximum absolute error is 5 x -12 10 y0(x) 7 for |x| < 8 otherwise, the maximum absolute error is 5 x -12 10 y1(x) 7 for |x| < 8 otherwise, the maximum absolute error is 5 x -12 10 yn(x) For |x| > 1.5n, the maximum absolute error is 5 -12 x 10 cyl\_bessel\_i0(x) 6 (full range) cyl\_bessel\_i1(x) 6 (full range) fmod(x,y) 0 (full range) remainder(x,y) 0 (full range) remquo(x,y,iptr) 0 (full range) modf(x,iptr) 0 (full range) fdim(x,y) 0 (full range) trunc(x) 0 (full range) round(x) 0 (full range) rint(x) 0 (full range) nearbyint(x) 0 (full range) ceil(x) 0 (full range) floor(x) 0 (full range) lrint(x) 0 (full range) lround(x) 0 (full range) llrint(x) 0 (full range) llround(x) 0 (full range) www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 221 ----------------------- Page 238----------------------- Mathematical Functions E.2. Intrinsic Functions The functions from this section can only be used in device code. Among these functions are the less accurate, but faster versions of some of the functions of Standard Functions .They have the same name prefixed with \_\_ (such as \_\_sinf(x)). They are faster as they map to fewer native instructions. The compiler has an option (-use\_fast\_math) that forces each function in Table 9 to compile to its intrinsic counterpart. In addition to reducing the accuracy of the affected functions, it may also cause some differences in special case handling. A more robust approach is to selectively replace mathematical function calls by calls to intrinsic functions only where it is merited by the performance gains and where changed properties such as reduced accuracy and different special case handling can be tolerated. Table 9 Functions Affected by -use\_fast\_math Operator/Function Device Function x/y \_\_fdividef(x,y) sinf(x) \_\_sinf(x) cosf(x) \_\_cosf(x) tanf(x) \_\_tanf(x) sincosf(x,sptr,cptr) \_\_sincosf(x,sptr,cptr) logf(x) \_\_logf(x) log2f(x) \_\_log2f(x) log10f(x) \_\_log10f(x) expf(x) \_\_expf(x) exp10f(x) \_\_exp10f(x) powf(x,y) \_\_powf(x,y) Single-Precision Floating-Point Functions \_\_fadd\_[rn,rz,ru,rd]() and \_\_fmul\_[rn,rz,ru,rd]() map to addition and multiplication operations that the compiler never merges into FMADs. By contrast, additions and multiplications generated from the '\*' and '+' operators will frequently be combined into FMADs. Functions suffixed with \_rn operate using the round to nearest even rounding mode. Functions suffixed with \_rz operate using the round towards zero rounding mode. Functions suffixed with \_ru operate using the round up (to positive infinity) rounding mode. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 222 ----------------------- Page 239----------------------- Mathematical Functions Functions suffixed with \_rd operate using the round down (to negative infinity) rounding mode. The accuracy of floating-point division varies depending on whether the code is compiled with -prec-div=false or -prec-div=true. When the code is compiled with -prec-div=false, both the regular division / operator and \_\_fdividef(x,y) have the same accuracy, but for 2126 < y < 2128, \_\_fdividef(x,y) delivers a result of zero, whereas the / operator delivers the correct result to within the accuracy stated in Table 10. Also, for 2126 < y < 2128, if x is infinity, \_\_fdividef(x,y) delivers a NaN (as a result of multiplying infinity by zero), while the / operator returns infinity. On the other hand, the / operator is IEEE-compliant when the code is compiled with -prec- div=true or without any -prec-div option at all since its default value is true. Table 10 Single-Precision Floating-Point Intrinsic Functions (Supported by the CUDA Runtime Library with Respective Error Bounds) Function Error bounds \_\_fadd\_[rn,rz,ru,rd](x,y) IEEE-compliant. \_\_fsub\_[rn,rz,ru,rd](x,y) IEEE-compliant. \_\_fmul\_[rn,rz,ru,rd](x,y) IEEE-compliant. \_\_fmaf\_[rn,rz,ru,rd](x,y,z) IEEE-compliant. \_\_frcp\_[rn,rz,ru,rd](x) IEEE-compliant. \_\_fsqrt\_[rn,rz,ru,rd](x) IEEE-compliant. \_\_frsqrt\_rn(x) IEEE-compliant. \_\_fdiv\_[rn,rz,ru,rd](x,y) IEEE-compliant. -126 126 \_\_fdividef(x,y) For y in [2 , 2 ], the maximum ulp error is 2. \_\_expf(x) The maximum ulp error is 2 + floor(abs(1.16 \* x)). \_\_exp10f(x) The maximum ulp error is 2+ floor(abs(2.95 \* x)). \_\_logf(x) For x in [0.5, 2], the maximum absolute error is -21.41 2 , otherwise, the maximum ulp error is 3. \_\_log2f(x) For x in [0.5, 2], the maximum absolute error is -22 2 , otherwise, the maximum ulp error is 2. \_\_log10f(x) For x in [0.5, 2], the maximum absolute error is -24 2 , otherwise, the maximum ulp error is 3. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 223 ----------------------- Page 240----------------------- Mathematical Functions Function Error bounds \_\_sinf(x) For x in [-π,π], the maximum absolute error is -21.41 2 , and larger otherwise. \_\_cosf(x) For x in [-π,π], the maximum absolute error is -21.19 2 , and larger otherwise. \_\_sincosf(x,sptr,cptr) Same as \_\_sinf(x) and \_\_cosf(x). \_\_tanf(x) Derived from its implementation as \_\_sinf(x) \* (1/\_\_cosf(x)). \_\_powf(x, y) Derived from its implementation as exp2f(y \* \_\_log2f(x)). Double-Precision Floating-Point Functions \_\_dadd\_rn() and \_\_dmul\_rn() map to addition and multiplication operations that the compiler never merges into FMADs. By contrast, additions and multiplications generated from the '\*' and '+' operators will frequently be combined into FMADs. Table 11 Double-Precision Floating-Point Intrinsic Functions (Supported by the CUDA Runtime Library with Respective Error Bounds) Function Error bounds \_\_dadd\_[rn,rz,ru,rd](x,y) IEEE-compliant. \_\_dsub\_[rn,rz,ru,rd](x,y) IEEE-compliant. \_\_dmul\_[rn,rz,ru,rd](x,y) IEEE-compliant. \_\_fma\_[rn,rz,ru,rd](x,y,z) IEEE-compliant. \_\_ddiv\_[rn,rz,ru,rd](x,y)(x,y) IEEE-compliant. Requires compute capability > 2. \_\_drcp\_[rn,rz,ru,rd](x) IEEE-compliant. Requires compute capability > 2. \_\_dsqrt\_[rn,rz,ru,rd](x) IEEE-compliant. Requires compute capability > 2. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 224 ----------------------- Page 241----------------------- Appendix F. C++ LANGUAGE SUPPORT As described in Compilation with NVCC, CUDA source files compiled with nvcc can include a mix of host code and device code. The CUDA frontend compiler aims to emulate the host compiler behavior with respect to C++ input code. The input source code is processed according to the C++ ISO/IEC 14882:2003, C++ ISO/IEC 14882:2011 or C ++ ISO/IEC 14882:2014 specifications, and the CUDA frontend compiler aims to emulate any host compiler divergences from the ISO specification. In addition, the supported language is extended with CUDA-specific constructs described in this document 10, and is subject to the restrictions described below. C++11 Language Features and C++14 Language Features provide support matrices for the C++11 and C++14 features, respectively. Restrictions lists the language restrictions. Polymorphic Function Wrappers and Extended Lambdas describe additional features. Code Samples gives code samples. F.1. C++11 Language Features The following table lists new language features that have been accepted into the C++11 standard. The "Proposal" column provides a link to the ISO C++ committee proposal that describes the feature, while the "Available in nvcc (device code)" column indicates the first version of nvcc that contains an implementation of this feature (if it has been implemented) for device code. Table 12 C++11 Language Features Available C++11 in nvcc Language Feature Proposal (device code) Rvalue references N2118 7.0 Rvalue references for \*this N2439 7.0 Initialization of class objects by rvalues N1610 7.0 10 e.g., the <<<...>>> syntax for launching kernels. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 225 ----------------------- Page 242----------------------- C++ Language Support Available C++11 in nvcc Language Feature Proposal (device code) Non-static data member initializers N2756 7.0 Variadic templates N2242 7.0 Extending variadic template template parameters N2555 7.0 Initializer lists N2672 7.0 Static assertions N1720 7.0 auto-typed variables N1984 7.0 Multi-declarator auto N1737 7.0 Removal of auto as a storage-class specifier N2546 7.0 New function declarator syntax N2541 7.0 Lambda expressions N2927 7.0 Declared type of an expression N2343 7.0 Incomplete return types N3276 7.0 Right angle brackets N1757 7.0 Default template arguments for function templates DR226 7.0 Solving the SFINAE problem for expressions DR339 7.0 Alias templates N2258 7.0 Extern templates N1987 7.0 Null pointer constant N2431 7.0 Strongly-typed enums N2347 7.0 N2764 Forward declarations for enums   7.0 DR1206 Standardized attribute syntax N2761 7.0 Generalized constant expressions N2235 7.0 Alignment support N2341 7.0 Conditionally-support behavior N1627 7.0 Changing undefined behavior into diagnosable errors N1727 7.0 Delegating constructors N1986 7.0 Inheriting constructors N2540 7.0 Explicit conversion operators N2437 7.0 New character types N2249 7.0 Unicode string literals N2442 7.0 Raw string literals N2442 7.0 Universal character names in literals N2170 7.0 www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 226 ----------------------- Page 243----------------------- C++ Language Support Available C++11 in nvcc Language Feature Proposal (device code) User-defined literals N2765 7.0 Standard Layout Types N2342 7.0 Defaulted functions N2346 7.0 Deleted functions N2346 7.0 Extended friend declarations N1791 7.0 N2253 Extending sizeof   7.0 DR850 Inline namespaces N2535 7.0 Unrestricted unions N2544 7.0 Local and unnamed types as template arguments N2657 7.0 Range-based for N2930 7.0 N2928 Explicit virtual overrides N3206 7.0 N3272 Minimal support for garbage collection and reachability-based leak N/A (see N2670 detection Restrictions) Allowing move constructors to throw [noexcept] N3050 7.0 Defining move special member functions N3053 7.0 Concurrency Sequence points N2239 Atomic operations N2427 Strong Compare and Exchange N2748 Bidirectional Fences N2752 Memory model N2429 Data-dependency ordering: atomics and memory model N2664 Propagating exceptions N2179 Allow atomics use in signal handlers N2547 Thread-local storage N2659 Dynamic initialization and destruction with concurrency N2660 C99 Features in C++11 \_\_func\_\_ predefined identifier N2340 7.0 C99 preprocessor N1653 7.0 long long N1811 7.0 www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 227 ----------------------- Page 244----------------------- C++ Language Support Available C++11 in nvcc Language Feature Proposal (device code) Extended integral types N1988 F.2. C++14 Language Features The following table lists new language features that have been accepted into the C++14 standard. Table 13 C++14 Language Features Available C++14 in nvcc Language Feature Proposal (device code) Tweak to certain C++ contextual conversions N3323 9.0 Binary literals N3472 9.0 Functions with deduced return type N3638 9.0 Generalized lambda capture (init-capture) N3648 9.0 Generic (polymorphic) lambda expressions N3649 9.0 Variable templates N3651 9.0 Relaxing requirements on constexpr functions N3652 9.0 Member initializers and aggregates N3653 9.0 Clarifying memory allocation N3664 Sized deallocation N3778 [[deprecated]] attribute N3760 9.0 Single-quotation-mark as a digit separator N3781 9.0 F.3. Restrictions F.3.1. Host Compiler Extensions Host compiler specific language extensions are not supported in device code. \_\_int128 and \_Complex types are only supported in host code. \_\_float128 type is only supported in host code on 64-bit x86 Linux platforms. A constant expression of \_\_float128 type may be processed by the compiler in a floating point representation with lower precision. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 228 ----------------------- Page 245----------------------- C++ Language Support F.3.2. Preprocessor Symbols F.3.2.1. \_\_CUDA\_ARCH\_\_ 1. The type signature of the following entities shall not depend on whether \_\_CUDA\_ARCH\_\_ is defined or not, or on a particular value of \_\_CUDA\_ARCH\_\_: ‣ \_\_global\_\_ functions and function templates ‣ \_\_device\_\_ and \_\_constant\_\_ variables ‣ textures and surfaces Example: #if !defined(\_\_CUDA\_ARCH\_\_) typedef int mytype; #else typedef double mytype; #endif \_\_device\_\_ mytype xxx; // error: xxx's type depends on \_\_CUDA\_ARCH\_\_ \_\_global\_\_ void foo(mytype in, // error: foo's type depends on \_\_CUDA\_ARCH\_\_ mytype \*ptr) { \*ptr = in; } 2. If a \_\_global\_\_ function template is instantiated and launched from the host, then the function template must be instantiated with the same template arguments irrespective of whether \_\_CUDA\_ARCH\_\_ is defined and regardless of the value of \_\_CUDA\_ARCH\_\_. Example: \_\_device\_\_ int result; template <typename T> \_\_global\_\_ void kern(T in) { result = in; } \_\_host\_\_ \_\_device\_\_ void foo(void) { #if !defined(\_\_CUDA\_ARCH\_\_) kern<<<1,1>>> (1); // error: "kern<int>" instantiation only // when \_\_CUDA\_ARCH\_\_ is undefined! #endif } int main(void) { foo(); cudaDeviceSynchronize(); return 0; } www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 229 ----------------------- Page 246----------------------- C++ Language Support 3. In separate compilation mode, the presence or absence of a definition of a function or variable with external linkage shall not depend on whether \_\_CUDA\_ARCH\_\_ is defined or on a particular value of \_\_CUDA\_ARCH\_\_ 11. Example: #if !defined(\_\_CUDA\_ARCH\_\_) void foo(void) { } // error: The definition of foo() // is only present when \_\_CUDA\_ARCH\_\_ // is undefined #endif 4. In separate compilation, \_\_CUDA\_ARCH\_\_ must not be used in headers such that different objects could contain different behavior. Or, it must be guaranteed that all objects will compile for the same compute\_arch. If a weak function or template function is defined in a header and its behavior depends on \_\_CUDA\_ARCH\_\_, then the instances of that function in the objects could conflict if the objects are compiled for different compute arch. For example, if an a.h contains: template<typename T> \_\_device\_\_ T\* getptr(void) { #if \_\_CUDA\_ARCH\_\_ == 200 return NULL; /\* no address \*/ #else \_\_shared\_\_ T arr[256]; return arr; #endif } Then if a.cu and b.cu both include a.h and instantiate getptr for the same type, and b.cu expects a non-NULL address, and compile with: nvcc –arch=compute\_20 –dc a.cu nvcc –arch=compute\_30 –dc b.cu nvcc –arch=sm\_30 a.o b.o At link time only one version of the getptr is used, so the behavior would depend on which version is picked. To avoid this, either a.cu and b.cu must be compiled for the same compute arch, or \_\_CUDA\_ARCH\_\_ should not be used in the shared header function. The compiler does not guarantee that a diagnostic will be generated for the unsupported uses of \_\_CUDA\_ARCH\_\_ described above. F.3.3. Qualifiers F.3.3.1. Device Memory Space Specifiers The \_\_device\_\_, \_\_shared\_\_, \_\_managed\_\_ and \_\_constant\_\_ memory space specifiers are not allowed on: ‣ class, struct, and union data members, 11 This does not apply to entities that may be defined in more than one translation unit, such as compiler generated template instantiations. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 230 ----------------------- Page 247----------------------- C++ Language Support ‣ formal parameters, ‣ non-extern variable declarations within a function that executes on the host. The \_\_device\_\_, \_\_constant\_\_ and \_\_managed\_\_ memory space specifiers are not allowed on variable declarations that are neither extern nor static within a function that executes on the device. A \_\_device\_\_, \_\_constant\_\_, \_\_managed\_\_ or \_\_shared\_\_ variable definition cannot have a class type with a non-empty constructor or a non-empty destructor. A constructor for a class type is considered empty at a point in the translation unit, if it is either a trivial constructor or it satisfies all of the following conditions: ‣ The constructor function has been defined. ‣ The constructor function has no parameters, the initializer list is empty and the function body is an empty compound statement. ‣ Its class has no virtual functions and no virtual base classes. ‣ The default constructors of all base classes of its class can be considered empty. ‣ For all the nonstatic data members of its class that are of class type (or array thereof), the default constructors can be considered empty. A destructor for a class is considered empty at a point in the translation unit, if it is either a trivial destructor or it satisfies all of the following conditions: ‣ The destructor function has been defined. ‣ The destructor function body is an empty compound statement. ‣ Its class has no virtual functions and no virtual base classes. ‣ The destructors of all base classes of its class can be considered empty. ‣ For all the nonstatic data members of its class that are of class type (or array thereof), the destructor can be considered empty. When compiling in the whole program compilation mode (see the nvcc user manual for a description of this mode), \_\_device\_\_, \_\_shared\_\_, \_\_managed\_\_ and \_\_constant\_\_ variables cannot be defined as external using the extern keyword. The only exception is for dynamically allocated \_\_shared\_\_ variables as described in \_\_shared\_\_. When compiling in the separate compilation mode (see the nvcc user manual for a description of this mode), \_\_device\_\_, \_\_shared\_\_, \_\_managed\_\_ and \_\_constant\_\_ variables can be defined as external using the extern keyword. nvlink will generate an error when it cannot find a definition for an external variable (unless it is a dynamically allocated \_\_shared\_\_ variable). F.3.3.2. \_\_managed\_\_ Memory Space Specifier Variables marked with the \_\_managed\_\_ memory space specifier ("managed" variables) have the following restrictions: ‣ The address of a managed variable is not a constant expression. ‣ A managed variable shall not have a const qualified type. ‣ A managed variable shall not have a reference type. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 231 ----------------------- Page 248----------------------- C++ Language Support ‣ The address or value of a managed variable shall not be used when the CUDA runtime may not be in a valid state, including the following cases: ‣ In static/dynamic initialization or destruction of an object with static or thread local storage duration. ‣ In code that executes after exit() has been called (e.g., a function marked with gcc's "\_\_attribute\_\_((destructor))"). ‣ In code that executes when CUDA runtime may not be initialized (e.g., a function marked with gcc's "\_\_attribute\_\_((constructor))"). ‣ A managed variable cannot be used as an unparenthesized id-expression argument to a decltype() expression. ‣ Managed variables have the same coherence and consistency behavior as specified for dynamically allocated managed memory. ‣ When a CUDA program containing managed variables is run on an execution platform with multiple GPUs, the variables are allocated only once, and not per GPU. ‣ A managed variable declaration without the extern linkage is not allowed within a function that executes on the host. ‣ A managed variable declaration without the extern or static linkage is not allowed within a function that executes on the device. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 232 ----------------------- Page 249----------------------- C++ Language Support Here are examples of legal and illegal uses of managed variables: \_\_device\_\_ \_\_managed\_\_ int xxx = 10; // OK int \*ptr = &xxx; // error: use of managed variable // (xxx) in static initialization struct S1\_t { int field; S1\_t(void) : field(xxx) { }; }; struct S2\_t { ~S2\_t(void) { xxx = 10; } }; S1\_t temp1; // error: use of managed variable // (xxx) in dynamic initialization S2\_t temp2; // error: use of managed variable // (xxx) in the destructor of // object with static storage // duration \_\_device\_\_ \_\_managed\_\_ const int yyy = 10; // error: const qualified type \_\_device\_\_ \_\_managed\_\_ int &zzz = xxx; // error: reference type template <int \*addr> struct S3\_t { }; S3\_t<&xxx> temp; // error: address of managed // variable(xxx) not a // constant expression \_\_global\_\_ void kern(int \*ptr) { assert(ptr == &xxx); // OK xxx = 20; // OK } int main(void) { int \*ptr = &xxx; // OK kern<<<1,1>>> (ptr); cudaDeviceSynchronize(); xxx++; // OK decltype(xxx) qqq; // error: managed variable(xxx) used // as unparenthized argument to // decltype decltype((xxx)) zzz = yyy; // OK } F.3.3.3. Volatile Qualifier The compiler is free to optimize reads and writes to global or shared memory (for example, by caching global reads into registers or L1 cache) as long as it respects the memory ordering semantics of memory fence functions (Memory Fence Functions) and memory visibility semantics of synchronization functions (Synchronization Functions). These optimizations can be disabled using the volatile keyword: If a variable located in global or shared memory is declared as volatile, the compiler assumes that its value can be changed or used at any time by another thread and therefore any reference to this variable compiles to an actual memory read or write instruction. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 233 ----------------------- Page 250----------------------- C++ Language Support F.3.4. Pointers Dereferencing a pointer either to global or shared memory in code that is executed on the host, or to host memory in code that is executed on the device results in an undefined behavior, most often in a segmentation fault and application termination. The address obtained by taking the address of a \_\_device\_\_, \_\_shared\_\_ or \_\_constant\_\_ variable can only be used in device code. The address of a \_\_device\_\_ or \_\_constant\_\_ variable obtained through cudaGetSymbolAddress() as described in Device Memory can only be used in host code. F.3.5. Operators F.3.5.1. Assignment Operator \_\_constant\_\_ variables can only be assigned from the host code through runtime functions (Device Memory); they cannot be assigned from the device code. \_\_shared\_\_ variables cannot have an initialization as part of their declaration. It is not allowed to assign values to any of the built-in variables defined in Built-in Variables. F.3.5.2. Address Operator It is not allowed to take the address of any of the built-in variables defined in Built-in Variables. F.3.6. Run Time Type Information (RTTI) The following RTTI-related features are supported in host code, but not in device code. ‣ typeid operator ‣ std::type\_info ‣ dynamic\_cast operator F.3.7. Exception Handling Exception handling is only supported in host code, but not in device code. Exception specification is not supported for \_\_global\_\_ functions. F.3.8. Standard Library Standard libraries are only supported in host code, but not in device code, unless specified otherwise. F.3.9. Functions www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 234 ----------------------- Page 251----------------------- C++ Language Support F.3.9.1. External Linkage A call within some device code of a function declared with the extern qualifier is only allowed if the function is defined within the same compilation unit as the device code, i.e., a single file or several files linked together with relocatable device code and nvlink. F.3.9.2. Implicitly-declared and explicitly-defaulted functions Let F denote a function that is either implicitly-declared or is explicitly-defaulted on its first declaration The execution space specifiers (\_\_host\_\_, \_\_device\_\_) for F are the union of the execution space specifiers of all the functions that invoke it (note that a \_\_global\_\_ caller will be treated as a \_\_device\_\_ caller for this analysis). For example: class Base { int x; public: \_\_host\_\_ \_\_device\_\_ Base(void) : x(10) {} }; class Derived : public Base { int y; }; class Other: public Base { int z; }; \_\_device\_\_ void foo(void) { Derived D1; Other D2; } \_\_host\_\_ void bar(void) { Other D3; } Here, the implicitly-declared constructor function "Derived::Derived" will be treated as a \_\_device\_\_ function, since it is invoked only from the \_\_device\_\_ function "foo". The implicitly-declared constructor function "Other::Other" will be treated as a \_\_host\_\_ \_\_device\_\_ function, since it is invoked both from a \_\_device\_\_ function "foo" and a \_\_host\_\_ function "bar". In addition, if F is a virtual destructor, then the execution spaces of each virtual destructor D overridden by F are added to the set of execution spaces for F, if D is either not implicitly defined or is explicitly defaulted on a declaration other than its first declaration. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 235 ----------------------- Page 252----------------------- C++ Language Support For example: struct Base1 { virtual \_\_host\_\_ \_\_device\_\_ ~Base1() { } }; struct Derived1 : Base1 { }; // implicitly-declared virtual destructor // ~Derived1 has \_\_host\_\_ \_\_device\_\_ // execution space specifiers struct Base2 { virtual \_\_device\_\_ ~Base2(); }; \_\_device\_\_ Base2::~Base2() = default; struct Derived2 : Base2 { }; // implicitly-declared virtual destructor // ~Derived2 has \_\_device\_\_ execution // space specifiers F.3.9.3. Function Parameters \_\_global\_\_ function parameters are passed to the device via constant memory and are limited to 4 KB. \_\_global\_\_ functions cannot have a variable number of arguments. F.3.9.4. Static Variables within Function Within the body of a \_\_device\_\_ or \_\_global\_\_ function, only \_\_shared\_\_ variables or variables without any device memory space specifiers may be declared with static storage class. Within the body of a \_\_device\_\_ \_\_host\_\_ function, only unannotated static variables (i.e., without device memory space specifiers) may be declared with static storage class. Unannotated function-scope static variables have the same restrictions as \_\_device\_\_ variables defined in namespace scope. They cannot have a non-empty constructor or a non-empty destructor, if they are of class type (see Device Memory Space Specifiers). www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 236 ----------------------- Page 253----------------------- C++ Language Support Examples of legal and illegal uses of function-scope static variables are shown below. struct S1\_t { int x; }; struct S2\_t { int x; \_\_device\_\_ S2\_t(void) { x = 10; } }; struct S3\_t { int x; \_\_device\_\_ S3\_t(int p) : x(p) { } }; \_\_device\_\_ void f1() { static int i1; // OK static int i2 = 11; // OK static S1\_t i3; // OK static S1\_t i4 = {22}; // OK static \_\_shared\_\_ int i5; // OK int x = 33; static int i6 = x; // error: dynamic initialization is not allowed static S1\_t i7 = {x}; // error: dynamic initialization is not allowed static S2\_t i8; // error: dynamic initialization is not allowed static S3\_t i9(44); // error: dynamic initialization is not allowed } \_\_host\_\_ \_\_device\_\_ void f2() { static int i1; // OK static \_\_shared\_\_ int i2; // error: \_\_shared\_\_ variable inside // a host function } F.3.9.5. Function Pointers The address of a \_\_global\_\_ function taken in host code cannot be used in device code (e.g. to launch the kernel). Similarly, the address of a \_\_global\_\_ function taken in device code 12 cannot be used in host code. It is not allowed to take the address of a \_\_device\_\_ function in host code. F.3.9.6. Function Recursion \_\_global\_\_ functions do not support recursion. F.3.9.7. Friend Functions A \_\_global\_\_ function or function template cannot be defined in a friend declaration. 12 supported with architectures >= sm\_35 www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 237 ----------------------- Page 254----------------------- C++ Language Support Example: struct S1\_t { friend \_\_global\_\_ void foo1(void); // OK: not a definition template<typename T> friend \_\_global\_\_ void foo2(void); // OK: not a definition friend \_\_global\_\_ void foo3(void) { } // error: definition in friend declaration template<typename T> friend \_\_global\_\_ void foo4(void) { } // error: definition in friend declaration }; F.3.9.8. Operator Function An operator function cannot be a \_\_global\_\_ function. F.3.10. Classes F.3.10.1. Data Members Static data members are not supported except for those that are also const-qualified (see Const-qualified variables). F.3.10.2. Function Members Static member functions cannot be \_\_global\_\_ functions. F.3.10.3. Virtual Functions When a function in a derived class overrides a virtual function in a base class, the execution space specifiers (i.e., \_\_host\_\_, \_\_device\_\_) on the overridden and overriding functions must match. It is not allowed to pass as an argument to a \_\_global\_\_ function an object of a class with virtual functions. If an object is created in host code, invoking a virtual function for that object in device code has undefined behavior. If an object is created in device code, invoking a virtual function for that object in host code has undefined behavior. See Windows-Specific for additional constraints when using the Microsoft host compiler. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 238 ----------------------- Page 255----------------------- C++ Language Support Example: struct S1 { virtual \_\_host\_\_ \_\_device\_\_ void foo() { } }; \_\_managed\_\_ S1 \*ptr1, \*ptr2; \_\_managed\_\_ \_\_align\_\_(16) char buf1[128]; \_\_global\_\_ void kern() { ptr1->foo(); // error: virtual function call on a object // created in host code. ptr2 = new(buf1) S1(); } int main(void) { void \*buf; cudaMallocManaged(&buf, sizeof(S1), cudaMemAttachGlobal); ptr1 = new (buf) S1(); kern<<<1,1>>> (); cudaDeviceSynchronize(); ptr2->foo(); // error: virtual function call on an object // created in device code. } F.3.10.4. Virtual Base Classes It is not allowed to pass as an argument to a \_\_global\_\_ function an object of a class derived from virtual base classes. See Windows-Specific for additional constraints when using the Microsoft host compiler. F.3.10.5. Anonymous Unions Member variables of a namespace scope anonymous union cannot be referenced in a \_\_global\_\_ or \_\_device\_\_ function. F.3.10.6. Windows-Specific The CUDA compiler follows the IA64 ABI for class layout, while the Microsoft host compiler does not. Let T denote a pointer to member type, or a class type that satisfies any of the following conditions: ‣ T has virtual functions. ‣ T has a virtual base class. ‣ T has multiple inheritance with more than one direct or indirect empty base class. ‣ All direct and indirect base classes B of T are empty and the type of the first field F of T uses B in its definition, such that B is laid out at offset 0 in the definition of F. Let C denote T or a class type that has T as a field type or as a base class type. The CUDA compiler may compute the class layout and size differently than the Microsoft host compiler for the type C. As long as the type C is used exclusively in host or device code, the program should work correctly. Passing an object of type C between host and device code has undefined behavior e.g., as an argument to a \_\_global\_\_ function or through cudaMemcpy\*() calls. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 239 ----------------------- Page 256----------------------- C++ Language Support Accessing an object of type C or any subobject in device code, or invoking a member function in device code, has undefined behavior if the object is created in host code. Accessing an object of type C or any subobject in host code, or invoking a member function in host code, has undefined behavior if the object is created in device code 13. F.3.11. Templates A type or template cannot be used in the type, non-type or template template argument of a \_\_global\_\_ function template instantiation or a \_\_device\_\_/\_\_constant\_\_ variable instantiation if either: ‣ The type or template is defined within a \_\_host\_\_ or \_\_host\_\_ \_\_device\_\_. ‣ The type or template is a class member with private or protected access and its parent class is not defined within a \_\_device\_\_ or \_\_global\_\_ function. ‣ The type is unnamed. ‣ The type is compounded from any of the types above. Example: template <typename T> \_\_global\_\_ void myKernel(void) { } class myClass { private: struct inner\_t { }; public: static void launch(void) { // error: inner\_t is used in template argument // but it is private myKernel<inner\_t><<<1,1>>> (); } }; // C++14 only template <typename T> \_\_device\_\_ T d1; template <typename T1, typename T2> \_\_device\_\_ T1 d2; void fn() { struct S1\_t { }; // error (C++14 only): S1\_t is local to the function fn d1<S1\_t> = {}; auto lam1 = [] { }; // error (C++14 only): a closure type cannot be used for // instantiating a variable template d2<int, decltype(lam1)> = 10; } F.3.12. Trigraphs and Digraphs Trigraphs are not supported on any platform. Digraphs are not supported on Windows. 13 One way to debug suspected layout mismatch of a type C is to use printf to output the values of sizeof(C) and offsetof(C, field) in host and device code. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 240 ----------------------- Page 257----------------------- C++ Language Support F.3.13. Const-qualified variables Let 'V' denote a namespace scope variable or a class static member variable that has const qualified type and does not have execution space annotations (e.g., \_\_device\_\_, \_\_constant\_\_, \_\_shared\_\_). V is considered to be a host code variable. The value of V may be directly used in device code, if ‣ V has been initialized with a constant expression before the point of use, ‣ the type of V is not volatile-qualified, and ‣ it has one of the following types: ‣ builtin floating point type except when the Microsoft compiler is used as the host compiler, ‣ builtin integral type. Device source code cannot contain a reference to V or take the address of V. Example: const int xxx = 10; struct S1\_t { static const int yyy = 20; }; extern const int zzz; const float www = 5.0; \_\_device\_\_ void foo(void) { int local1[xxx]; // OK int local2[S1\_t::yyy]; // OK int val1 = xxx; // OK int val2 = S1\_t::yyy; // OK int val3 = zzz; // error: zzz not initialized with constant // expression at the point of use. const int &val3 = xxx; // error: reference to host variable const int \*val4 = &xxx; // error: address of host variable const float val5 = www; // OK except when the Microsoft compiler is used as // the host compiler. } const int zzz = 20; F.3.14. Long Double The use of long double type is not supported in device code. F.3.15. Deprecation Annotation nvcc supports the use of deprecated attribute when using gcc, clang, xlC, icc or pgcc host compilers, and the use of deprecated declspec when using the cl.exe host compiler. It also supports the [[deprecated]] standard attribute when the C++14 dialect has been enabled. The CUDA frontend compiler will generate a deprecation diagnostic for a reference to a deprecated entity from within the body of a \_\_device\_\_, \_\_global\_\_ or \_\_host\_\_ \_\_device\_\_ function when \_\_CUDA\_ARCH\_\_ is defined (i.e., during device compilation phase). Other references to deprecated entities will be handled by the host compiler, e.g., a reference from within a \_\_host\_\_ function. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 241 ----------------------- Page 258----------------------- C++ Language Support The CUDA frontend compiler does not support the #pragma gcc diagnostic or #pragma warning mechanisms supported by various host compilers. Therefore, deprecation diagnostics generated by the CUDA frontend compiler are not affected by these pragmas, but diagnostics generated by the host compiler will be affected. The nvcc flag -Wno-deprecated-declarations can be used to suppress all deprecation warnings, and the flag -Werror=deprecated-declarations can be used to turn deprecation warnings into errors. F.3.16. C++11 Features C++11 features that are enabled by default by the host compiler are also supported by nvcc, subject to the restrictions described in this document. In addition, invoking nvcc with -std=c++11 flag turns on all C++11 features and also invokes the host preprocessor, compiler and linker with the corresponding C++11 dialect option 14. F.3.16.1. Lambda Expressions The execution space specifiers for all member functions15 of the closure class associated with a lambda expression are derived by the compiler as follows. As described in the C++11 standard, the compiler creates a closure type in the smallest block scope, class scope or namespace scope that contains the lambda expression. The innermost function scope enclosing the closure type is computed, and the corresponding function's execution space specifiers are assigned to the closure class member functions. If there is no enclosing function scope, the execution space specifier is \_\_host\_\_. 14 At present, the -std=c++11 flag is supported only for the following host compilers : gcc version >= 4.7, clang, icc >= 15, and xlc >= 13.1 15 including operator() www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 242 ----------------------- Page 259----------------------- C++ Language Support Examples of lambda expressions and computed execution space specifiers are shown below (in comments). auto globalVar = [] { return 0; }; // \_\_host\_\_ void f1(void) { auto l1 = [] { return 1; }; // \_\_host\_\_ } \_\_device\_\_ void f2(void) { auto l2 = [] { return 2; }; // \_\_device\_\_ } \_\_host\_\_ \_\_device\_\_ void f3(void) { auto l3 = [] { return 3; }; // \_\_host\_\_ \_\_device\_\_ } \_\_device\_\_ void f4(int (\*fp)() = [] { return 4; } /\* \_\_host\_\_ \*/) { } \_\_global\_\_ void f5(void) { auto l5 = [] { return 5; }; // \_\_device\_\_ } \_\_device\_\_ void f6(void) { struct S1\_t { static void helper(int (\*fp)() = [] {return 6; } /\* \_\_device\_\_ \*/) { } }; } The closure type of a lambda expression cannot be used in the type or non-type argument of a \_\_global\_\_ function template instantiation, unless the lambda is defined within a \_\_device\_\_ or \_\_global\_\_ function. Example: template <typename T> \_\_global\_\_ void foo(T in) { }; template <typename T> struct S1\_t { }; void bar(void) { auto temp1 = [] { }; foo<<<1,1>>> (temp1); // error: lambda closure type used in // template type argument foo<<<1,1>>> ( S1\_t<decltype(temp1)>()); // error: lambda closure type used in // template type argument } F.3.16.2. std::initializer\_list By default, the CUDA compiler will implicitly consider the member functions of std::initializer\_list to have \_\_host\_\_ \_\_device\_\_ execution space specifiers, and therefore they can be invoked directly from device code. The nvcc flag --no- host-device-initializer-list will disable this behavior; member functions of std::initializer\_list will then be considered as \_\_host\_\_ functions and will not be directly invokable from device code. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 243 ----------------------- Page 260----------------------- C++ Language Support Example: #include <initializer\_list> \_\_device\_\_ int foo(std::initializer\_list<int> in); \_\_device\_\_ void bar(void) { foo({4,5,6}); // (a) initializer list containing only // constant expressions. int i = 4; foo({i,5,6}); // (b) initializer list with at least one // non-constant element. // This form may have better performance than (a). } F.3.16.3. Rvalue references By default, the CUDA compiler will implicitly consider std::move and std::forward function templates to have \_\_host\_\_ \_\_device\_\_ execution space specifiers, and therefore they can be invoked directly from device code. The nvcc flag --no-host- device-move-forward will disable this behavior; std::move and std::forward will then be considered as \_\_host\_\_ functions and will not be directly invokable from device code. F.3.16.4. Constexpr functions and function templates By default, a constexpr function cannot be called from a function with incompatible execution space 16. The experimental nvcc flag --expt-relaxed-constexpr removes this restriction 17. When this flag is specified, host code can invoke a \_\_device\_\_ constexpr function and device code can invoke a \_\_host\_\_ constexpr function. nvcc will define the macro \_\_CUDACC\_RELAXED\_CONSTEXPR\_\_ when --expt-relaxed- constexpr has been specified. Note that a function template instantiation may not be a constexpr function even if the corresponding template is marked with the keyword constexpr (C++11 Standard Section [dcl.constexpr.p6]). F.3.16.5. Constexpr variables Let 'V' denote a namespace scope variable or a class static member variable that has been marked constexpr and that does not have execution space annotations (e.g., \_\_device\_\_, \_\_constant\_\_, \_\_shared\_\_). V is considered to be a host code variable. If V is of scalar type 18 other than long double and the type is not volatile-qualified, the value of V can be directly used in device code. In addition, if V is of a non-scalar type then scalar elements of V can be used inside a constexpr \_\_device\_\_ or \_\_host\_\_ \_\_device\_\_ function, if the call to the function is a constant expression 19. Device source code cannot contain a reference to V or take the address of V. 16 The restrictions are the same as with a non-constexpr callee function. 17 Note that the behavior of experimental flags may change in future compiler releases. 18 C++ Standard Section [basic.types] 19 C++ Standard Section [expr.const] www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 244 ----------------------- Page 261----------------------- C++ Language Support Example: constexpr int xxx = 10; constexpr int yyy = xxx + 4; struct S1\_t { static constexpr int qqq = 100; }; constexpr int host\_arr[] = { 1, 2, 3}; constexpr \_\_device\_\_ int get(int idx) { return host\_arr[idx]; } \_\_device\_\_ int foo(int idx) { int v1 = xxx + yyy + S1\_t::qqq; // OK const int &v2 = xxx; // error: reference to host constexpr // variable const int \*v3 = &xxx; // error: address of host constexpr // variable const int &v4 = S1\_t::qqq; // error: reference to host constexpr // variable const int \*v5 = &S1\_t::qqq; // error: address of host constexpr // variable v1 += get(2); // OK: 'get(2)' is a constant // expression. v1 += get(idx); // error: 'get(idx)' is not a constant // expression v1 += host\_arr[2]; // error: 'host\_arr' does not have // scalar type. return v1; } F.3.16.6. Inline namespaces For an input CUDA translation unit, the CUDA compiler may invoke the host compiler for compiling the host code within the translation unit. In the code passed to the host compiler, the CUDA compiler will inject additional compiler generated code, if the input CUDA translation unit contained a definition of any of the following entities: ‣ \_\_global\_\_ function or function template instantiation ‣ \_\_device\_\_, \_\_constant\_\_ ‣ variables with surface or texture type The compiler generated code contains a reference to the defined entity. If the entity is defined within an inline namespace and another entity of the same name and type signature is defined in an enclosing namespace, this reference may be considered ambiguous by the host compiler and host compilation will fail. This limitation can be avoided by using unique names for such entities defined within an inline namespace. Example: \_\_device\_\_ int Gvar; inline namespace N1 { \_\_device\_\_ int Gvar; } // <-- CUDA compiler inserts a reference to "Gvar" at this point in the // translation unit. This reference will be considered ambiguous by the // host compiler and compilation will fail. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 245 ----------------------- Page 262----------------------- C++ Language Support Example: inline namespace N1 { namespace N2 { \_\_device\_\_ int Gvar; } } namespace N2 { \_\_device\_\_ int Gvar; } // <-- CUDA compiler inserts reference to "::N2::Gvar" at this point in // the translation unit. This reference will be considered ambiguous by // the host compiler and compilation will fail. F.3.16.6.1. Inline unnamed namespaces The following entities cannot be declared in namespace scope within an inline unnamed namespace: ‣ \_\_managed\_\_, \_\_device\_\_, \_\_shared\_\_ and \_\_constant\_\_ variables ‣ \_\_global\_\_ function and function templates ‣ variables with surface or texture type Example: inline namespace { namespace N2 { template <typename T> \_\_global\_\_ void foo(void); // error \_\_global\_\_ void bar(void) { } // error template <> \_\_global\_\_ void foo<int>(void) { } // error \_\_device\_\_ int x1b; // error \_\_constant\_\_ int x2b; // error \_\_shared\_\_ int x3b; // error texture<int> q2; // error surface<int> s2; // error } }; F.3.16.7. thread\_local The thread\_local storage specifier is not allowed in device code. F.3.16.8. \_\_global\_\_ functions and function templates If the closure type associated with a lambda expression is used in a template argument of a \_\_global\_\_ function template instantiation, the lambda expression must either be defined in the immediate or nested block scope of a \_\_device\_\_ or \_\_global\_\_ function, or must be an extended lambda. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 246 ----------------------- Page 263----------------------- C++ Language Support Example: template <typename T> \_\_global\_\_ void kernel(T in) { } \_\_device\_\_ void foo\_device(void) { // All kernel instantiations in this function // are valid, since the lambdas are defined inside // a \_\_device\_\_ function. kernel<<<1,1>>> ( [] \_\_device\_\_ { } ); kernel<<<1,1>>> ( [] \_\_host\_\_ \_\_device\_\_ { } ); kernel<<<1,1>>> ( [] { } ); } auto lam1 = [] { }; auto lam2 = [] \_\_host\_\_ \_\_device\_\_ { }; void foo\_host(void) { // OK: instantiated with closure type of an extended \_\_device\_\_ lambda kernel<<<1,1>>> ( [] \_\_device\_\_ { } ); // OK: instantiated with closure type of an extended \_\_host\_\_ \_\_device\_\_ // lambda kernel<<<1,1>>> ( [] \_\_host\_\_ \_\_device\_\_ { } ); // error: unsupported: instantiated with closure type of a lambda // that is not an extended lambda kernel<<<1,1>>> ( [] { } ); // error: unsupported: instantiated with closure type of a lambda // that is not an extended lambda kernel<<<1,1>>> ( lam1); // error: unsupported: instantiated with closure type of a lambda // that is not an extended lambda kernel<<<1,1>>> ( lam2); } A \_\_global\_\_ function or function template cannot be declared as constexpr. A \_\_global\_\_ function or function template cannot have a parameter of type std::initializer\_list or va\_list. A \_\_global\_\_ function cannot have a parameter of rvalue reference type. A variadic \_\_global\_\_ function template has the following restrictions: ‣ Only a single pack parameter is allowed. ‣ The pack parameter must be listed last in the template parameter list. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 247 ----------------------- Page 264----------------------- C++ Language Support Example: // ok template <template <typename...> class Wrapper, typename... Pack> \_\_global\_\_ void foo1(Wrapper<Pack...>); // error: pack parameter is not last in parameter list template <typename... Pack, template <typename...> class Wrapper> \_\_global\_\_ void foo2(Wrapper<Pack...>); // error: multiple parameter packs template <typename... Pack1, int...Pack2, template<typename...> class Wrapper1, template<int...> class Wrapper2> \_\_global\_\_ void foo3(Wrapper1<Pack1...>, Wrapper2<Pack2...>); F.3.16.9. \_\_device\_\_/\_\_constant\_\_/\_\_shared\_\_ variables \_\_device\_\_, \_\_constant\_\_ and \_\_shared\_\_ variables cannot be marked with the keyword constexpr. F.3.16.10. Defaulted functions Execution space specifiers on a function that is explicitly-defaulted on its first declaration are ignored by the CUDA compiler. Instead, the CUDA compiler will infer the execution space specifiers as described in Implicitly-declared and explicitly- defaulted functions. Execution space specifiers are not ignored if the function is explicitly-defaulted, but not on its first declaration. Example: struct S1 { // warning: \_\_host\_\_ annotation is ignored on a function that // is explicitly-defaulted on its first declaration \_\_host\_\_ S1() = default; }; \_\_device\_\_ void foo1() { //note: \_\_device\_\_ execution space is derived for S1::S1 // based on implicit call from within \_\_device\_\_ function // foo1 S1 s1; } struct S2 { \_\_host\_\_ S2(); }; //note: S2::S2 is not defaulted on its first declaration, and // its execution space is fixed to \_\_host\_\_ based on its // first declaration. S2::S2() = default; \_\_device\_\_ void foo2() { // error: call from \_\_device\_\_ function 'foo2' to // \_\_host\_\_ function 'S2::S2' S2 s2; } www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 248 ----------------------- Page 265----------------------- C++ Language Support F.3.17. C++14 Features C++14 features enabled by default by the host compiler are also supported by nvcc. Passing nvcc -std=c++14 flag turns on all C++14 features and also invokes the host preprocessor, compiler and linker with the corresponding C++14 dialect option 20. This section describes the restrictions on the supported C++14 features. F.3.17.1. Functions with deduced return type A \_\_global\_\_ function cannot have a deduced return type. If a \_\_device\_\_ function has deduced return type, the CUDA frontend compiler will change the function declaration to have a void return type, before invoking the host compiler. This may cause issues for introspecting the deduced return type of the \_\_device\_\_ function in host code. Thus, the CUDA compiler will issue compile-time errors for referencing such deduced return type outside device function bodies, except if the reference is absent when \_\_CUDA\_ARCH\_\_ is undefined. Examples: \_\_device\_\_ auto fn1(int x) { return x; } \_\_device\_\_ decltype(auto) fn2(int x) { return x; } \_\_device\_\_ void device\_fn1() { // OK int (\*p1)(int) = fn1; } // error: referenced outside device function bodies decltype(fn1(10)) g1; void host\_fn1() { // error: referenced outside device function bodies int (\*p1)(int) = fn1; struct S\_local\_t { // error: referenced outside device function bodies decltype(fn2(10)) m1; S\_local\_t() : m1(10) { } }; } // error: referenced outside device function bodies template <typename T = decltype(fn2)> void host\_fn2() { } template<typename T> struct S1\_t { }; // error: referenced outside device function bodies struct S1\_derived\_t : S1\_t<decltype(fn1)> { }; 20 At present, the -std=c++14 flag is supported only for the following host compilers : gcc version >= 5.1, clang version >= 3.7 and icc version >= 17 www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 249 ----------------------- Page 266----------------------- C++ Language Support F.3.17.2. Variable templates A \_\_device\_\_/\_\_constant\_\_ variable template cannot have a const qualified type when using the Microsoft host compiler. Examples: // error: a \_\_device\_\_ variable template cannot // have a const qualified type on Windows template <typename T> \_\_device\_\_ const T d1(2); int \*const x = nullptr; // error: a \_\_device\_\_ variable template cannot // have a const qualified type on Windows template <typename T> \_\_device\_\_ T \*const d2(x); // OK template <typename T> \_\_device\_\_ const T \*d3; \_\_device\_\_ void fn() { int t1 = d1<int>; int \*const t2 = d2<int>; const int \*t3 = d3<int>; } F.4. Polymorphic Function Wrappers A polymorphic function wrapper class template nvstd::function is provided in the nvfunctional header. Instances of this class template can be used to store, copy and invoke any callable target, e.g., lambda expressions. nvstd::function can be used in both host and device code. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 250 ----------------------- Page 267----------------------- C++ Language Support Example: #include <nvfunctional> \_\_device\_\_ int foo\_d() { return 1; } \_\_host\_\_ \_\_device\_\_ int foo\_hd () { return 2; } \_\_host\_\_ int foo\_h() { return 3; } \_\_global\_\_ void kernel(int \*result) { nvstd::function<int ()> fn1 = foo\_d; nvstd::function<int ()> fn2 = foo\_hd; nvstd::function<int ()> fn3 = []() { return 10; }; \*result = fn1() + fn2() + fn3(); } \_\_host\_\_ \_\_device\_\_ void hostdevice\_func(int \*result) { nvstd::function<int ()> fn1 = foo\_hd; nvstd::function<int ()> fn2 = []() { return 10; }; \*result = fn1() + fn2(); } \_\_host\_\_ void host\_func(int \*result) { nvstd::function<int ()> fn1 = foo\_h; nvstd::function<int ()> fn2 = foo\_hd; nvstd::function<int ()> fn3 = []() { return 10; }; \*result = fn1() + fn2() + fn3(); } Instances of nvstd::function in host code cannot be initialized with the address of a \_\_device\_\_ function or with a functor whose operator() is a \_\_device\_\_ function. Instances of nvstd::function in device code cannot be initialized with the address of a \_\_host\_\_ function or with a functor whose operator() is a \_\_host\_\_ function. nvstd::function instances cannot be passed from host code to device code (and vice versa) at run time. nvstd::function cannot be used in the parameter type of a \_\_global\_\_ function, if the \_\_global\_\_ function is launched from host code. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 251 ----------------------- Page 268----------------------- C++ Language Support Example: #include <nvfunctional> \_\_device\_\_ int foo\_d() { return 1; } \_\_host\_\_ int foo\_h() { return 3; } auto lam\_h = [] { return 0; }; \_\_global\_\_ void k(void) { // error: initialized with address of \_\_host\_\_ function nvstd::function<int ()> fn1 = foo\_h; // error: initialized with address of functor with // \_\_host\_\_ operator() function nvstd::function<int ()> fn2 = lam\_h; } \_\_global\_\_ void kern(nvstd::function<int ()> f1) { } void foo(void) { // error: initialized with address of \_\_device\_\_ function nvstd::function<int ()> fn1 = foo\_d; auto lam\_d = [=] \_\_device\_\_ { return 1; }; // error: initialized with address of functor with // \_\_device\_\_ operator() function nvstd::function<int ()> fn2 = lam\_d; // error: passing nvstd::function from host to device kern<<<1,1>>> (fn2); } www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 252 ----------------------- Page 269----------------------- C++ Language Support nvstd::function is defined in the nvfunctional header as follows: namespace nvstd { template <class \_RetType, class ...\_ArgTypes> class function<\_RetType(\_ArgTypes...)> { public: // constructors \_\_device\_\_ \_\_host\_\_ function() noexcept; \_\_device\_\_ \_\_host\_\_ function(nullptr\_t) noexcept; \_\_device\_\_ \_\_host\_\_ function(const function &); \_\_device\_\_ \_\_host\_\_ function(function &&); template<class \_F> \_\_device\_\_ \_\_host\_\_ function(\_F); // destructor \_\_device\_\_ \_\_host\_\_ ~function(); // assignment operators \_\_device\_\_ \_\_host\_\_ function& operator=(const function&); \_\_device\_\_ \_\_host\_\_ function& operator=(function&&); \_\_device\_\_ \_\_host\_\_ function& operator=(nullptr\_t); \_\_device\_\_ \_\_host\_\_ function& operator=(\_F&&); // swap \_\_device\_\_ \_\_host\_\_ void swap(function&) noexcept; // function capacity \_\_device\_\_ \_\_host\_\_ explicit operator bool() const noexcept; // function invocation \_\_device\_\_ \_RetType operator()(\_ArgTypes...) const; }; // null pointer comparisons template <class \_R, class... \_ArgTypes> \_\_device\_\_ \_\_host\_\_ bool operator==(const function<\_R(\_ArgTypes...)>&, nullptr\_t) noexcept; template <class \_R, class... \_ArgTypes> \_\_device\_\_ \_\_host\_\_ bool operator==(nullptr\_t, const function<\_R(\_ArgTypes...)>&) noexcept; template <class \_R, class... \_ArgTypes> \_\_device\_\_ \_\_host\_\_ bool operator!=(const function<\_R(\_ArgTypes...)>&, nullptr\_t) noexcept; template <class \_R, class... \_ArgTypes> \_\_device\_\_ \_\_host\_\_ bool operator!=(nullptr\_t, const function<\_R(\_ArgTypes...)>&) noexcept; // specialized algorithms template <class \_R, class... \_ArgTypes> \_\_device\_\_ \_\_host\_\_ void swap(function<\_R(\_ArgTypes...)>&, function<\_R(\_ArgTypes...)>&); } www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 253 ----------------------- Page 270----------------------- C++ Language Support F.5. Extended Lambdas The nvcc flag '--extended-lambda' allows explicit execution space annotations in a lambda expression 21. The execution space annotations should be present after the 'lambda-introducer' and before the optional 'lambda-declarator'. nvcc will define the macro \_\_CUDACC\_EXTENDED\_LAMBDA\_\_ when the '--extended-lambda' flag has been specified. An 'extended \_\_device\_\_ lambda' is a lambda expression that is annotated explicitly with '\_\_device\_\_', and is defined within the immediate or nested block scope of a \_\_host\_\_ or \_\_host\_\_ \_\_device\_\_ function. An 'extended \_\_host\_\_ \_\_device\_\_ lambda' is a lambda expression that is annotated explicitly with both '\_\_host\_\_' and '\_\_device\_\_', and is defined within the immediate or nested block scope of a \_\_host\_\_ or \_\_host\_\_ \_\_device\_\_ function. An 'extended lambda' denotes either an extended \_\_device\_\_ lambda or an extended \_\_host\_\_ \_\_device\_\_ lambda. Extended lambdas can be used in the type arguments of \_\_global\_\_ function template instantiation. If the execution space annotations are not explicitly specified, they are computed based on the scopes enclosing the closure class associated with the lambda, as described in the section on C++11 support. The execution space annotations are applied to all methods of the closure class associated with the lambda. 21 When using the icc host compiler, this flag is only supported for icc >= 1800. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 254 ----------------------- Page 271----------------------- C++ Language Support Example: void foo\_host(void) { // not an extended lambda: no explicit execution space annotations auto lam1 = [] { }; // extended \_\_device\_\_ lambda auto lam2 = [] \_\_device\_\_ { }; // extended \_\_host\_\_ \_\_device\_\_ lambda auto lam3 = [] \_\_host\_\_ \_\_device\_\_ { }; // not an extended lambda: explicitly annotated with only '\_\_host\_\_' auto lam4 = [] \_\_host\_\_ { }; } \_\_host\_\_ \_\_device\_\_ void foo\_host\_device(void) { // not an extended lambda: no explicit execution space annotations auto lam1 = [] { }; // extended \_\_device\_\_ lambda auto lam2 = [] \_\_device\_\_ { }; // extended \_\_host\_\_ \_\_device\_\_ lambda auto lam3 = [] \_\_host\_\_ \_\_device\_\_ { }; // not an extended lambda: explicitly annotated with only '\_\_host\_\_' auto lam4 = [] \_\_host\_\_ { }; } \_\_device\_\_ void foo\_device(void) { // none of the lambdas within this function are extended lambdas, // because the enclosing function is not a \_\_host\_\_ or \_\_host\_\_ \_\_device\_\_ // function. auto lam1 = [] { }; auto lam2 = [] \_\_device\_\_ { }; auto lam3 = [] \_\_host\_\_ \_\_device\_\_ { }; auto lam4 = [] \_\_host\_\_ { }; } // lam1 and lam2 are not extended lambdas because they are not defined // within a \_\_host\_\_ or \_\_host\_\_ \_\_device\_\_ function. auto lam1 = [] { }; auto lam2 = [] \_\_host\_\_ \_\_device\_\_ { }; F.5.1. Extended Lambda Type Traits The compiler provides type traits to detect closure types for extended lambdas at compile time: \_\_nv\_is\_extended\_device\_lambda\_closure\_type(type): If 'type' is the closure class created for an extended \_\_device\_\_ lambda, then the trait is true, otherwise it is false. \_\_nv\_is\_extended\_host\_device\_lambda\_closure\_type(type): If 'type' is the closure class created for an extended \_\_host\_\_ \_\_device\_\_ lambda, then the trait is true, otherwise it is false. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 255 ----------------------- Page 272----------------------- C++ Language Support These traits can be used in all compilation modes, irrespective of whether lambdas or extended lambdas are enabled22. Example: #define IS\_D\_LAMBDA(X) \_\_nv\_is\_extended\_device\_lambda\_closure\_type(X) #define IS\_HD\_LAMBDA(X) \_\_nv\_is\_extended\_host\_device\_lambda\_closure\_type(X) auto lam0 = [] \_\_host\_\_ \_\_device\_\_ { }; void foo(void) { auto lam1 = [] { }; auto lam2 = [] \_\_device\_\_ { }; auto lam3 = [] \_\_host\_\_ \_\_device\_\_ { }; // lam0 is not an extended lambda (since defined outside function scope) static\_assert(!IS\_D\_LAMBDA(decltype(lam0)), ""); static\_assert(!IS\_HD\_LAMBDA(decltype(lam0)), ""); // lam1 is not an extended lambda (since no execution space annotations) static\_assert(!IS\_D\_LAMBDA(decltype(lam1)), ""); static\_assert(!IS\_HD\_LAMBDA(decltype(lam1)), ""); // lam2 is an extended \_\_device\_\_ lambda static\_assert(IS\_D\_LAMBDA(decltype(lam2)), ""); static\_assert(!IS\_HD\_LAMBDA(decltype(lam2)), ""); // lam3 is an extended \_\_host\_\_ \_\_device\_\_ lambda static\_assert(!IS\_D\_LAMBDA(decltype(lam3)), ""); static\_assert(IS\_HD\_LAMBDA(decltype(lam3)), ""); } F.5.2. Extended Lambda Restrictions The CUDA compiler will replace an extended lambda expression with an instance of a placeholder type defined in namespace scope, before invoking the host compiler. The template argument of the placeholder type requires taking the address of a function enclosing the original extended lambda expression. This is required for the correct execution of any \_\_global\_\_ function template whose template argument involves the closure type of an extended lambda. The enclosing function is computed as follows. By definition, the extended lambda is present within the immediate or nested block scope of a \_\_host\_\_ or \_\_host\_\_ \_\_device\_\_ function. If this function is not the operator() of a lambda expression, then it is considered the enclosing function for the extended lambda. Otherwise, the extended lambda is defined within the immediate or nested block scope of the operator() of one or more enclosing lambda expressions. If the outermost such lambda expression is defined in the immediate or nested block scope of a function F, then F is the computed enclosing function, else the enclosing function does not exist. 22 The traits will always return false if extended lambda mode is not active. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 256 ----------------------- Page 273----------------------- C++ Language Support Example: void foo(void) { // enclosing function for lam1 is "foo" auto lam1 = [] \_\_device\_\_ { }; auto lam2 = [] { auto lam3 = [] { // enclosing function for lam4 is "foo" auto lam4 = [] \_\_host\_\_ \_\_device\_\_ { }; }; }; } auto lam6 = [] { // enclosing function for lam7 does not exist auto lam7 = [] \_\_host\_\_ \_\_device\_\_ { }; }; Here are the restrictions on extended lambdas: 1. An extended lambda cannot be defined inside another extended lambda expression. Example: void foo(void) { auto lam1 = [] \_\_host\_\_ \_\_device\_\_ { // error: extended lambda defined within another extended lambda auto lam2 = [] \_\_host\_\_ \_\_device\_\_ { }; }; 2. An extended lambda cannot be defined inside a generic lambda expression. Example: void foo(void) { auto lam1 = [] (auto) { // error: extended lambda defined within a generic lambda auto lam2 = [] \_\_host\_\_ \_\_device\_\_ { }; }; 3. If an extended lambda is defined within the immediate or nested block scope of one or more nested lambda expression, the outermost such lambda expression must be defined inside the immediate or nested block scope of a function. Example: auto lam1 = [] { // error: outer enclosing lambda is not defined within a // non-lambda-operator() function. auto lam2 = [] \_\_host\_\_ \_\_device\_\_ { }; }; 4. The enclosing function for the extended lambda must be named and its address can be taken. If the enclosing function is a class member, then the following conditions must be satisfied: ‣ All classes enclosing the member function must have a name. ‣ The member function must not have private or protected access within its parent class. ‣ All enclosing classes must not have private or protected access within their respective parent classes. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 257 ----------------------- Page 274----------------------- C++ Language Support Example: void foo(void) { // OK auto lam1 = [] \_\_device\_\_ { return 0; }; { // OK auto lam2 = [] \_\_device\_\_ { return 0; }; // OK auto lam3 = [] \_\_device\_\_ \_\_host\_\_ { return 0; }; } } struct S1\_t { S1\_t(void) { // Error: cannot take address of enclosing function auto lam4 = [] \_\_device\_\_ { return 0; }; } }; class C0\_t { void foo(void) { // Error: enclosing function has private access in parent class auto temp1 = [] \_\_device\_\_ { return 10; }; } struct S2\_t { void foo(void) { // Error: enclosing class S2\_t has private access in its // parent class auto temp1 = [] \_\_device\_\_ { return 10; }; } }; }; 5. An extended lambda cannot be defined in a class that is local to a function. Example: void foo(void) { struct S1\_t { void bar(void) { // Error: bar is member of a class that is local to a function. auto lam4 = [] \_\_host\_\_ \_\_device\_\_ { return 0; }; } }; } 6. The enclosing function for an extended lambda cannot have deduced return type. Example: auto foo(void) { // Error: the return type of foo is deduced. auto lam1 = [] \_\_host\_\_ \_\_device\_\_ { return 0; }; } 7. \_\_host\_\_ \_\_device\_\_ extended lambdas cannot be generic lambdas. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 258 ----------------------- Page 275----------------------- C++ Language Support Example: void foo(void) { // Error: \_\_host\_\_ \_\_device\_\_ extended lambdas cannot be // generic lambdas. auto lam1 = [] \_\_host\_\_ \_\_device\_\_ (auto i) { return i; }; // Error: \_\_host\_\_ \_\_device\_\_ extended lambdas cannot be // generic lambdas. auto lam2 = [] \_\_host\_\_ \_\_device\_\_ (auto ...i) { return sizeof...(i); }; } 8. If the enclosing function is an instantiation of a function template or a member function template, and/or the function is a member of a class template, the template(s) must satisfy the following constraints: ‣ The template must have at most one variadic parameter, and it must be listed last in the template parameter list. ‣ The template parameters must be named. ‣ The template instantiation argument types cannot involve types that are either local to a function (except for closure types for extended lambdas), or are private or protected class members. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 259 ----------------------- Page 276----------------------- C++ Language Support Example: template <typename T> \_\_global\_\_ void kern(T in) { in(); } template <typename... T> struct foo {}; template < template <typename...> class T, typename... P1, typename... P2> void bar1(const T<P1...>, const T<P2...>) { // Error: enclosing function has multiple parameter packs auto lam1 = [] \_\_device\_\_ { return 10; }; } template < template <typename...> class T, typename... P1, typename T2> void bar2(const T<P1...>, T2) { // Error: for enclosing function, the // parameter pack is not last in the template parameter list. auto lam1 = [] \_\_device\_\_ { return 10; }; } template <typename T, T> void bar3(void) { // Error: for enclosing function, the second template // parameter is not named. auto lam1 = [] \_\_device\_\_ { return 10; }; } int main() { foo<char, int, float> f1; foo<char, int> f2; bar1(f1, f2); bar2(f1, 10); bar3<int, 10>(); } www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 260 ----------------------- Page 277----------------------- C++ Language Support Example: template <typename T> \_\_global\_\_ void kern(T in) { in(); } template <typename T> void bar4(void) { auto lam1 = [] \_\_device\_\_ { return 10; }; kern<<<1,1>>> (lam1); } struct C1\_t { struct S1\_t { }; friend int main(void); }; int main() { struct S1\_t { }; // Error: enclosing function for device lambda in bar4 // is instantiated with a type local to main. bar4<S1\_t>(); // Error: enclosing function for device lambda in bar4 // is instantiated with a type that is a private member // of a class. bar4<C1\_t::S1\_t>(); } 9. With Visual Studio 2013 and later Visual Studio host compilers, the enclosing function must have external linkage. The restriction is present because this host compiler does not support using the address of non-extern linkage functions as template arguments, which is needed by the CUDA compiler transformations to support extended lambdas. 10. An extended lambda has the following restrictions on captured variables: ‣ In the code sent to the host compiler, the variable may be passed by value to a sequence of helper functions before being used to direct-initialize the field of the class type used to represent the closure type for the extended lambda23. ‣ A variable can only be captured by value. ‣ A variable of array type cannot be captured if the number of array dimensions is greater than 7. ‣ For a variable of array type, in the code sent to the host compiler, the closure type's array field is first default-initialized, and then each element of the array field is copy-assigned from the corresponding element of the captured array variable. Therefore, the array element type must be default-constructible and copy-assignable in host code. ‣ A function parameter that is an element of a variadic argument pack cannot be captured. ‣ The type of the captured variable cannot involve types that are either local to a function (except for closure types of extended lambdas), or are private or protected class members. ‣ For a \_\_host\_\_ \_\_device\_\_ extended lambda, the types used in the return or parameter types of the lambda expression's operator() cannot involve types that are either local to a function (except for closure types of extended lambdas), or are private or protected class members. 23 In contrast, the C++ standard specifies that the captured variable is used to direct-initialize the field of the closure type. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 261 ----------------------- Page 278----------------------- C++ Language Support ‣ Init-capture is not supported for \_\_host\_\_ \_\_device\_\_ extended lambdas. Init- capture is supported for \_\_device\_\_ extended lambdas, except when the init- capture is of array type or of type std::initializer\_list. ‣ The function call operator for an extended lambda is not constexpr. The closure type for an extended lambda is not a literal type. The constexpr specifier cannot be used in the declaration of an extended lambda. Example void foo(void) { // OK: an init-capture is allowed for an // extended \_\_device\_\_ lambda. auto lam1 = [x = 1] \_\_device\_\_ () { return x; }; // Error: an init-capture is not allowed for // an extended \_\_host\_\_ \_\_device\_\_ lambda. auto lam2 = [x = 1] \_\_host\_\_ \_\_device\_\_ () { return x; }; int a = 1; // Error: an extended \_\_device\_\_ lambda cannot capture // variables by reference. auto lam3 = [&a] \_\_device\_\_ () { return a; }; // Error: by-reference capture is not allowed // for an extended \_\_device\_\_ lambda. auto lam4 = [&x = a] \_\_device\_\_ () { return x; }; struct S1\_t { }; S1\_t s1; // Error: a type local to a function cannot be used in the type // of a captured variable. auto lam6 = [s1] \_\_device\_\_ () { }; // Error: an init-capture cannot be of type std::initializer\_list. auto lam7 = [x = {11}] \_\_device\_\_ () { }; std::initializer\_list<int> b = {11,22,33}; // Error: an init-capture cannot be of type std::initializer\_list. auto lam8 = [x = b] \_\_device\_\_ () { }; } 11. When parsing a function, the CUDA compiler assigns a counter value to each extended lambda within that function. This counter value is used in the substituted named type passed to the host compiler. Hence, whether or not an extended lambda is defined within a function should not depend on a particular value of \_\_CUDA\_ARCH\_\_, or on \_\_CUDA\_ARCH\_\_ being undefined. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 262 ----------------------- Page 279----------------------- C++ Language Support Example template <typename T> \_\_global\_\_ void kernel(T in) { in(); } \_\_host\_\_ \_\_device\_\_ void foo(void) { // Error: the number and relative declaration // order of extended lambdas depends on // \_\_CUDA\_ARCH\_\_ #if defined(\_\_CUDA\_ARCH\_\_) auto lam1 = [] \_\_device\_\_ { return 0; }; auto lam1b = [] \_\_host\_\_\_ \_\_device\_\_ { return 10; }; #endif auto lam2 = [] \_\_device\_\_ { return 4; }; kernel<<<1,1>>> (lam2); } 12. As described above, the CUDA compiler replaces a \_\_device\_\_ extended lambda defined in a host function with a placeholder type defined in namespace scope. This placeholder type does not define a operator() function equivalent to the original lambda declaration. An attempt to determine the return type or parameter types of the operator() function may therefore work incorrectly in host code, as the code processed by the host compiler will be semantically different than the input code processed by the CUDA compiler. However, it is ok to introspect the return type or parameter types of the operator() function within device code. Note that this restriction does not apply to \_\_host\_\_ \_\_device\_\_ extended lambdas. Example #include <type\_traits> void foo(void) { auto lam1 = [] \_\_device\_\_ { return 10; }; // Error: attempt to extract the return type // of a \_\_device\_\_ lambda in host code std::result\_of<decltype(lam1)()>::type xx1 = 1; auto lam2 = [] \_\_host\_\_ \_\_device\_\_ { return 10; }; // OK : lam2 represents a \_\_host\_\_ \_\_device\_\_ extended lambda std::result\_of<decltype(lam2)()>::type xx2 = 1; } 13. If the functor object represented by an extended lambda is passed from host to device code (e.g., as the argument of a \_\_global\_\_ function), then any expression in the body of the lambda expression that captures variables must be remain unchanged irrespective of whether the \_\_CUDA\_ARCH\_\_ macro is defined, and whether the macro has a particular value. This restriction arises because the lambda's closure class layout depends on the order in which captured variables are encountered when the compiler processes the lambda expression; the program may execute incorrectly if the closure class layout differs in device and host compilation. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 263 ----------------------- Page 280----------------------- C++ Language Support Example \_\_device\_\_ int result; template <typename T> \_\_global\_\_ void kernel(T in) { result = in(); } void foo(void) { int x1 = 1; auto lam1 = [=] \_\_host\_\_ \_\_device\_\_ { // Error: "x1" is only captured when \_\_CUDA\_ARCH\_\_ is defined. #ifdef \_\_CUDA\_ARCH\_\_ return x1 + 1; #else return 10; #endif }; kernel<<<1,1>>> (lam1); } 14. As described previously, the CUDA compiler replaces an extended \_\_device\_\_ lambda expression with an instance of a placeholder type in the code sent to the host compiler. This placeholder type does not define a pointer-to-function conversion operator in host code, however the conversion operator is provided in device code. Note that this restriction does not apply to \_\_host\_\_ \_\_device\_\_ extended lambdas. Example template <typename T> \_\_global\_\_ void kern(T in) { int (\*fp)(double) = in; // OK: conversion in device code is supported fp(0); auto lam1 = [](double) { return 1; }; // OK: conversion in device code is supported fp = lam1; fp(0); } void foo(void) { auto lam\_d = [] \_\_device\_\_ (double) { return 1; }; auto lam\_hd = [] \_\_host\_\_ \_\_device\_\_ (double) { return 1; }; kern<<<1,1>>> (lam\_d); kern<<<1,1>>> (lam\_hd); // OK : conversion for \_\_host\_\_ \_\_device\_\_ lambda is supported // in host code int (\*fp)(double) = lam\_hd; // Error: conversion for \_\_device\_\_ lambda is not supported in // host code. int (\*fp2)(double) = lam\_d; } The CUDA compiler will generate compiler diagnostics for a subset of cases described in 1-11; no diagnostic will be generated for cases 12-15, but the host compiler may fail to compile the generated code. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 264 ----------------------- Page 281----------------------- C++ Language Support F.5.3. Notes on \_\_host\_\_ \_\_device\_\_ lambdas Unlike \_\_device\_\_ lambdas, \_\_host\_\_ \_\_device\_\_ lambdas can be called from host code. As described earlier, the CUDA compiler replaces an extended lambda expression defined in host code with an instance of a named placeholder type. The placeholder type for an extended \_\_host\_\_ \_\_device\_\_ lambda invokes the orignal lambda's operator() with an indirect function call 24. The presence of the indirect function call may cause an extended \_\_host\_\_ \_\_device\_\_ lambda to be less optimized by the host compiler than lambdas that are implicitly or explicitly \_\_host\_\_ only. In the latter case, the host compiler can easily inline the body of the lambda into the calling context. But in case of an extended \_\_host\_\_ \_\_device\_\_ lambda, the host compiler encounters the indirect function call and may not be able to easily inline the original \_\_host\_\_ \_\_device\_\_ lambda body. F.5.4. \*this Capture By Value When a lambda is defined within a non-static class member function, and the body of the lambda refers to a class member variable, C++11/C++14 rules require that the this pointer of the class is captured by value, instead of the referenced member variable. If the lambda is an extended \_\_device\_\_ or \_\_host\_\_ \_\_device\_\_ lambda defined in a host function, and the lambda is executed on the GPU, accessing the referenced member variable on the GPU will cause a run time error if the this pointer points to host memory. 24 The closure object is stored in a type-elided container similar to std::function. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 265 ----------------------- Page 282----------------------- C++ Language Support Example: #include <cstdio> template <typename T> \_\_global\_\_ void foo(T in) { printf("\n value = %d", in()); } struct S1\_t { int xxx; \_\_host\_\_ \_\_device\_\_ S1\_t(void) : xxx(10) { }; void doit(void) { auto lam1 = [=] \_\_device\_\_ { // reference to "xxx" causes // the 'this' pointer (S1\_t\*) to be captured by value return xxx + 1; }; // Kernel launch fails at run time because 'this->xxx' // is not accessible from the GPU foo<<<1,1>>> (lam1); cudaDeviceSynchronize(); } }; int main(void) { S1\_t s1; s1.doit(); } C++17 solves this problem by adding a new "\*this" capture mode. In this mode, the compiler makes a copy of the object denoted by "\*this" instead of capturing the pointer this by value. The "\*this" capture mode is described in more detail here: http:// www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0018r3.html . The CUDA compiler supports the "\*this" capture mode for lambdas defined within \_\_device\_\_ and \_\_global\_\_ functions and for extended \_\_device\_\_ lambdas defined in host code, when the --extended-lambda nvcc flag is used. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 266 ----------------------- Page 283----------------------- C++ Language Support Here's the above example modified to use "\*this" capture mode: #include <cstdio> template <typename T> \_\_global\_\_ void foo(T in) { printf("\n value = %d", in()); } struct S1\_t { int xxx; \_\_host\_\_ \_\_device\_\_ S1\_t(void) : xxx(10) { }; void doit(void) { // note the "\*this" capture specification auto lam1 = [=, \*this] \_\_device\_\_ { // reference to "xxx" causes // the object denoted by '\*this' to be captured by // value, and the GPU code will access copy\_of\_star\_this->xxx return xxx + 1; }; // Kernel launch succeeds foo<<<1,1>>> (lam1); cudaDeviceSynchronize(); } }; int main(void) { S1\_t s1; s1.doit(); } www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 267 ----------------------- Page 284----------------------- C++ Language Support "\*this" capture mode is not allowed for unannotated lambdas defined in host code, or for extended \_\_host\_\_ \_\_device\_\_ lambdas. Examples of supported and unsupported usage: struct S1\_t { int xxx; \_\_host\_\_ \_\_device\_\_ S1\_t(void) : xxx(10) { }; void host\_func(void) { // OK: use in an extended \_\_device\_\_ lambda auto lam1 = [=, \*this] \_\_device\_\_ { return xxx; }; // Error: use in an extended \_\_host\_\_ \_\_device\_\_ lambda auto lam2 = [=, \*this] \_\_host\_\_ \_\_device\_\_ { return xxx; }; // Error: use in an unannotated lambda in host function auto lam3 = [=, \*this] { return xxx; }; } \_\_device\_\_ void device\_func(void) { // OK: use in a lambda defined in a \_\_device\_\_ function auto lam1 = [=, \*this] \_\_device\_\_ { return xxx; }; // OK: use in a lambda defined in a \_\_device\_\_ function auto lam2 = [=, \*this] \_\_host\_\_ \_\_device\_\_ { return xxx; }; // OK: use in a lambda defined in a \_\_device\_\_ function auto lam3 = [=, \*this] { return xxx; }; } \_\_host\_\_ \_\_device\_\_ void host\_device\_func(void) { // OK: use in an extended \_\_device\_\_ lambda auto lam1 = [=, \*this] \_\_device\_\_ { return xxx; }; // Error: use in an extended \_\_host\_\_ \_\_device\_\_ lambda auto lam2 = [=, \*this] \_\_host\_\_ \_\_device\_\_ { return xxx; }; // Error: use in an unannotated lambda in a \_\_host\_\_ \_\_device\_\_ function auto lam3 = [=, \*this] { return xxx; }; } }; F.5.5. Additional Notes 1. ADL Lookup: As described earlier, the CUDA compiler will replace an extended lambda expression with an instance of a placeholder type, before invoking the host compiler. One template argument of the placeholder type uses the address of the function enclosing the original lambda expression. This may cause additional namespaces to participate in argument dependent lookup (ADL), for any host function call whose argument types involve the closure type of the extended lambda expression. This may cause an incorrect function to be selected by the host compiler. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 268 ----------------------- Page 285----------------------- C++ Language Support Example: namespace N1 { struct S1\_t { }; template <typename T> void foo(T); }; namespace N2 { template <typename T> int foo(T); template <typename T> void doit(T in) { foo(in); } } void bar(N1::S1\_t in) { /\* extended \_\_device\_\_ lambda. In the code sent to the host compiler, this is replaced with the placeholder type instantiation expression ' \_\_nv\_dl\_wrapper\_t< \_\_nv\_dl\_tag<void (\*)(N1::S1\_t in),(&bar),1> > { }' As a result, the namespace 'N1' participates in ADL lookup of the call to "foo" in the body of N2::doit, causing ambiguity. \*/ auto lam1 = [=] \_\_device\_\_ { }; N2::doit(lam1); } In the example above, the CUDA compiler replaced the extended lambda with a placeholder type that involves the N1 namespace. As a result, the namespace N1 participates in the ADL lookup for foo(in) in the body of N2::doit, and host compilation fails because multiple overload candidates N1::foo and N2::foo are found. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 269 ----------------------- Page 286----------------------- C++ Language Support F.6. Code Samples F.6.1. Data Aggregation Class class PixelRGBA { public: \_\_device\_\_ PixelRGBA(): r\_(0), g\_(0), b\_(0), a\_(0) { } \_\_device\_\_ PixelRGBA(unsigned char r, unsigned char g, unsigned char b, unsigned char a = 255): r\_(r), g\_(g), b\_(b), a\_(a) { } private: unsigned char r\_, g\_, b\_, a\_; friend PixelRGBA operator+(const PixelRGBA&, const PixelRGBA&); }; \_\_device\_\_ PixelRGBA operator+(const PixelRGBA& p1, const PixelRGBA& p2) { return PixelRGBA(p1.r\_ + p2.r\_, p1.g\_ + p2.g\_, p1.b\_ + p2.b\_, p1.a\_ + p2.a\_); } \_\_device\_\_ void func(void) { PixelRGBA p1, p2; // ... // Initialization of p1 and p2 here PixelRGBA p3 = p1 + p2; } F.6.2. Derived Class \_\_device\_\_ void\* operator new(size\_t bytes, MemoryPool& p); \_\_device\_\_ void operator delete(void\*, MemoryPool& p); class Shape { public: \_\_device\_\_ Shape(void) { } \_\_device\_\_ void putThis(PrintBuffer \*p) const; \_\_device\_\_ virtual void Draw(PrintBuffer \*p) const { p->put("Shapeless"); } \_\_device\_\_ virtual ~Shape() {} }; class Point : public Shape { public: \_\_device\_\_ Point() : x(0), y(0) {} \_\_device\_\_ Point(int ix, int iy) : x(ix), y(iy) { } \_\_device\_\_ void PutCoord(PrintBuffer \*p) const; \_\_device\_\_ void Draw(PrintBuffer \*p) const; \_\_device\_\_ ~Point() {} private: int x, y; }; \_\_device\_\_ Shape\* GetPointObj(MemoryPool& pool) { Shape\* shape = new(pool) Point(rand(-20,10), rand(-100,-20)); return shape; } www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 270 ----------------------- Page 287----------------------- C++ Language Support F.6.3. Class Template template <class T> class myValues { T values[MAX\_VALUES]; public: \_\_device\_\_ myValues(T clear) { ... } \_\_device\_\_ void setValue(int Idx, T value) { ... } \_\_device\_\_ void putToMemory(T\* valueLocation) { ... } }; template <class T> void \_\_global\_\_ useValues(T\* memoryBuffer) { myValues<T> myLocation(0); ... } \_\_device\_\_ void\* buffer; int main() { ... useValues<int><<<blocks, threads>>> (buffer); ... } F.6.4. Function Template template <typename T> \_\_device\_\_ bool func(T x) { ... return (...); } template <> \_\_device\_\_ bool func<int>(T x) // Specialization { return true; } // Explicit argument specification bool result = func<double>(0.5); // Implicit argument deduction int x = 1; bool result = func(x); www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 271 ----------------------- Page 288----------------------- C++ Language Support F.6.5. Functor Class class Add { public: \_\_device\_\_ float operator() (float a, float b) const { return a + b; } }; class Sub { public: \_\_device\_\_ float operator() (float a, float b) const { return a - b; } }; // Device code template<class O> \_\_global\_\_ void VectorOperation(const float \* A, const float \* B, float \* C, unsigned int N, O op) { unsigned int iElement = blockDim .x \* blockIdx .x + threadIdx.x; if (iElement < N) C[iElement] = op(A[iElement], B[iElement]); } // Host code int main() { ... VectorOperation<<<blocks, threads>>> (v1, v2, v3, N, Add()); ... } www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 272 ----------------------- Page 289----------------------- Appendix G. TEXTURE FETCHING This appendix gives the formula used to compute the value returned by the texture functions of Texture Functions depending on the various attributes of the texture reference (see Texture and Surface Memory). The texture bound to the texture reference is represented as an array T of ‣ N texels for a one-dimensional texture, ‣ N x M texels for a two-dimensional texture, ‣ N x M x L texels for a three-dimensional texture. It is fetched using non-normalized texture coordinates x, y, and z, or the normalized texture coordinates x/N, y/M, and z/L as described in Texture Memory. In this appendix, the coordinates are assumed to be in the valid range. Texture Memory explained how out-of-range coordinates are remapped to the valid range based on the addressing mode. G.1. Nearest-Point Sampling In this filtering mode, the value returned by the texture fetch is ‣ tex(x)=T[i] for a one-dimensional texture, ‣ tex(x,y)=T[i,j] for a two-dimensional texture, ‣ tex(x,y,z)=T[i,j,k] for a three-dimensional texture, where i=floor(x), j=floor(y), and k=floor(z). Figure 15 illustrates nearest-point sampling for a one-dimensional texture with N=4. For integer textures, the value returned by the texture fetch can be optionally remapped to [0.0, 1.0] (see Texture Memory). www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 273 ----------------------- Page 290----------------------- Texture Fetching tex(x) T[3] T[0] T[2] T[1] x 0 1 2 3 4 Non-Norm alized 0 0.25 0.5 0.75 1 Norm alized Figure 15 Nearest-Point Sampling Filtering Mode Nearest-point sampling of a one-dimensional texture of four texels. G.2. Linear Filtering In this filtering mode, which is only available for floating-point textures, the value returned by the texture fetch is ‣ tex(x)=(1−α)T[i]+αT[i+1] for a one-dimensional texture, ‣ tex(x,y)=(1−α)(1−β)T[i,j]+α(1−β)T[i+1,j]+(1−α)βT[i,j+1]+αβT[i+1,j+1] for a two- dimensional texture, ‣ tex(x,y,z) = (1−α)(1−β)(1−γ)T[i,j,k]+α(1−β)(1−γ)T[i+1,j,k]+ (1−α)β(1−γ)T[i,j+1,k]+αβ(1−γ)T[i+1,j+1,k]+ (1−α)(1−β)γT[i,j,k+1]+α(1−β)γT[i+1,j,k+1]+ (1−α)βγT[i,j+1,k+1]+αβγT[i+1,j+1,k+1] for a three-dimensional texture, where: ‣ i=floor(x ), α=frac(x ), x =x-0.5, B B B ‣ j=floor(y ), β=frac(y ), y =y-0.5, B B B ‣ k=floor(z ), γ=frac(z ), z = z-0.5, B B B α, β, and γ are stored in 9-bit fixed point format with 8 bits of fractional value (so 1.0 is exactly represented). Figure 16 illustrates linear filtering of a one-dimensional texture with N=4. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 274 ----------------------- Page 291----------------------- Texture Fetching tex(x) T[3] T[0] T[2] T[1] x 0 1 2 3 4 Non-Norm alized 0 0.25 0.5 0.75 1 Norm alized Figure 16 Linear Filtering Mode Linear filtering of a one-dimensional texture of four texels in clamp addressing mode. G.3. Table Lookup A table lookup TL(x) where x spans the interval [0,R] can be implemented as TL(x)=tex((N-1)/R)x+0.5) in order to ensure that TL(0)=T[0] and TL(R)=T[N-1]. Figure 17 illustrates the use of texture filtering to implement a table lookup with R=4 or R=1 from a one-dimensional texture with N=4. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 275 ----------------------- Page 292----------------------- Texture Fetching TL(x) T[3] T[0] T[2] T[1] x 0 4/3 8/3 4 0 1/3 2/3 1 Figure 17 One-Dimensional Table Lookup Using Linear Filtering www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 276 ----------------------- Page 293----------------------- Appendix H. COMPUTE CAPABILITIES The general specifications and features of a compute device depend on its compute capability (see Compute Capability). Table 14 gives the features and technical specifications associated to each compute capability. Floating-Point Standard reviews the compliance with the IEEE floating-point standard. Sections Compute Capability 3.x, Compute Capability 5.x, Compute Capability 6.x, and Compute Capability 7.x give more details on the architecture of devices of compute capability 3.x, 5.x, 6.x, and 7.x respectively. H.1. Features and Technical Specifications Table 14 Feature Support per Compute Capability Feature Support Compute Capability 3.5, (Unlisted features are supported for all 3.7, 3.0 3.2 5.3 6.x 7.x compute capabilities) 5.0, 5.2 Atomic functions operating on 32-bit integer Yes values in global memory (Atomic Functions) atomicExch() operating on 32-bit floating Yes point values in global memory (atomicExch()) Atomic functions operating on 32-bit integer Yes values in shared memory (Atomic Functions) atomicExch() operating on 32-bit floating Yes point values in shared memory (atomicExch()) Atomic functions operating on 64-bit integer Yes values in global memory (Atomic Functions) Atomic functions operating on 64-bit integer Yes values in shared memory (Atomic Functions) www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 277 ----------------------- Page 294----------------------- Compute Capabilities Feature Support Compute Capability 3.5, (Unlisted features are supported for all 3.7, 3.0 3.2 5.3 6.x 7.x compute capabilities) 5.0, 5.2 Atomic addition operating on 32-bit floating point values in global and shared memory Yes (atomicAdd()) Atomic addition operating on 64-bit floating point values in global memory and shared No Yes memory (atomicAdd()) Warp vote and ballot functions (Warp Vote Functions) \_\_threadfence\_system() (Memory Fence Functions) \_\_syncthreads\_count(), \_\_syncthreads\_and(), Yes \_\_syncthreads\_or() (Synchronization Functions) Surface functions (Surface Functions) 3D grid of thread blocks Unified Memory Programming Funnel shift (see reference manual) No Yes Dynamic Parallelism No Yes Half-precision floating-point operations: addition, subtraction, multiplication, No Yes comparison, warp shuffle functions, conversion Tensor Core No Yes Table 15 Technical Specifications per Compute Capability Compute Capability Technical Specifications 3.0 3.2 3.5 3.7 5.0 5.2 5.3 6.0 6.1 6.2 7.0 7.5 Maximum number of resident grids per device (Concurrent 16 4 32 16 128 32 16 128 Kernel Execution) Maximum dimensionality of 3 grid of thread blocks Maximum x-dimension of a 31 2 -1 grid of thread blocks Maximum y- or z-dimension 65535 of a grid of thread blocks www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 278 ----------------------- Page 295----------------------- Compute Capabilities Compute Capability Technical Specifications 3.0 3.2 3.5 3.7 5.0 5.2 5.3 6.0 6.1 6.2 7.0 7.5 Maximum dimensionality of 3 thread block Maximum x- or y-dimension 1024 of a block Maximum z-dimension of a 64 block Maximum number of threads 1024 per block Warp size 32 Maximum number of resident 16 32 16 blocks per multiprocessor Maximum number of resident 64 32 warps per multiprocessor Maximum number of resident 2048 1024 threads per multiprocessor Number of 32-bit registers 128 64 K 64 K per multiprocessor K Maximum number of 32-bit 64 K 32 K 64 K 32 K 64 K 32 K 64 K registers per thread block Maximum number of 32-bit 63 255 registers per thread Maximum amount of shared 112 64 96 96 64 96 64 48 KB 64 KB memory per multiprocessor KB KB KB KB KB KB KB Maximum amount of shared 96 64 25 48 KB memory per thread block KB KB Number of shared memory 32 banks Amount of local memory per 512 KB thread Constant memory size 64 KB Cache working set per 4 multiprocessor for constant 8 KB 8 KB KB memory 32 Cache working set per 32 ~ Between 24 or multiprocessor for texture Between 12 KB and 48 KB 128 KB and 48 KB 64 memory KB KB Maximum width for a 1D texture reference bound to a 65536 CUDA array 25 above 48 KB requires dynamic shared memory www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 279 ----------------------- Page 296----------------------- Compute Capabilities Compute Capability Technical Specifications 3.0 3.2 3.5 3.7 5.0 5.2 5.3 6.0 6.1 6.2 7.0 7.5 Maximum width for a 1D 27 texture reference bound to 2 linear memory Maximum width and number of layers for a 1D layered 16384 x 2048 texture reference Maximum width and height for a 2D texture reference 65536 x 65535 bound to a CUDA array Maximum width and height for a 2D texture reference 65000 x 65000 bound to linear memory Maximum width and height for a 2D texture reference 16384 x 16384 bound to a CUDA array supporting texture gather Maximum width, height, and number of layers for a 2D 16384 x 16384 x 2048 layered texture reference Maximum width, height, and depth for a 3D texture 4096 x 4096 x 4096 reference bound to a CUDA array Maximum width (and height) for a cubemap texture 16384 reference Maximum width (and height) and number of layers for a 16384 x 2046 cubemap layered texture reference Maximum number of textures that can be bound to a 256 kernel Maximum width for a 1D surface reference bound to a 65536 CUDA array Maximum width and number of layers for a 1D layered 65536 x 2048 surface reference Maximum width and height for a 2D surface reference 65536 x 32768 bound to a CUDA array Maximum width, height, and number of layers for a 2D 65536 x 32768 x 2048 layered surface reference Maximum width, height, 65536 x 32768 x 2048 and depth for a 3D surface www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 280 ----------------------- Page 297----------------------- Compute Capabilities Compute Capability Technical Specifications 3.0 3.2 3.5 3.7 5.0 5.2 5.3 6.0 6.1 6.2 7.0 7.5 reference bound to a CUDA array Maximum width (and height) for a cubemap surface 32768 reference bound to a CUDA array Maximum width (and height) and number of layers for a 32768 x 2046 cubemap layered surface reference Maximum number of surfaces that can be bound to a 16 kernel Maximum number of 512 million instructions per kernel H.2. Floating-Point Standard All compute devices follow the IEEE 754-2008 standard for binary floating-point arithmetic with the following deviations: ‣ There is no dynamically configurable rounding mode; however, most of the operations support multiple IEEE rounding modes, exposed via device intrinsics; ‣ There is no mechanism for detecting that a floating-point exception has occurred and all operations behave as if the IEEE-754 exceptions are always masked, and deliver the masked response as defined by IEEE-754 if there is an exceptional event; for the same reason, while SNaN encodings are supported, they are not signaling and are handled as quiet; ‣ The result of a single-precision floating-point operation involving one or more input NaNs is the quiet NaN of bit pattern 0x7fffffff; ‣ Double-precision floating-point absolute value and negation are not compliant with IEEE-754 with respect to NaNs; these are passed through unchanged; Code must be compiled with -ftz=false, -prec-div=true, and -prec-sqrt=true to ensure IEEE compliance (this is the default setting; see the nvcc user manual for description of these compilation flags). Regardless of the setting of the compiler flag -ftz, ‣ Atomic single-precision floating-point adds on global memory always operate in flush-to-zero mode, i.e., behave equivalent to FADD.F32.FTZ.RN, ‣ Atomic single-precision floating-point adds on shared memory always operate with denormal support, i.e., behave equivalent to FADD.F32.RN. In accordance to the IEEE-754R standard, if one of the input parameters to fminf(), fmin(), fmaxf(), or fmax() is NaN, but not the other, the result is the non-NaN parameter. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 281 ----------------------- Page 298----------------------- Compute Capabilities The conversion of a floating-point value to an integer value in the case where the floating-point value falls outside the range of the integer format is left undefined by IEEE-754. For compute devices, the behavior is to clamp to the end of the supported range. This is unlike the x86 architecture behavior. The behavior of integer division by zero and integer overflow is left undefined by IEEE-754. For compute devices, there is no mechanism for detecting that such integer operation exceptions have occurred. Integer division by zero yields an unspecified, machine-specific value. http://developer.nvidia.com/content/precision-performance-floating-point-and-ieee-754- compliance-nvidia-gpus includes more information on the floating point accuracy and compliance of NVIDIA GPUs. H.3. Compute Capability 3.x H.3.1. Architecture A multiprocessor consists of: ‣ 192 CUDA cores for arithmetic operations (see Arithmetic Instructions for throughputs of arithmetic operations), ‣ 32 special function units for single-precision floating-point transcendental functions, ‣ 4 warp schedulers. When a multiprocessor is given warps to execute, it first distributes them among the four schedulers. Then, at every instruction issue time, each scheduler issues two independent instructions for one of its assigned warps that is ready to execute, if any. A multiprocessor has a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory. There is an L1 cache for each multiprocessor and an L2 cache shared by all multiprocessors. The L1 cache is used to cache accesses to local memory, including temporary register spills. The L2 cache is used to cache accesses to local and global memory. The cache behavior (e.g., whether reads are cached in both L1 and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load or store instruction. Some devices of compute capability 3.5 and devices of compute capability 3.7 allow opt-in to caching of global memory in both L1 and L2 via compiler options. The same on-chip memory is used for both L1 and shared memory: It can be configured as 48 KB of shared memory and 16 KB of L1 cache or as 16 KB of shared memory www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 282 ----------------------- Page 299----------------------- Compute Capabilities and 48 KB of L1 cache or as 32 KB of shared memory and 32 KB of L1 cache, using cudaFuncSetCacheConfig()/cuFuncSetCacheConfig(): // Device code \_\_global\_\_ void MyKernel() { ... } // Host code // Runtime API // cudaFuncCachePreferShared: shared memory is 48 KB // cudaFuncCachePreferEqual: shared memory is 32 KB // cudaFuncCachePreferL1: shared memory is 16 KB // cudaFuncCachePreferNone: no preference cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared) The default cache configuration is "prefer none," meaning "no preference." If a kernel is configured to have no preference, then it will default to the preference of the current thread/context, which is set using cudaDeviceSetCacheConfig()/cuCtxSetCacheConfig() (see the reference manual for details). If the current thread/context also has no preference (which is again the default setting), then whichever cache configuration was most recently used for any kernel will be the one that is used, unless a different cache configuration is required to launch the kernel (e.g., due to shared memory requirements). The initial configuration is 48 KB of shared memory and 16 KB of L1 cache. Devices of compute capability 3.7 add an additional 64 KB of shared memory to each of the above configurations, yielding 112 KB, 96 KB, and 80 KB shared memory per multiprocessor, respectively. However, the maximum shared memory per thread block remains 48 KB. Applications may query the L2 cache size by checking the l2CacheSize device property (see Device Enumeration). The maximum L2 cache size is 1.5 MB. Each multiprocessor has a read-only data cache of 48 KB to speed up reads from device memory. It accesses this cache either directly (for devices of compute capability 3.5 or 3.7), or via a texture unit that implements the various addressing modes and data filtering mentioned in Texture and Surface Memory. When accessed via the texture unit, the read-only data cache is also referred to as texture cache. H.3.2. Global Memory Global memory accesses for devices of compute capability 3.x are cached in L2 and for devices of compute capability 3.5 or 3.7, may also be cached in the read-only data cache described in the previous section; they are normally not cached in L1. Some devices of compute capability 3.5 and devices of compute capability 3.7 allow opt-in to caching of global memory accesses in L1 via the -Xptxas -dlcm=ca option to nvcc. A cache line is 128 bytes and maps to a 128 byte aligned segment in device memory. Memory accesses that are cached in both L1 and L2 are serviced with 128-byte memory transactions whereas memory accesses that are cached in L2 only are serviced with 32-byte memory transactions. Caching in L2 only can therefore reduce over-fetch, for example, in the case of scattered memory accesses. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 283 ----------------------- Page 300----------------------- Compute Capabilities If the size of the words accessed by each thread is more than 4 bytes, a memory request by a warp is first split into separate 128-byte memory requests that are issued independently: ‣ Two memory requests, one for each half-warp, if the size is 8 bytes, ‣ Four memory requests, one for each quarter-warp, if the size is 16 bytes. Each memory request is then broken down into cache line requests that are issued independently. A cache line request is serviced at the throughput of L1 or L2 cache in case of a cache hit, or at the throughput of device memory, otherwise. Note that threads can access any words in any order, including the same words. If a non-atomic instruction executed by a warp writes to the same location in global memory for more than one of the threads of the warp, only one thread performs a write and which thread does it is undefined. Data that is read-only for the entire lifetime of the kernel can also be cached in the read- only data cache described in the previous section by reading it using the \_\_ldg() function (see Read-Only Data Cache Load Function). When the compiler detects that the read-only condition is satisfied for some data, it will use \_\_ldg() to read it. The compiler might not always be able to detect that the read-only condition is satisfied for some data. Marking pointers used for loading such data with both the const and \_\_restrict\_\_ qualifiers increases the likelihood that the compiler will detect the read- only condition. Figure 18 shows some examples of global memory accesses and corresponding memory transactions. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 284 ----------------------- Page 301----------------------- Compute Capabilities Figure 18 Examples of Global Memory Accesses Examples of Global Memory Accesses by a Warp, 4-Byte Word per Thread, and Associated Memory Transactions for Compute Capabilities 3.x and Beyond H.3.3. Shared Memory Shared memory has 32 banks with two addressing modes that are described below. The addressing mode can be queried using cudaDeviceGetSharedMemConfig() and set using cudaDeviceSetSharedMemConfig() (see reference manual for more details). Each bank has a bandwidth of 64 bits per clock cycle. Figure 19 shows some examples of strided access. Figure 20 shows some examples of memory read accesses that involve the broadcast mechanism. 64-Bit Mode Successive 64-bit words map to successive banks. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 285 ----------------------- Page 302----------------------- Compute Capabilities A shared memory request for a warp does not generate a bank conflict between two threads that access any sub-word within the same 64-bit word (even though the addresses of the two sub-words fall in the same bank): In that case, for read accesses, the 64-bit word is broadcast to the requesting threads and for write accesses, each sub-word is written by only one of the threads (which thread performs the write is undefined). 32-Bit Mode Successive 32-bit words map to successive banks. A shared memory request for a warp does not generate a bank conflict between two threads that access any sub-word within the same 32-bit word or within two 32-bit words whose indices i and j are in the same 64-word aligned segment (i.e., a segment whose first index is a multiple of 64) and such that j=i+32 (even though the addresses of the two sub-words fall in the same bank): In that case, for read accesses, the 32-bit words are broadcast to the requesting threads and for write accesses, each sub-word is written by only one of the threads (which thread performs the write is undefined). H.4. Compute Capability 5.x H.4.1. Architecture A multiprocessor consists of: ‣ 128 CUDA cores for arithmetic operations (see Arithmetic Instructions for throughputs of arithmetic operations), ‣ 32 special function units for single-precision floating-point transcendental functions, ‣ 4 warp schedulers. When a multiprocessor is given warps to execute, it first distributes them among the four schedulers. Then, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps that is ready to execute, if any. A multiprocessor has: ‣ a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory, ‣ a unified L1/texture cache of 24 KB used to cache reads from global memory, ‣ 64 KB of shared memory for devices of compute capability 5.0 or 96 KB of shared memory for devices of compute capability 5.2. The unified L1/texture cache is also used by the texture unit that implements the various addressing modes and data filtering mentioned in Texture and Surface Memory. There is also an L2 cache shared by all multiprocessors that is used to cache accesses to local or global memory, including temporary register spills. Applications may query the L2 cache size by checking the l2CacheSize device property (see Device Enumeration). www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 286 ----------------------- Page 303----------------------- Compute Capabilities The cache behavior (e.g., whether reads are cached in both the unified L1/texture cache and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load instruction. H.4.2. Global Memory Global memory accesses are always cached in L2 and caching in L2 behaves in the same way as for devices of compute capability 3.x (see Global Memory). Data that is read-only for the entire lifetime of the kernel can also be cached in the unified L1/texture cache described in the previous section by reading it using the \_\_ldg() function (see Read-Only Data Cache Load Function). When the compiler detects that the read-only condition is satisfied for some data, it will use \_\_ldg() to read it. The compiler might not always be able to detect that the read-only condition is satisfied for some data. Marking pointers used for loading such data with both the const and \_\_restrict\_\_ qualifiers increases the likelihood that the compiler will detect the read-only condition. Data that is not read-only for the entire lifetime of the kernel cannot be cached in the unified L1/texture cache for devices of compute capability 5.0. For devices of compute capability 5.2, it is, by default, not cached in the unified L1/texture cache, but caching may be enabled using the following mechanisms: ‣ Perform the read using inline assembly with the appropriate modifier as described in the PTX reference manual; ‣ Compile with the -Xptxas -dlcm=ca compilation flag, in which case all reads are cached, except reads that are performed using inline assembly with a modifier that disables caching; ‣ Compile with the -Xptxas -fscm=ca compilation flag, in which case all reads are cached, including reads that are performed using inline assembly regardless of the modifier used. When caching is enabled using one of the three mechanisms listed above, devices of compute capability 5.2 will cache global memory reads in the unified L1/texture cache for all kernel launches except for the kernel launches for which thread blocks consume too much of the multiprocessor's register file. These exceptions are reported by the profiler. H.4.3. Shared Memory Shared memory has 32 banks that are organized such that successive 32-bit words map to successive banks. Each bank has a bandwidth of 32 bits per clock cycle. A shared memory request for a warp does not generate a bank conflict between two threads that access any address within the same 32-bit word (even though the two addresses fall in the same bank): In that case, for read accesses, the word is broadcast to the requesting threads and for write accesses, each address is written by only one of the threads (which thread performs the write is undefined). Figure 19 shows some examples of strided access. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 287 ----------------------- Page 304----------------------- Compute Capabilities Figure 20 shows some examples of memory read accesses that involve the broadcast mechanism. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 288 ----------------------- Page 305----------------------- Compute Capabilities Threads: Banks: Threads: Banks: Threads: Banks: 0 0 0 0 0 0 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4 5 5 5 5 5 5 6 6 6 6 6 6 7 7 7 7 7 7 8 8 8 8 8 8 9 9 9 9 9 9 10 10 10 10 10 10 11 11 11 11 11 11 12 12 12 12 12 12 13 13 13 13 13 13 14 14 14 14 14 14 15 15 15 15 15 15 16 16 16 16 16 16 17 17 17 17 17 17 18 18 18 18 18 18 19 19 19 19 19 19 2 0 2 0 2 0 2 0 2 0 2 0 2 1 2 1 2 1 2 1 2 1 2 1 22 22 22 22 22 22 2 3 2 3 2 3 2 3 2 3 2 3 24 24 24 24 24 24 2 5 2 5 2 5 2 5 2 5 2 5 26 26 2 6 2 6 2 6 2 6 2 7 2 7 2 7 2 7 2 7 2 7 2 8 2 8 2 8 2 8 2 8 2 8 2 9 2 9 2 9 2 9 2 9 2 9 30 30 30 30 30 30 3 1 3 1 3 1 3 1 3 1 3 1 Left Linear addressing with a stride of one 32-bit word (no bank conflict). Middle Linear addressing with a stride of two 32-bit words (two-way bank conflict). Right Linear addressing with a stride of three 32-bit words (no bank conflict). Figure 19 Strided Shared Memory Accesses Examples for devices of compute capability 3.x (in 32-bit mode) or compute capability 5.x and 6.x www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 289 ----------------------- Page 306----------------------- Compute Capabilities Threads: Banks: Threads: Banks: Threads: Banks: 0 0 0 0 0 0 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4 5 5 5 5 5 5 6 6 6 6 6 6 7 7 7 7 7 7 8 8 8 8 8 8 9 9 9 9 9 9 10 10 10 10 10 10 11 11 11 11 11 11 12 12 12 12 12 12 13 13 13 13 13 13 14 14 14 14 14 14 15 15 15 15 15 15 16 16 16 16 16 16 17 17 17 17 17 17 18 18 18 18 18 18 19 19 19 19 19 19 2 0 2 0 2 0 2 0 2 0 2 0 2 1 2 1 2 1 2 1 2 1 2 1 22 22 22 22 22 22 2 3 2 3 2 3 2 3 2 3 2 3 24 24 24 24 24 24 2 5 2 5 2 5 2 5 2 5 2 5 26 26 2 6 2 6 2 6 2 6 2 7 2 7 2 7 2 7 2 7 2 7 2 8 2 8 2 8 2 8 2 8 2 8 2 9 2 9 2 9 2 9 2 9 2 9 30 30 30 30 30 30 3 1 3 1 3 1 3 1 3 1 3 1 Left Conflict-free access via random permutation. Middle Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5. Right Conflict-free broadcast access (threads access the same word within a bank). Figure 20 Irregular Shared Memory Accesses Examples for devices of compute capability 3.x, 5.x, or 6.x. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 290 ----------------------- Page 307----------------------- Compute Capabilities H.5. Compute Capability 6.x H.5.1. Architecture A multiprocessor consists of: ‣ 64 (compute capablity 6.0) or 128 (6.1 and 6.2) CUDA cores for arithmetic operations, ‣ 16 (6.0) or 32 (6.1 and 6.2) special function units for single-precision floating-point transcendental functions, ‣ 2 (6.0) or 4 (6.1 and 6.2) warp schedulers. When a multiprocessor is given warps to execute, it first distributes them among its schedulers. Then, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps that is ready to execute, if any. A multiprocessor has: ‣ a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory, ‣ a unified L1/texture cache for reads from global memory of size 24 KB (6.0 and 6.2) or 48 KB (6.1), ‣ a shared memory of size 64 KB (6.0 and 6.2) or 96 KB (6.1). The unified L1/texture cache is also used by the texture unit that implements the various addressing modes and data filtering mentioned in Texture and Surface Memory. There is also an L2 cache shared by all multiprocessors that is used to cache accesses to local or global memory, including temporary register spills. Applications may query the L2 cache size by checking the l2CacheSize device property (see Device Enumeration). The cache behavior (e.g., whether reads are cached in both the unified L1/texture cache and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load instruction. H.5.2. Global Memory Global memory behaves the same way as devices of compute capability 5.x (See Global Memory). H.5.3. Shared Memory Shared memory behaves the same way as devices of compute capability 5.x (See Shared Memory). www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 291 ----------------------- Page 308----------------------- Compute Capabilities H.6. Compute Capability 7.x H.6.1. Architecture A multiprocessor consists of: ‣ 64 FP32 cores for single-precision arithmetic operations, ‣ 32 FP64 cores for double-precision arithmetic operations, 26 ‣ 64 INT32 cores for integer math, ‣ 8 mixed-precision Tensor Cores for deep learning matrix arithmetic ‣ 16 special function units for single-precision floating-point transcendental functions, ‣ 4 warp schedulers. A multiprocessor statically distributes its warps among its schedulers. Then, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps that is ready to execute, if any. A multiprocessor has: ‣ a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory, ‣ a unified data cache and shared memory with a total size of 128 KB (Volta) or 96 KB (Turing). Shared memory is partitioned out of unified data cache, and can be configured to various sizes (See Shared Memory.) The remaining data cache serves as an L1 cache and is also used by the texture unit that implements the various addressing and data filtering modes mentioned in Texture and Surface Memory. H.6.2. Independent Thread Scheduling The Volta architecture introduces Independent Thread Scheduling among threads in a warp, enabling intra-warp synchronization patterns previously unavailable and simplifying code changes when porting CPU code. However, this can lead to a rather different set of threads participating in the executed code than intended if the developer made assumptions about warp-synchronicity of previous hardware architectures. Below are code patterns of concern and suggested corrective actions for Volta-safe code. 1. For applications using warp intrinsics (\_\_shfl\*, \_\_any, \_\_all, \_\_ballot), it is necessary that developers port their code to the new, safe, synchronizing counterpart, with the \*\_sync suffix. The new warp intrinsics take in a mask of threads that explicitly define which lanes (threads of a warp) must participate in the warp intrinsic. See Warp Vote Functions and Warp Shuffle Functions for details. 26 2 FP64 cores for double-precision arithmetic operations for devices of compute capabilities 7.5 www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 292 ----------------------- Page 309----------------------- Compute Capabilities Since the intrinsics are available with CUDA 9.0+, (if necessary) code can be executed conditionally with the following preprocessor macro: #if defined(CUDART\_VERSION) && CUDART\_VERSION >= 9000 // \*\_sync intrinsic #endif These intrinsics are available on all architectures, not just Volta or Turing, and in most cases a single code-base will suffice for all architectures. Note, however, that for Pascal and earlier architectures, all threads in mask must execute the same warp intrinsic instruction in convergence, and the union of all values in mask must be equal to the warp's active mask. The following code pattern is valid on Volta, but not on Pascal or earlier architectures. if (tid % warpSize < 16) { ... float swapped = \_\_shfl\_xor\_sync(0xffffffff, val, 16); ... } else { ... float swapped = \_\_shfl\_xor\_sync(0xffffffff, val, 16); ... } The replacement for \_\_ballot(1) is \_\_activemask(). Note that threads within a warp can diverge even within a single code path. As a result, \_\_activemask() and \_\_ballot(1) may return only a subset of the threads on the current code path. The following invalid code example sets bit i of output to 1 when data[i] is greater than threshold. \_\_activemask() is used in an attempt to enable cases where dataLen is not a multiple of 32. // Sets bit in output[] to 1 if the correspond element in data[i] // is greater than ‘threshold’, using 32 threads in a warp. for(int i=warpLane; i<dataLen; i+=warpSize) { unsigned active = \_\_activemask(); unsigned bitPack = \_\_ballot\_sync(active, data[i] > threshold); if (warpLane == 0) output[i/32] = bitPack; } This code is invalid because CUDA does not guarantee that the warp will diverge ONLY at the loop condition. When divergence happens for other reasons, conflicting results will be computed for the same 32-bit output element by different subsets of threads in the warp. A correct code might use a non-divergent loop condition together with \_\_ballot\_sync() to safely enumerate the set of threads in the warp participating in the threshold calculation as follows. for(int i=warpLane; i-warpLane<dataLen; i+=warpSize) { unsigned active = \_\_ballot\_sync(0xFFFFFFFF, i < dataLen); if (i < dataLen) { unsigned bitPack = \_\_ballot\_sync(active, data[i] > threshold); if (warpLane == 0) output[i/32] = bitPack; } } Discovery Pattern demonstrates a valid usecase for \_\_activemask(). www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 293 ----------------------- Page 310----------------------- Compute Capabilities 2. If applications have warp-synchronous codes, they will need to insert the new \_\_syncwarp() warp-wide barrier synchronization instruction between any steps where data is exchanged between threads via global or shared memory. Assumptions that code is executed in lockstep or that reads/writes from separate threads are visible across a warp without synchronization are invalid. float s\_buff[tid] = val; \_\_syncthreads(); // Inter-warp reduction for (int i=BSIZE/2; i>32; i/=2){ s\_buff[tid] += s\_buff[tid+i]; \_\_syncthreads(); } // Intra-warp reduction // Butterfly reduction simplifies syncwarp mask if (tid < 32) { float temp; temp = s\_buff[tid^16]; \_\_syncwarp(); s\_buff[tid] += temp; \_\_syncwarp(); temp = s\_buff[tid^ 8]; \_\_syncwarp(); s\_buff[tid] += temp; \_\_syncwarp(); temp = s\_buff[tid^ 4]; \_\_syncwarp(); s\_buff[tid] += temp; \_\_syncwarp(); temp = s\_buff[tid^ 2]; \_\_syncwarp(); s\_buff[tid] += temp; \_\_syncwarp(); } if (tid == 0) { \*output = s\_buff[0] + s\_buff[1]; } \_\_syncthreads(); 3. Although \_\_syncthreads() has been consistently documented as synchronizing all threads in the thread block, Pascal and prior architectures could only enforce synchronization at the warp level. In certain cases, this allowed a barrier to succeed without being executed by every thread as long as at least some thread in every warp reached the barrier. Starting with Volta, the CUDA built-in \_\_syncthreads() and PTX instruction bar.sync (and their derivatives) are enforced per thread and thus will not succeed until reached by all non-exited threads in the block. Code exploiting the previous behavior will likely deadlock and must be modified to ensure that all non-exited threads reach the barrier. The racecheck and synccheck tools provided by cuda-memcheck can aid in locating violations of points 2 and 3. To aid migration while implementing the above-mentioned corrective actions, developers can opt-in to the Pascal scheduling model that does not support independent thread scheduling. See Application Compatibility for details. H.6.3. Global Memory Global memory behaves the same way as devices of compute capability 5.x (See Global Memory). www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 294 ----------------------- Page 311----------------------- Compute Capabilities H.6.4. Shared Memory Similar to the Kepler architecture, the amount of the unified data cache reserved for shared memory is configurable on a per kernel basis. For the Volta architecture (compute capability 7.0), the unified data cache has a size of 128 KB, and the shared memory capacity can be set to 0, 8, 16, 32, 64 or 96 KB. For the Turing architecture (compute capability 7.5), the unified data cache has a size of 96 KB, and the shared memory capacity can be set to either 32 KB or 64 KB. Unlike Kepler, the driver automatically configures the shared memory capacity for each kernel to avoid shared memory occupancy bottlenecks while also allowing concurrent execution with already launched kernels where possible. In most cases, the driver's default behavior should provide optimal performance. Because the driver is not always aware of the full workload, it is sometimes useful for applications to provide additional hints regarding the desired shared memory configuration. For example, a kernel with little or no shared memory use may request a larger carveout in order to encourage concurrent execution with later kernels that require more shared memory. The new cudaFuncSetAttribute() API allows applications to set a preferred shared memory capacity, or carveout, as a percentage of the maximum supported shared memory capacity (96 KB for Volta, and 64 KB for Turing). cudaFuncSetAttribute() relaxes enforcement of the preferred shared capacity compared to the legacy cudaFuncSetCacheConfig() API introduced with Kepler. The legacy API treated shared memory capacities as hard requirements for kernel launch. As a result, interleaving kernels with different shared memory configurations would needlessly serialize launches behind shared memory reconfigurations. With the new API, the carveout is treated as a hint. The driver may choose a different configuration if required to execute the function or to avoid thrashing. // Device code \_\_global\_\_ void MyKernel(...) { \_\_shared\_\_ float buffer[BLOCK\_DIM]; ... } // Host code int carveout = 50; // prefer shared memory capacity 50% of maximum // Named Carevout Values: // carveout = cudaSharedmemCarveoutDefault; // (-1) // carveout = cudaSharedmemCarveoutMaxL1; // (0) // carveout = cudaSharedmemCarveoutMaxShared; // (100) cudaFuncSetAttribute(MyKernel, cudaFuncAttributePreferredSharedMemoryCarveout, carveout); MyKernel <<<gridDim, BLOCK\_DIM>>> (...); In addition to an integer percentage, several convenience enums are provided as listed in the code comments above. Where a chosen integer percentage does not map exactly to a supported capacity (SM 7.0 devices support shared capacities of 0, 8, 16, 32, 64, or 96 KB), the next larger capacity is used. For instance, in the example above, 50% of the 96 KB maximum is 48 KB, which is not a supported shared memory capacity. Thus, the preference is rounded up to 64 KB. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 295 ----------------------- Page 312----------------------- Compute Capabilities Compute capability 7.x devices allow a single thread block to address the full capacity of shared memory: 96 KB on Volta, 64 KB on Turing. Kernels relying on shared memory allocations over 48 KB per block are architecture-specific, as such they must use dynamic shared memory (rather than statically sized arrays) and require an explicit opt-in using cudaFuncSetAttribute() as follows. // Device code \_\_global\_\_ void MyKernel(...) { ... } // Host code int maxbytes = 98304; // 96 KB cudaFuncSetAttribute(MyKernel, cudaFuncAttributeMaxDynamicSharedMemorySize, maxbytes); MyKernel <<<gridDim, blockDim>>> (...); Otherwise, shared memory behaves the same way as devices of compute capability 5.x (See Shared Memory). www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 296 ----------------------- Page 313----------------------- Appendix I. DRIVER API This appendix assumes knowledge of the concepts described in CUDA Runtime. The driver API is implemented in the cuda dynamic library (cuda.dll or cuda.so) which is copied on the system during the installation of the device driver. All its entry points are prefixed with cu. It is a handle-based, imperative API: Most objects are referenced by opaque handles that may be specified to functions to manipulate the objects. The objects available in the driver API are summarized in Table 16. Table 16 Objects Available in the CUDA Driver API Object Handle Description Device CUdevice CUDA-enabled device Context CUcontext Roughly equivalent to a CPU process Module CUmodule Roughly equivalent to a dynamic library Function CUfunction Kernel Heap memory CUdeviceptr Pointer to device memory CUDA array CUarray Opaque container for one-dimensional or two- dimensional data on the device, readable via texture or surface references Texture reference CUtexref Object that describes how to interpret texture memory data Surface reference CUsurfref Object that describes how to read or write CUDA arrays Stream CUstream Object that describes a CUDA stream Event CUevent Object that describes a CUDA event The driver API must be initialized with cuInit() before any function from the driver API is called. A CUDA context must then be created that is attached to a specific device and made current to the calling host thread as detailed in Context. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 297 ----------------------- Page 314----------------------- Driver API Within a CUDA context, kernels are explicitly loaded as PTX or binary objects by the host code as described in Module. Kernels written in C++ must therefore be compiled separately into PTX or binary objects. Kernels are launched using API entry points as described in Kernel Execution. Any application that wants to run on future device architectures must load PTX, not binary code. This is because binary code is architecture-specific and therefore incompatible with future architectures, whereas PTX code is compiled to binary code at load time by the device driver. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 298 ----------------------- Page 315----------------------- Driver API Here is the host code of the sample from Kernels written using the driver API: int main() { int N = ...; size\_t size = N \* sizeof(float); // Allocate input vectors h\_A and h\_B in host memory float\* h\_A = (float\*)malloc(size); float\* h\_B = (float\*)malloc(size); // Initialize input vectors ... // Initialize cuInit(0); // Get number of devices supporting CUDA int deviceCount = 0; cuDeviceGetCount(&deviceCount); if (deviceCount == 0) { printf("There is no device supporting CUDA.\n"); exit (0); } // Get handle for device 0 CUdevice cuDevice; cuDeviceGet(&cuDevice, 0); // Create context CUcontext cuContext; cuCtxCreate(&cuContext, 0, cuDevice); // Create module from binary file CUmodule cuModule; cuModuleLoad(&cuModule, "VecAdd.ptx"); // Allocate vectors in device memory CUdeviceptr d\_A; cuMemAlloc(&d\_A, size); CUdeviceptr d\_B; cuMemAlloc(&d\_B, size); CUdeviceptr d\_C; cuMemAlloc(&d\_C, size); // Copy vectors from host memory to device memory cuMemcpyHtoD(d\_A, h\_A, size); cuMemcpyHtoD(d\_B, h\_B, size); // Get function handle from module CUfunction vecAdd; cuModuleGetFunction(&vecAdd, cuModule, "VecAdd"); // Invoke kernel int threadsPerBlock = 256; int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock; void\* args[] = { &d\_A, &d\_B, &d\_C, &N }; cuLaunchKernel(vecAdd, blocksPerGrid, 1, 1, threadsPerBlock, 1, 1, 0, 0, args, 0); ... } Full code can be found in the vectorAddDrv CUDA sample. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 299 ----------------------- Page 316----------------------- Driver API I.1. Context A CUDA context is analogous to a CPU process. All resources and actions performed within the driver API are encapsulated inside a CUDA context, and the system automatically cleans up these resources when the context is destroyed. Besides objects such as modules and texture or surface references, each context has its own distinct address space. As a result, CUdeviceptr values from different contexts reference different memory locations. A host thread may have only one device context current at a time. When a context is created with cuCtxCreate(), it is made current to the calling host thread. CUDA functions that operate in a context (most functions that do not involve device enumeration or context management) will return CUDA\_ERROR\_INVALID\_CONTEXT if a valid context is not current to the thread. Each host thread has a stack of current contexts. cuCtxCreate() pushes the new context onto the top of the stack. cuCtxPopCurrent() may be called to detach the context from the host thread. The context is then "floating" and may be pushed as the current context for any host thread. cuCtxPopCurrent() also restores the previous current context, if any. A usage count is also maintained for each context. cuCtxCreate() creates a context with a usage count of 1. cuCtxAttach() increments the usage count and cuCtxDetach() decrements it. A context is destroyed when the usage count goes to 0 when calling cuCtxDetach() or cuCtxDestroy(). The driver API is interoperable with the runtime and it is possible to access the primary context (see Initialization) managed by the runtime from the driver API via cuDevicePrimaryCtxRetain(). Usage count facilitates interoperability between third party authored code operating in the same context. For example, if three libraries are loaded to use the same context, each library would call cuCtxAttach() to increment the usage count and cuCtxDetach() to decrement the usage count when the library is done using the context. For most libraries, it is expected that the application will have created a context before loading or initializing the library; that way, the application can create the context using its own heuristics, and the library simply operates on the context handed to it. Libraries that wish to create their own contexts - unbeknownst to their API clients who may or may not have created contexts of their own - would use cuCtxPushCurrent() and cuCtxPopCurrent() as illustrated in Figure 21. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 300 ----------------------- Page 317----------------------- Driver API Library Initialization Call Initialize cuCtxCreate() context cuCtxPopCurrent() Library Call Use cuCtxPushCurrent() context cuCtxPopCurrent() Figure 21 Library Context Management I.2. Module Modules are dynamically loadable packages of device code and data, akin to DLLs in Windows, that are output by nvcc (see Compilation with NVCC). The names for all symbols, including functions, global variables, and texture or surface references, are maintained at module scope so that modules written by independent third parties may interoperate in the same CUDA context. This code sample loads a module and retrieves a handle to some kernel: CUmodule cuModule; cuModuleLoad(&cuModule, "myModule.ptx"); CUfunction myKernel; cuModuleGetFunction(&myKernel, cuModule, "MyKernel"); This code sample compiles and loads a new module from PTX code and parses compilation errors: #define BUFFER\_SIZE 8192 CUmodule cuModule; CUjit\_option options[3]; void\* values[3]; char\* PTXCode = "some PTX code"; char error\_log[BUFFER\_SIZE]; int err; options[0] = CU\_JIT\_ERROR\_LOG\_BUFFER; values[0] = (void\*)error\_log; options[1] = CU\_JIT\_ERROR\_LOG\_BUFFER\_SIZE\_BYTES; values[1] = (void\*)BUFFER\_SIZE; options[2] = CU\_JIT\_TARGET\_FROM\_CUCONTEXT; values[2] = 0; err = cuModuleLoadDataEx(&cuModule, PTXCode, 3, options, values); if (err != CUDA\_SUCCESS) printf("Link error:\n%s\n", error\_log); www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 301 ----------------------- Page 318----------------------- Driver API This code sample compiles, links, and loads a new module from multiple PTX codes and parses link and compilation errors: #define BUFFER\_SIZE 8192 CUmodule cuModule; CUjit\_option options[6]; void\* values[6]; float walltime; char error\_log[BUFFER\_SIZE], info\_log[BUFFER\_SIZE]; char\* PTXCode0 = "some PTX code"; char\* PTXCode1 = "some other PTX code"; CUlinkState linkState; int err; void\* cubin; size\_t cubinSize; options[0] = CU\_JIT\_WALL\_TIME; values[0] = (void\*)&walltime; options[1] = CU\_JIT\_INFO\_LOG\_BUFFER; values[1] = (void\*)info\_log; options[2] = CU\_JIT\_INFO\_LOG\_BUFFER\_SIZE\_BYTES; values[2] = (void\*)BUFFER\_SIZE; options[3] = CU\_JIT\_ERROR\_LOG\_BUFFER; values[3] = (void\*)error\_log; options[4] = CU\_JIT\_ERROR\_LOG\_BUFFER\_SIZE\_BYTES; values[4] = (void\*)BUFFER\_SIZE; options[5] = CU\_JIT\_LOG\_VERBOSE; values[5] = (void\*)1; cuLinkCreate(6, options, values, &linkState); err = cuLinkAddData(linkState, CU\_JIT\_INPUT\_PTX, (void\*)PTXCode0, strlen(PTXCode0) + 1, 0, 0, 0, 0); if (err != CUDA\_SUCCESS) printf("Link error:\n%s\n", error\_log); err = cuLinkAddData(linkState, CU\_JIT\_INPUT\_PTX, (void\*)PTXCode1, strlen(PTXCode1) + 1, 0, 0, 0, 0); if (err != CUDA\_SUCCESS) printf("Link error:\n%s\n", error\_log); cuLinkComplete(linkState, &cubin, &cubinSize); printf("Link completed in %fms. Linker Output:\n%s\n", walltime, info\_log); cuModuleLoadData(cuModule, cubin); cuLinkDestroy(linkState); Full code can be found in the ptxjit CUDA sample. I.3. Kernel Execution cuLaunchKernel() launches a kernel with a given execution configuration. Parameters are passed either as an array of pointers (next to last parameter of cuLaunchKernel()) where the nth pointer corresponds to the nth parameter and points to a region of memory from which the parameter is copied, or as one of the extra options (last parameter of cuLaunchKernel()). When parameters are passed as an extra option (the CU\_LAUNCH\_PARAM\_BUFFER\_POINTER option), they are passed as a pointer to a single buffer where parameters are assumed to be properly offset with respect to each other by matching the alignment requirement for each parameter type in device code. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 302 ----------------------- Page 319----------------------- Driver API Alignment requirements in device code for the built-in vector types are listed in Table 4. For all other basic types, the alignment requirement in device code matches the alignment requirement in host code and can therefore be obtained using \_\_alignof(). The only exception is when the host compiler aligns double and long long (and long on a 64-bit system) on a one-word boundary instead of a two-word boundary (for example, using gcc's compilation flag -mno-align-double) since in device code these types are always aligned on a two-word boundary. CUdeviceptr is an integer, but represents a pointer, so its alignment requirement is \_\_alignof(void\*). The following code sample uses a macro (ALIGN\_UP()) to adjust the offset of each parameter to meet its alignment requirement and another macro (ADD\_TO\_PARAM\_BUFFER()) to add each parameter to the parameter buffer passed to the CU\_LAUNCH\_PARAM\_BUFFER\_POINTER option. #define ALIGN\_UP(offset, alignment) \ (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1) char paramBuffer[1024]; size\_t paramBufferSize = 0; #define ADD\_TO\_PARAM\_BUFFER(value, alignment) \ do { \ paramBufferSize = ALIGN\_UP(paramBufferSize, alignment); \ memcpy(paramBuffer + paramBufferSize, \ &(value), sizeof(value)); \ paramBufferSize += sizeof(value); \ } while (0) int i; ADD\_TO\_PARAM\_BUFFER(i, \_\_alignof(i)); float4 f4; ADD\_TO\_PARAM\_BUFFER(f4, 16); // float4's alignment is 16 char c; ADD\_TO\_PARAM\_BUFFER(c, \_\_alignof(c)); float f; ADD\_TO\_PARAM\_BUFFER(f, \_\_alignof(f)); CUdeviceptr devPtr; ADD\_TO\_PARAM\_BUFFER(devPtr, \_\_alignof(devPtr)); float2 f2; ADD\_TO\_PARAM\_BUFFER(f2, 8); // float2's alignment is 8 void\* extra[] = { CU\_LAUNCH\_PARAM\_BUFFER\_POINTER, paramBuffer, CU\_LAUNCH\_PARAM\_BUFFER\_SIZE, &paramBufferSize, CU\_LAUNCH\_PARAM\_END }; cuLaunchKernel(cuFunction, blockWidth, blockHeight, blockDepth, gridWidth, gridHeight, gridDepth, 0, 0, 0, extra); The alignment requirement of a structure is equal to the maximum of the alignment requirements of its fields. The alignment requirement of a structure that contains built- in vector types, CUdeviceptr, or non-aligned double and long long, might therefore differ between device code and host code. Such a structure might also be padded differently. The following structure, for example, is not padded at all in host code, but it www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 303 ----------------------- Page 320----------------------- Driver API is padded in device code with 12 bytes after field f since the alignment requirement for field f4 is 16. typedef struct { float f; float4 f4; } myStruct; I.4. Interoperability between Runtime and Driver APIs An application can mix runtime API code with driver API code. If a context is created and made current via the driver API, subsequent runtime calls will pick up this context instead of creating a new one. If the runtime is initialized (implicitly as mentioned in CUDA Runtime), cuCtxGetCurrent() can be used to retrieve the context created during initialization. This context can be used by subsequent driver API calls. The implicitly created context from the runtime is called the primary context (see Initialization). It can be managed from the driver API with the Primary Context Management functions. Device memory can be allocated and freed using either API. CUdeviceptr can be cast to regular pointers and vice-versa: CUdeviceptr devPtr; float\* d\_data; // Allocation using driver API cuMemAlloc(&devPtr, size); d\_data = (float\*)devPtr; // Allocation using runtime API cudaMalloc(&d\_data, size); devPtr = (CUdeviceptr)d\_data; In particular, this means that applications written using the driver API can invoke libraries written using the runtime API (such as cuFFT, cuBLAS, ...). All functions from the device and version management sections of the reference manual can be used interchangeably. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 304 ----------------------- Page 321----------------------- Appendix J. CUDA ENVIRONMENT VARIABLES Environment variables related to the Multi-Process Service are documented in the Multi- Process Service section of the GPU Deployment and Management guide. Table 17 CUDA Environment Variables Category Variable Values Description Device CUDA\_VISIBLE\_DEVICES A comma- GPU identifiers are given as integer Enumeration separated indices or as UUID strings. GPU and Properties sequence of UUID strings should follow the same GPU identifiers format as given by nvidia-smi, such as GPU-8932f937-d72c-4106- c12f-20bd9faed9f6. However, for convenience, abbreviated forms are allowed; simply specify enough digits from the beginning of the GPU UUID to uniquely identify that GPU in the target system. For example, CUDA\_VISIBLE\_DEVICES=GPU-8932f937 may be a valid way to refer to the above GPU UUID, assuming no other GPU in the system shares this prefix. Only the devices whose index is present in the sequence are visible to CUDA applications and they are enumerated in the order of the sequence. If one of the indices is invalid, only the devices whose index precedes the invalid index are visible to CUDA applications. For example, setting CUDA\_VISIBLE\_DEVICES to 2,1 causes device 0 to be invisible and device 2 to be enumerated before device 1. Setting CUDA\_VISIBLE\_DEVICES to 0,2,-1,1 causes devices 0 and 2 to be visible and device 1 to be invisible. CUDA\_MANAGED\_ 0 or 1 (default Forces the driver to place all managed FORCE\_DEVICE\_ALLOC is 0) allocations in device memory. CUDA\_DEVICE\_ORDER FASTEST\_FIRST, FASTEST\_FIRST causes CUDA to guess PCI\_BUS\_ID, which device is fastest using a simple www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 305 ----------------------- Page 322----------------------- CUDA Environment Variables Category Variable Values Description (default is heuristic, and make that device 0, leaving FASTEST\_FIRST) the order of the rest of the devices unspecified. PCI\_BUS\_ID orders devices by PCI bus ID in ascending order. Compilation CUDA\_CACHE\_DISABLE 0 or 1 (default Disables caching (when set to 1) or is 0) enables caching (when set to 0) for just- in-time-compilation. When disabled, no binary code is added to or retrieved from the cache. CUDA\_CACHE\_PATH filepath Specifies the folder where the just-in- time compiler caches binary codes; the default values are: ‣ on Windows, %APPDATA%\NVIDIA \ComputeCache, ‣ on MacOS, $HOME/Library/ Application\ Support/NVIDIA/ ComputeCache, ‣ on Linux, ~/.nv/ComputeCache CUDA\_CACHE\_MAXSIZE integer (default Specifies the size in bytes of the cache is 268435456 used by the just-in-time compiler. Binary (256 MiB) and codes whose size exceeds the cache size maximum is are not cached. Older binary codes are 4294967296 (4 evicted from the cache to make room for GiB)) newer binary codes if needed. CUDA\_FORCE\_PTX\_JIT 0 or 1 (default When set to 1, forces the device driver is 0) to ignore any binary code embedded in an application (see Application Compatibility) and to just-in-time compile embedded PTX code instead. If a kernel does not have embedded PTX code, it will fail to load. This environment variable can be used to validate that PTX code is embedded in an application and that its just-in-time compilation works as expected to guarantee application forward compatibility with future architectures (see Just-in-Time Compilation). Execution CUDA\_LAUNCH\_ 0 or 1 (default Disables (when set to 1) or enables (when BLOCKING is 0) set to 0) asynchronous kernel launches. CUDA\_DEVICE\_MAX\_ 1 to 32 (default Sets the number of compute and copy CONNECTIONS is 8) engine concurrent connections (work queues) from the host to each device of compute capability 3.5 and above. CUDA\_AUTO\_BOOST 0 or 1 Overrides the autoboost behavior set by the --auto-boost-default option of nvidia-smi. If an application requests via this environment variable a behavior that is different from nvidia-smi's, its request is honored if there is no other application currently running on the www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 306 ----------------------- Page 323----------------------- CUDA Environment Variables Category Variable Values Description same GPU that successfully requested a different behavior, otherwise it is ignored. CUDA\_ENABLE\_CRC\_ 0 or 1 (default When set to 1, commands and data sent CHECK is 0) to the GPU are error-checked to ensure they are not corrupted. cuda-gdb (on CUDA\_DEVICE\_WAITS\_ 0 or 1 (default When set to 1, a CUDA application will Mac and Linux ON\_EXCEPTION is 0) halt when a device exception occurs, platforms) allowing a debugger to be attached for further debugging. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 307 ----------------------- Page 324----------------------- Appendix K. UNIFIED MEMORY PROGRAMMING K.1. Unified Memory Introduction Unified Memory is a component of the CUDA programming model, first introduced in CUDA 6.0, that defines a managed memory space in which all processors see a single coherent memory image with a common address space. A processor refers to any independent execution unit with a dedicated MMU. This includes both CPUs and GPUs of any type and architecture. The underlying system manages data access and locality within a CUDA program without need for explicit memory copy calls. This benefits GPU programming in two primary ways: ‣ GPU programming is simplified by unifying memory spaces coherently across all GPUs and CPUs in the system and by providing tighter and more straightforward language integration for CUDA programmers. ‣ Data access speed is maximized by transparently migrating data towards the processor using it. In simple terms, Unified Memory eliminates the need for explicit data movement via the cudaMemcpy\*() routines without the performance penalty incurred by placing all data into zero-copy memory. Data movement, of course, still takes place, so a program’s run time typically does not decrease; Unified Memory instead enables the writing of simpler and more maintainable code. Unified Memory offers a “single-pointer-to-data” model that is conceptually similar to CUDA’s zero-copy memory. One key difference between the two is that with zero-copy allocations the physical location of memory is pinned in CPU system memory such that a program may have fast or slow access to it depending on where it is being accessed from. Unified Memory, on the other hand, decouples memory and execution spaces so that all data accesses are fast. The term Unified Memory describes a system that provides memory management services to a wide range of programs, from those targeting the Runtime API down to www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 308 ----------------------- Page 325----------------------- Unified Memory Programming those using the Virtual ISA (PTX). Part of this system defines the managed memory space that opts in to Unified Memory services. Managed memory is interoperable and interchangeable with device-specific allocations, such as those created using the cudaMalloc() routine. All CUDA operations that are valid on device memory are also valid on managed memory; the primary difference is that the host portion of a program is able to reference and access the memory as well. Unified memory is not supported on discrete GPUs attached to Tegra. K.1.1. System Requirements Unified Memory has two basic requirements: ‣ a GPU with SM architecture 3.0 or higher (Kepler class or newer) ‣ a 64-bit host application and non-embedded operating system (Linux, Windows, macOS) GPUs with SM architecture 6.x or higher (Pascal class or newer) provide additional Unified Memory features such as on-demand page migration and GPU memory oversubscription that are outlined throughout this document. Note that currently these features are only supported on Linux operating systems. Applications running on Windows (whether in TCC or WDDM mode) or macOS will use the basic Unified Memory model as on pre-6.x architectures even when they are running on hardware with compute capability 6.x or higher. See Data Migration and Coherency for details. K.1.2. Simplifying GPU Programming Unification of memory spaces means that there is no longer any need for explicit memory transfers between host and device. Any allocation created in the managed memory space is automatically migrated to where it is needed. A program allocates managed memory in one of two ways: via the cudaMallocManaged() routine, which is semantically similar to cudaMalloc(); or by defining a global \_\_managed\_\_ variable, which is semantically similar to a \_\_device\_\_ variable. Precise definitions of these are found later in this document. On supporting platforms with devices of compute capability 6.x and higher, Unified Memory will enable applications to allocate and share data using the default system allocator. This allows the GPU to access the entire system virtual memory without using a special allocator. See System Allocator for more detail. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 309 ----------------------- Page 326----------------------- Unified Memory Programming The following code examples illustrate how the use of managed memory can change the way in which host code is written. First, a simple program written without the benefit of Unified Memory: \_\_global\_\_ void AplusB(int \*ret, int a, int b) { ret[threadIdx.x] = a + b + threadIdx.x; } int main() { int \*ret; cudaMalloc(&ret, 1000 \* sizeof(int)); AplusB<<< 1, 1000 >>> (ret, 10, 100); int \*host\_ret = (int \*)malloc(1000 \* sizeof(int)); cudaMemcpy(host\_ret, ret, 1000 \* sizeof(int), cudaMemcpyDefault); for(int i = 0; i < 1000; i++) printf("%d: A+B = %d\n", i, host\_ret[i]); free(host\_ret); cudaFree(ret); return 0; } This first example combines two numbers together on the GPU with a per-thread ID and returns the values in an array. Without managed memory, both host- and device-side storage for the return values is required (host\_ret and ret in the example), as is an explicit copy between the two using cudaMemcpy(). Compare this with the Unified Memory version of the program, which allows direct access of GPU data from the host. Notice the cudaMallocManaged() routine, which returns a pointer valid from both host and device code. This allows ret to be used without a separate host\_ret copy, greatly simplifying and reducing the size of the program. \_\_global\_\_ void AplusB(int \*ret, int a, int b) { ret[threadIdx.x] = a + b + threadIdx.x; } int main() { int \*ret; cudaMallocManaged(&ret, 1000 \* sizeof(int)); AplusB<<< 1, 1000 >>> (ret, 10, 100); cudaDeviceSynchronize(); for(int i = 0; i < 1000; i++) printf("%d: A+B = %d\n", i, ret[i]); cudaFree(ret); return 0; } Finally, language integration allows direct reference of a GPU-declared \_\_managed\_\_ variable and simplifies a program further when global variables are used. \_\_device\_\_ \_\_managed\_\_ int ret[1000]; \_\_global\_\_ void AplusB(int a, int b) { ret[threadIdx.x] = a + b + threadIdx.x; } int main() { AplusB<<< 1, 1000 >>> (10, 100); cudaDeviceSynchronize(); for(int i = 0; i < 1000; i++) printf("%d: A+B = %d\n", i, ret[i]); return 0; } Note the absence of explicit cudaMemcpy() commands and the fact that the return array ret is visible on both CPU and GPU. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 310 ----------------------- Page 327----------------------- Unified Memory Programming It is worth a comment on the synchronization between host and device. Notice how in the non-managed example, the synchronous cudaMemcpy() routine is used both to synchronize the kernel (that is, to wait for it to finish running), and to transfer the data to the host. The Unified Memory examples do not call cudaMemcpy() and so require an explicit cudaDeviceSynchronize() before the host program can safely use the output from the GPU. K.1.3. Data Migration and Coherency Unified Memory attempts to optimize memory performance by migrating data towards the device where it is being accessed (that is, moving data to host memory if the CPU is accessing it and to device memory if the GPU will access it). Data migration is fundamental to Unified Memory, but is transparent to a program. The system will try to place data in the location where it can most efficiently be accessed without violating coherency. The physical location of data is invisible to a program and may be changed at any time, but accesses to the data’s virtual address will remain valid and coherent from any processor regardless of locality. Note that maintaining coherence is the primary requirement, ahead of performance; within the constraints of the host operating system, the system is permitted to either fail accesses or move data in order to maintain global coherence between processors. GPU architectures of compute capability lower than 6.x do not support fine-grained movement of the managed data to GPU on-demand. Whenever a GPU kernel is launched all managed memory generally has to be transfered to GPU memory to avoid faulting on memory access. With compute capability 6.x a new GPU page faulting mechanism is introduced that provides more seamless Unified Memory functionality. Combined with the system-wide virtual address space, page faulting provides several benefits. First, page faulting means that the CUDA system software doesn’t need to synchronize all managed memory allocations to the GPU before each kernel launch. If a kernel running on the GPU accesses a page that is not resident in its memory, it faults, allowing the page to be automatically migrated to the GPU memory on-demand. Alternatively, the page may be mapped into the GPU address space for access over the PCIe or NVLink interconnects (mapping on access can sometimes be faster than migration). Note that Unified Memory is system-wide: GPUs (and CPUs) can fault on and migrate memory pages either from CPU memory or from the memory of other GPUs in the system. K.1.4. GPU Memory Oversubscription Devices of compute capability lower than 6.x cannot allocate more managed memory than the physical size of GPU memory. Devices of compute capability 6.x extend addressing mode to support 49-bit virtual addressing. This is large enough to cover the 48-bit virtual address spaces of modern CPUs, as well as the GPU’s own memory. The large virtual address space and page faulting capability enable applications to access the entire system virtual memory, not limited by the physical memory size of any one processor. This means that applications can oversubscribe the memory system: in other words they can allocate, access, and www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 311 ----------------------- Page 328----------------------- Unified Memory Programming share arrays larger than the total physical capacity of the system, enabling out-of-core processing of very large datasets. cudaMallocManaged will not run out of memory as long as there is enough system memory available for the allocation. K.1.5. Multi-GPU For devices of compute capability lower than 6.x managed memory allocation behaves identically to unmanaged memory allocated using cudaMalloc(): the current active device is the home for the physical allocation, and all other GPUs receive peer mappings to the memory. This means that other GPUs in the system will access the memory at reduced bandwidth over the PCIe bus. Note that if peer mappings are not supported between the GPUs in the system, then the managed memory pages are placed in CPU system memory (“zero-copy” memory), and all GPUs will experience PCIe bandwidth restrictions. See Managed Memory with Multi-GPU Programs on pre-6.x Architectures for details. Managed allocations on systems with devices of compute capability 6.x are visible to all GPUs and can migrate to any processor on-demand. Unified Memory performance hints (see Performance Tuning) allow developers to explore custom usage patterns, such as read duplication of data across GPUs and direct access to peer GPU memory without migration. K.1.6. System Allocator Devices of compute capability 7.0 support Address Translation Services (ATS) over NVLink. If supported by the host CPU and operating system, ATS allows the GPU to directly access the CPU’s page tables. A miss in the GPU MMU will result in an Address Translation Request (ATR) to the CPU. The CPU looks in its page tables for the virtual- to-physical mapping for that address and supplies the translation back to the GPU. ATS provides the GPU full access to system memory, such as memory allocated with malloc, memory allocated on stack, global variables and file-backed memory. An application can query whether the device supports coherently accessing pageable memory via ATS by checking the new pageableMemoryAccessUsesHostPageTables property. Here is an example code that works on any system that satisfies the basic requirements for Unified Memory (see System Requirements): int \*data; cudaMallocManaged(&data, sizeof(int) \* n); kernel<<<grid, block>>> (data); These new access patterns are supported on systems with pageableMemoryAccess property: int \*data = (int\*)malloc(sizeof(int) \* n); kernel<<<grid, block>>> (data); int data[1024]; kernel<<<grid, block>>> (data); extern int \*data; kernel<<<grid, block>>> (data); In the example above, data could be initialized by a third party CPU library, and then directly accessed by the GPU kernel. On systems with pageableMemoryAccess, users www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 312 ----------------------- Page 329----------------------- Unified Memory Programming may also prefetch pageable memory to the GPU by using cudaMemPrefetchAsync. This could yield performance benefits through optimized data locality. ATS over NVLink is currently supported only on IBM Power9 systems. K.1.7. Hardware Coherency The second generation of NVLink allows direct load/store/atomic access from the CPU to each GPU’s memory. Coupled with a new CPU mastering capability, NVLink supports coherency operations allowing data reads from GPU memory to be stored in the CPU’s cache hierarchy. The lower latency of access from the CPU’s cache is key for CPU performance. Devices of compute capability 6.x support only peer GPU atomics. Devices of compute capability 7.x can send GPU atomics across NVLink and have them completed at the target CPU, thus the second generation of NVLink adds support for atomics initiated by either the GPU or the CPU. Note that cudaMalloc allocations are not accessible from the CPU. Therefore, to take advantage of hardware coherency users must use Unified Memory allocators such as cudaMallocManaged or system allocator with ATS support (see System Allocator). The new property directManagedMemAccessFromHost indicates if the host can directly access managed memory on the device without migration. By default, any CPU access of cudaMallocManaged allocations resident in GPU memory will trigger page faults and data migration. Applications can use cudaMemAdviseSetAccessedBy performance hint with cudaCpuDeviceId to enable direct access of GPU memory on supported systems. Consider an example code below: \_\_global\_\_ void write(int \*ret, int a, int b) { ret[threadIdx.x] = a + b + threadIdx.x; } \_\_global\_\_ void append(int \*ret, int a, int b) { ret[threadIdx.x] += a + b + threadIdx.x; } int main() { int \*ret; cudaMallocManaged(&ret, 1000 \* sizeof(int)); cudaMemAdvise(ret, 1000 \* sizeof(int), cudaMemAdviseSetAccessedBy, cudaCpuDeviceId); // set direct access hint write<<< 1, 1000 >>> (ret, 10, 100); // pages populated in GPU memory cudaDeviceSynchronize(); for(int i = 0; i < 1000; i++) printf("%d: A+B = %d\n", i, ret[i]); // directManagedMemAccessFromHost=1: CPU accesses GPU memory directly without migrations // directManagedMemAccessFromHost=0: CPU faults and triggers device-to-host migrations append<<< 1, 1000 >>> (ret, 10, 100); // directManagedMemAccessFromHost=1: GPU accesses GPU memory without migrations cudaDeviceSynchronize(); // directManagedMemAccessFromHost=0: GPU faults and triggers host-to-device migrations cudaFree(ret); return 0; } www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 313 ----------------------- Page 330----------------------- Unified Memory Programming After write kernel is completed, ret will be created and initialized in GPU memory. Next, the CPU will access ret followed by append kernel using the same ret memory again. This code will show different behavior depending on the system architecture and support of hardware coherency: ‣ On systems with directManagedMemAccessFromHost=1: CPU accesses to the managed buffer will not trigger any migrations; the data will remain resident in GPU memory and any subsequent GPU kernels can continue to access it directly without inflicting faults or migrations. ‣ On systems with directManagedMemAccessFromHost=0: CPU accesses to the managed buffer will page fault and initiate data migration; any GPU kernel trying to access the same data first time will page fault and migrate pages back to GPU memory. K.1.8. Access Counters Devices of compute capability 7.0 introduce a new Access Counter feature that keeps track of the frequency of access that a GPU makes to memory located on other processors. Access Counters help ensure memory pages are moved to the physical memory of the processor that is accessing the pages most frequently. The Access Counters feature can guide migrations between CPU and GPU, and between peer GPUs. For cudaMallocManaged, Access Counters migration can be opt-in by using cudaMemAdviseSetAccessedBy hint with the corresponding device id. The driver may also use Access Counters for more efficient thrashing mitigation or memory oversubscription scenarios. Access Counters are currently enabled only on IBM Power9 systems and only for the cudaMallocManaged allocator. K.2. Programming Model K.2.1. Managed Memory Opt In Most platforms require a program to opt in to automatic data management by either annotating a \_\_device\_\_ variable with the \_\_managed\_\_ keyword (see the Language Integration section) or by using a new cudaMallocManaged() call to allocate data. Devices of compute capability lower than 6.x must always allocate managed memory on the heap, either with an allocator or by declaring global storage. It is not possible either to associate previously allocated memory with Unified Memory, or to have the Unified Memory system manage a CPU or a GPU stack pointer. Starting with CUDA 8.0 and on supporting systems with devices of compute capability 6.x, memory allocated with the default OS allocator (e.g. malloc or new) can be accessed from both GPU code and CPU code using the same pointer. On these systems, Unified Memory is the default: there is no need to use a special allocator or the creation of a specially managed memory pool. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 314 ----------------------- Page 331----------------------- Unified Memory Programming K.2.1.1. Explicit Allocation Using cudaMallocManaged() Unified memory is most commonly created using an allocation function that is semantically and syntactically similar to the standard CUDA allocator, cudaMalloc(). The function description is as follows: cudaError\_t cudaMallocManaged(void \*\*devPtr, size\_t size, unsigned int flags=0); The cudaMallocManaged() function reserves size bytes of managed memory and returns a pointer in devPtr. Note the difference in cudaMallocManaged() behavior between various GPU architectures. By default, the devices of compute capability lower than 6.x allocate managed memory directly on the GPU. However, the devices of compute capability 6.x and greater do not allocate physical memory when calling cudaMallocManaged(): in this case physical memory is populated on first touch and may be resident on the CPU or the GPU. The managed pointer is valid on all GPUs and the CPU in the system, although program accesses to this pointer must obey the concurrency rules of the Unified Memory programming model (see Coherency and Concurrency). Below is a simple example, showing the use of cudaMallocManaged(): \_\_global\_\_ void printme(char \*str) { printf(str); } int main() { // Allocate 100 bytes of memory, accessible to both Host and Device code char \*s; cudaMallocManaged(&s, 100); // Note direct Host-code use of "s" strncpy(s, "Hello Unified Memory\n", 99); // Here we pass "s" to a kernel without explicitly copying printme<<< 1, 1 >>> (s); cudaDeviceSynchronize(); // Free as for normal CUDA allocations cudaFree(s); return 0; } A program’s behavior is functionally unchanged when cudaMalloc() is replaced with cudaMallocManaged(); however, the program should go on to eliminate explicit memory copies and take advantage of automatic migration. Additionally, dual pointers (one to host and one to device memory) can be eliminated. Device code is not able to call cudaMallocManaged(). All managed memory must be allocated from the host or at global scope (see the next section). Allocations on the device heap using malloc() in a kernel will not be created in the managed memory space, and so will not be accessible to CPU code. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 315 ----------------------- Page 332----------------------- Unified Memory Programming K.2.1.2. Global-Scope Managed Variables Using \_\_managed\_\_ File-scope and global-scope CUDA \_\_device\_\_ variables may also opt-in to Unified Memory management by adding a new \_\_managed\_\_ annotation to the declaration. These may then be referenced directly from either host or device code, as follows: \_\_device\_\_ \_\_managed\_\_ int x[2]; \_\_device\_\_ \_\_managed\_\_ int y; \_\_global\_\_ void kernel() { x[1] = x[0] + y; } int main() { x[0] = 3; y = 5; kernel<<< 1, 1 >>> (); cudaDeviceSynchronize(); printf("result = %d\n", x[1]); return 0; } All semantics of the original \_\_device\_\_ memory space, along with some additional unified-memory-specific constraints, are inherited by the managed variable (see Compilation with NVCC). Note that variables marked \_\_constant\_\_ may not also be marked as \_\_managed\_\_; this annotation is reserved for \_\_device\_\_ variables only. Constant memory must be set either statically at compile time or by using cudaMemcpyToSymbol() as usual in CUDA. K.2.2. Coherency and Concurrency Simultaneous access to managed memory on devices of compute capability lower than 6.x is not possible, because coherence could not be guaranteed if the CPU accessed a Unified Memory allocation while a GPU kernel was active. However, devices of compute capability 6.x on supporting operating systems allow the CPUs and GPUs to access Unified Memory allocations simultaneously via the new page faulting mechanism. A program can query whether a device supports concurrent access to managed memory by checking a new concurrentManagedAccess property. Note, as with any parallel application, developers need to ensure correct synchronization to avoid data hazards between processors. K.2.2.1. GPU Exclusive Access To Managed Memory To ensure coherency on pre-6.x GPU architectures, the Unified Memory programming model puts constraints on data accesses while both the CPU and GPU are executing concurrently. In effect, the GPU has exclusive access to all managed data while any kernel operation is executing, regardless of whether the specific kernel is actively using the data. When managed data is used with cudaMemcpy\*() or cudaMemset\*(), the system may choose to access the source or destination from the host or the device, which will put constraints on concurrent CPU access to that data while the cudaMemcpy\*() or cudaMemset\*() is executing. See Memcpy()/Memset() Behavior With Managed Memory for further details. It is not permitted for the CPU to access any managed allocations or variables while the GPU is active for devices with concurrentManagedAccess property set to 0. On these www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 316 ----------------------- Page 333----------------------- Unified Memory Programming systems concurrent CPU/GPU accesses, even to different managed memory allocations, will cause a segmentation fault because the page is considered inaccessible to the CPU. \_\_device\_\_ \_\_managed\_\_ int x, y=2; \_\_global\_\_ void kernel() { x = 10; } int main() { kernel<<< 1, 1 >>> (); y = 20; // Error on GPUs not supporting concurrent access cudaDeviceSynchronize(); return 0; } In example above, the GPU program kernel is still active when the CPU touches y. (Note how it occurs before cudaDeviceSynchronize().) The code runs successfully on devices of compute capability 6.x due to the GPU page faulting capability which lifts all restrictions on simultaneous access. However, such memory access is invalid on pre-6.x architectures even though the CPU is accessing different data than the GPU. The program must explicitly synchronize with the GPU before accessing y: \_\_device\_\_ \_\_managed\_\_ int x, y=2; \_\_global\_\_ void kernel() { x = 10; } int main() { kernel<<< 1, 1 >>> (); cudaDeviceSynchronize(); y = 20; // Success on GPUs not supporing concurrent access return 0; } As this example shows, on systems with pre-6.x GPU architectures, a CPU thread may not access any managed data in between performing a kernel launch and a subsequent synchronization call, regardless of whether the GPU kernel actually touches that same data (or any managed data at all). The mere potential for concurrent CPU and GPU access is sufficient for a process-level exception to be raised. Note that if memory is dynamically allocated with cudaMallocManaged() or cuMemAllocManaged() while the GPU is active, the behavior of the memory is unspecified until additional work is launched or the GPU is synchronized. Attempting to access the memory on the CPU during this time may or may not cause a segmentation fault. This does not apply to memory allocated using the flag cudaMemAttachHost or CU\_MEM\_ATTACH\_HOST. K.2.2.2. Explicit Synchronization and Logical GPU Activity Note that explicit synchronization is required even if kernel runs quickly and finishes before the CPU touches y in the above example. Unified Memory uses logical activity to determine whether the GPU is idle. This aligns with the CUDA programming model, which specifies that a kernel can run at any time following a launch and is not guaranteed to have finished until the host issues a synchronization call. Any function call that logically guarantees the GPU completes its work is valid. This includes cudaDeviceSynchronize(); cudaStreamSynchronize() and cudaStreamQuery() (provided it returns cudaSuccess and not cudaErrorNotReady) where the specified stream is the only stream still executing on the GPU; www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 317 ----------------------- Page 334----------------------- Unified Memory Programming cudaEventSynchronize() and cudaEventQuery() in cases where the specified event is not followed by any device work; as well as uses of cudaMemcpy() and cudaMemset() that are documented as being fully synchronous with respect to the host. Dependencies created between streams will be followed to infer completion of other streams by synchronizing on a stream or event. Dependencies can be created via cudaStreamWaitEvent() or implicitly when using the default (NULL) stream. It is legal for the CPU to access managed data from within a stream callback, provided no other stream that could potentially be accessing managed data is active on the GPU. In addition, a callback that is not followed by any device work can be used for synchronization: for example, by signaling a condition variable from inside the callback; otherwise, CPU access is valid only for the duration of the callback(s). There are several important points of note: ‣ It is always permitted for the CPU to access non-managed zero-copy data while the GPU is active. ‣ The GPU is considered active when it is running any kernel, even if that kernel does not make use of managed data. If a kernel might use data, then access is forbidden, unless device property concurrentManagedAccess is 1. ‣ There are no constraints on concurrent inter-GPU access of managed memory, other than those that apply to multi-GPU access of non-managed memory. ‣ There are no constraints on concurrent GPU kernels accessing managed data. Note how the last point allows for races between GPU kernels, as is currently the case for non-managed GPU memory. As mentioned previously, managed memory functions identically to non-managed memory from the perspective of the GPU. The following code example illustrates these points: int main() { cudaStream\_t stream1, stream2; cudaStreamCreate(&stream1); cudaStreamCreate(&stream2); int \*non\_managed, \*managed, \*also\_managed; cudaMallocHost(&non\_managed, 4); // Non-managed, CPU-accessible memory cudaMallocManaged(&managed, 4); cudaMallocManaged(&also\_managed, 4); // Point 1: CPU can access non-managed data. kernel<<< 1, 1, 0, stream1 >>> (managed); \*non\_managed = 1; // Point 2: CPU cannot access any managed data while GPU is busy, // unless concurrentManagedAccess = 1 // Note we have not yet synchronized, so "kernel" is still active. \*also\_managed = 2; // Will issue segmentation fault // Point 3: Concurrent GPU kernels can access the same data. kernel<<< 1, 1, 0, stream2 >>> (managed); // Point 4: Multi-GPU concurrent access is also permitted. cudaSetDevice(1); kernel<<< 1, 1 >>> (managed); return 0; } K.2.2.3. Managing Data Visibility and Concurrent CPU + GPU Access with Streams Until now it was assumed that for SM architectures before 6.x: 1) any active kernel may use any managed memory, and 2) it was invalid to use managed memory from the CPU www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 318 ----------------------- Page 335----------------------- Unified Memory Programming while a kernel is active. Here we present a system for finer-grained control of managed memory designed to work on all devices supporting managed memory, including older architectures with concurrentManagedAccess equal to 0. The CUDA programming model provides streams as a mechanism for programs to indicate dependence and independence among kernel launches. Kernels launched into the same stream are guaranteed to execute consecutively, while kernels launched into different streams are permitted to execute concurrently. Streams describe independence between work items and hence allow potentially greater efficiency through concurrency. Unified Memory builds upon the stream-independence model by allowing a CUDA program to explicitly associate managed allocations with a CUDA stream. In this way, the programmer indicates the use of data by kernels based on whether they are launched into a specified stream or not. This enables opportunities for concurrency based on program-specific data access patterns. The function to control this behaviour is: cudaError\_t cudaStreamAttachMemAsync(cudaStream\_t stream, void \*ptr, size\_t length=0, unsigned int flags=0); The cudaStreamAttachMemAsync() function associates length bytes of memory starting from ptr with the specified stream. (Currently, length must always be 0 to indicate that the entire region should be attached.) Because of this association, the Unified Memory system allows CPU access to this memory region so long as all operations in stream have completed, regardless of whether other streams are active. In effect, this constrains exclusive ownership of the managed memory region by an active GPU to per-stream activity instead of whole-GPU activity. Most importantly, if an allocation is not associated with a specific stream, it is visible to all running kernels regardless of their stream. This is the default visibility for a cudaMallocManaged() allocation or a \_\_managed\_\_ variable; hence, the simple-case rule that the CPU may not touch the data while any kernel is running. By associating an allocation with a specific stream, the program makes a guarantee that only kernels launched into that stream will touch that data. No error checking is performed by the Unified Memory system: it is the programmer’s responsibility to ensure that guarantee is honored. In addition to allowing greater concurrency, the use of cudaStreamAttachMemAsync() can (and typically does) enable data transfer optimizations within the Unified Memory system that may affect latencies and other overhead. K.2.2.4. Stream Association Examples Associating data with a stream allows fine-grained control over CPU + GPU concurrency, but what data is visible to which streams must be kept in mind when using www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 319 ----------------------- Page 336----------------------- Unified Memory Programming devices of compute capability lower than 6.x. Looking at the earlier synchronization example: \_\_device\_\_ \_\_managed\_\_ int x, y=2; \_\_global\_\_ void kernel() { x = 10; } int main() { cudaStream\_t stream1; cudaStreamCreate(&stream1); cudaStreamAttachMemAsync(stream1, &y, 0, cudaMemAttachHost); cudaDeviceSynchronize(); // Wait for Host attachment to occur. kernel<<< 1, 1, 0, stream1 >>> (); // Note: Launches into stream1. y = 20; // Success – a kernel is running but “y” // has been associated with no stream. return 0; } Here we explicitly associate y with host accessibility, thus enabling access at all times from the CPU. (As before, note the absence of cudaDeviceSynchronize() before the access.) Accesses to y by the GPU running kernel will now produce undefined results. Note that associating a variable with a stream does not change the associating of any other variable. E.g. associating x with stream1 does not ensure that only x is accessed by kernels launched in stream1, thus an error is caused by this code: \_\_device\_\_ \_\_managed\_\_ int x, y=2; \_\_global\_\_ void kernel() { x = 10; } int main() { cudaStream\_t stream1; cudaStreamCreate(&stream1); cudaStreamAttachMemAsync(stream1, &x);// Associate “x” with stream1. cudaDeviceSynchronize(); // Wait for “x” attachment to occur. kernel<<< 1, 1, 0, stream1 >>> (); // Note: Launches into stream1. y = 20; // ERROR: “y” is still associated globally // with all streams by default return 0; } Note how the access to y will cause an error because, even though x has been associated with a stream, we have told the system nothing about who can see y. The system therefore conservatively assumes that kernel might access it and prevents the CPU from doing so. K.2.2.5. Stream Attach With Multithreaded Host Programs The primary use for cudaStreamAttachMemAsync() is to enable independent task parallelism using CPU threads. Typically in such a program, a CPU thread creates its own stream for all work that it generates because using CUDA’s NULL stream would cause dependencies between threads. The default global visibility of managed data to any GPU stream can make it difficult to avoid interactions between CPU threads in a multi-threaded program. Function cudaStreamAttachMemAsync() is therefore used to associate a thread’s managed allocations with that thread’s own stream, and the association is typically not changed for the life of the thread. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 320 ----------------------- Page 337----------------------- Unified Memory Programming Such a program would simply add a single call to cudaStreamAttachMemAsync() to use unified memory for its data accesses: // This function performs some task, in its own private stream. void run\_task(int \*in, int \*out, int length) { // Create a stream for us to use. cudaStream\_t stream; cudaStreamCreate(&stream); // Allocate some managed data and associate with our stream. // Note the use of the host-attach flag to cudaMallocManaged(); // we then associate the allocation with our stream so that // our GPU kernel launches can access it. int \*data; cudaMallocManaged((void \*\*)&data, length, cudaMemAttachHost); cudaStreamAttachMemAsync(stream, data); cudaStreamSynchronize(stream); // Iterate on the data in some way, using both Host & Device. for(int i=0; i<N; i++) { transform<<< 100, 256, 0, stream >>> (in, data, length); cudaStreamSynchronize(stream); host\_process(data, length); // CPU uses managed data. convert<<< 100, 256, 0, stream >>> (out, data, length); } cudaStreamSynchronize(stream); cudaStreamDestroy(stream); cudaFree(data); } In this example, the allocation-stream association is established just once, and then data is used repeatedly by both the host and device. The result is much simpler code than occurs with explicitly copying data between host and device, although the result is the same. K.2.2.6. Advanced Topic: Modular Programs and Data Access Constraints In the previous example cudaMallocManaged() specifies the cudaMemAttachHost flag, which creates an allocation that is initially invisible to device-side execution. (The default allocation would be visible to all GPU kernels on all streams.) This ensures that there is no accidental interaction with another thread’s execution in the interval between the data allocation and when the data is acquired for a specific stream. Without this flag, a new allocation would be considered in-use on the GPU if a kernel launched by another thread happens to be running. This might impact the thread’s ability to access the newly allocated data from the CPU (for example, within a base- class constructor) before it is able to explicitly attach it to a private stream. To enable safe independence between threads, therefore, allocations should be made specifying this flag. An alternative would be to place a process-wide barrier across all threads after the allocation has been attached to the stream. This would ensure that all threads complete their data/stream associations before any kernels are launched, avoiding the hazard. A second barrier would be needed before the stream is destroyed because stream destruction causes allocations to revert to their default visibility. The cudaMemAttachHost flag exists both to simplify this process, and because it is not always possible to insert global barriers where required. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 321 ----------------------- Page 338----------------------- Unified Memory Programming K.2.2.7. Memcpy()/Memset() Behavior With Managed Memory Since managed memory can be accessed from either the host or the device, cudaMemcpy\*() relies on the type of transfer, specified using cudaMemcpyKind, to determine whether the data should be accessed as a host pointer or a device pointer. If cudaMemcpyHostTo\* is specified and the source data is managed, then it will accessed from the host if it is coherently accessible from the host in the copy stream (1); otherwise it will be accessed from the device. Similar rules apply to the destination when cudaMemcpy\*ToHost is specified and the destination is managed memory. If cudaMemcpyDeviceTo\* is specified and the source data is managed, then it will be accessed from the device. The source must be coherently accessible from the device in the copy stream (2); otherwise, an error is returned. Similar rules apply to the destination when cudaMemcpy\*ToDevice is specified and the destination is managed memory. If cudaMemcpyDefault is specified, then managed data will be accessed from the host either if it cannot be coherently accessed from the device in the copy stream (2) or if the preferred location for the data is cudaCpuDeviceId and it can be coherently accessed from the host in the copy stream (1); otherwise, it will be accessed from the device. When using cudaMemset\*() with managed memory, the data is always accessed from the device. The data must be coherently accessible from the device in the stream being used for the cudaMemset() operation (2); otherwise, an error is returned. When data is accessed from the device either by cudaMemcpy\* or cudaMemset\*, the stream of operation is considered to be active on the GPU. During this time, any CPU access of data that is associated with that stream or data that has global visibility, will result in a segmentation fault if the GPU has a zero value for the device attribute concurrentManagedAccess. The program must synchronize appropriately to ensure the operation has completed before accessing any associated data from the CPU. (1) For managed memory to be coherently accessible from the host in a given stream, at least one of the following conditions must be satisfied: ‣ The given stream is associated with a device that has a non-zero value for the device attribute concurrentManagedAccess. ‣ The memory neither has global visibility nor is it associated with the given stream. (2) For managed memory to be coherently accessible from the device in a given stream, at least one of the following conditions must be satisfied: ‣ The device has a non-zero value for the device attribute concurrentManagedAccess. ‣ The memory either has global visibility or is associated with the given stream. K.2.3. Language Integration Users of the CUDA Runtime API who compile their host code using nvcc have access to additional language integration features, such as shared symbol names and inline kernel launch via the <<<...>>> operator. Unified Memory adds one additional element to www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 322 ----------------------- Page 339----------------------- Unified Memory Programming CUDA’s language integration: variables annotated with the \_\_managed\_\_ keyword can be referenced directly from both host and device code. The following example, seen earlier in Simplifying GPU Programming, illustrates a simple use of \_\_managed\_\_ global declarations: // Managed variable declaration is an extra annotation with \_\_device\_\_ \_\_device\_\_ \_\_managed\_\_ int x; \_\_global\_\_ void kernel() { // Reference "x" directly - it's a normal variable on the GPU. printf( "GPU sees: x = %d\n" , x); } int main() { // Set "x" from Host code. Note it's just a normal variable on the CPU. x = 1234; // Launch a kernel which uses "x" from the GPU. kernel<<< 1, 1 >>> (); cudaDeviceSynchronize(); return 0; } The capability available with \_\_managed\_\_ variables is that the symbol is available in both device code and in host code without the need to dereference a pointer, and the data is shared by all. This makes it particularly easy to exchange data between host and device programs without the need for explicit allocations or copying. Semantically, the behavior of \_\_managed\_\_ variables is identical to that of storage allocated via cudaMallocManaged(). See Explicit Allocation Using cudaMallocManaged() for detailed explanation. Stream visibility defaults to cudaMemAttachGlobal, but may be constrained using cudaStreamAttachMemAsync(). A valid CUDA context is necessary for the correct operation of \_\_managed\_\_ variables. Accessing \_\_managed\_\_ variables can trigger CUDA context creation if a context for the current device hasn’t already been created. In the example above, accessing x before the kernel launch triggers context creation on device 0. In the absence of that access, the kernel launch would have triggered context creation. C++ objects declared as \_\_managed\_\_ are subject to certain specific constraints, particularly where static initializers are concerned. Please refer to C++ Language Support in the CUDA C++ Programming Guide for a list of these constraints. K.2.3.1. Host Program Errors with \_\_managed\_\_ Variables The use of \_\_managed\_\_ variables depends upon the underlying Unified Memory system functioning correctly. Incorrect functioning can occur if, for example, the CUDA installation failed or if the CUDA context creation was unsuccessful. When CUDA-specific operations fail, typically an error is returned that indicates the source of the failure. Using \_\_managed\_\_ variables introduces a new failure mode whereby a non-CUDA operation (for example, CPU access to what should be a valid host memory address) can fail if the Unified Memory system is not operating correctly. Such invalid memory accesses cannot easily be attributed to the underlying CUDA subsystem, although a debugger such as cuda-gdb will indicate that a managed memory address is the source of the failure. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 323 ----------------------- Page 340----------------------- Unified Memory Programming K.2.4. Querying Unified Memory Support K.2.4.1. Device Properties Unified Memory is supported only on devices with compute capability 3.0 or higher. A program may query whether a GPU device supports managed memory by using cudaGetDeviceProperties() and checking the new managedMemory property. The capability can also be determined using the individual attribute query function cudaDeviceGetAttribute() with the attribute cudaDevAttrManagedMemory. Either property will be set to 1 if managed memory allocations are permitted on the GPU and under the current operating system. Note that Unified Memory is not supported for 32-bit applications (unless on Android), even if a GPU is of sufficient capability. Devices of compute capability 6.x on supporting platforms can access pageable memory without calling cudaHostRegister on it. An application can query whether the device supports coherently accessing pageable memory by checking the new pageableMemoryAccess property. With the new page fault mechanism, global data coherency is guaranteed with Unified Memory. This means that the CPUs and GPUs can access Unified Memory allocations simultaneously. This was illegal on devices of compute capability lower than 6.x, because coherence could not be guaranteed if the CPU accessed a Unified Memory allocation while a GPU kernel was active. A program can query concurrent access support by checking concurrentManagedAccess property. See Coherency and Concurrency for details. K.2.4.2. Pointer Attributes To determine if a given pointer refers to managed memory, a program can call cudaPointerGetAttributes() and check the value of the isManaged attribute. This attribute is set to 1 if the pointer refers to managed memory and to 0 if not. K.2.5. Advanced Topics K.2.5.1. Managed Memory with Multi-GPU Programs on pre-6.x Architectures On systems with devices of compute capabilities lower than 6.x managed allocations are automatically visible to all GPUs in a system via the peer-to-peer capabilities of the GPUs. On Linux the managed memory is allocated in GPU memory as long as all GPUs that are actively being used by a program have the peer-to-peer support. If at any time the application starts using a GPU that doesn’t have peer-to-peer support with any of the other GPUs that have managed allocations on them, then the driver will migrate all managed allocations to system memory. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 324 ----------------------- Page 341----------------------- Unified Memory Programming On Windows if peer mappings are not available (for example, between GPUs of different architectures), then the system will automatically fall back to using zero-copy memory, regardless of whether both GPUs are actually used by a program. If only one GPU is actually going to be used, it is necessary to set the CUDA\_VISIBLE\_DEVICES environment variable before launching the program. This constrains which GPUs are visible and allows managed memory to be allocated in GPU memory. Alternatively, on Windows users can also set CUDA\_MANAGED\_FORCE\_DEVICE\_ALLOC to a non-zero value to force the driver to always use device memory for physical storage. When this environment variable is set to a non-zero value, all devices used in that process that support managed memory have to be peer-to-peer compatible with each other. The error ::cudaErrorInvalidDevice will be returned if a device that supports managed memory is used and it is not peer-to-peer compatible with any of the other managed memory supporting devices that were previously used in that process, even if ::cudaDeviceReset has been called on those devices. These environment variables are described in Appendix CUDA Environment Variables. Note that starting from CUDA 8.0 CUDA\_MANAGED\_FORCE\_DEVICE\_ALLOC has no effect on Linux operating systems. K.2.5.2. Using fork() with Managed Memory The Unified Memory system does not allow sharing of managed memory pointers between processes. It will not correctly manage memory handles that have been duplicated via a fork() operation. Results will be undefined if either the child or parent accesses managed data following a fork(). It is safe, however, to fork() a child process that then immediately exits via an exec() call, because the child drops the memory handles and the parent becomes the sole owner once again. It is not safe for the parent to exit and leave the child to access the handles. K.3. Performance Tuning In order to achieve good performance with Unified Memory, the following objectives must be met: ‣ Faults should be avoided: While replayable faults are fundamental to enabling a simpler programming model, they can be severely detrimental to application performance. Fault handling can take tens of microseconds because it may involve TLB invalidates, data migrations and page table updates. All the while, execution in certain portions of the application will be halted, thereby potentially impacting overall performance. ‣ Data should be local to the accessing processor: As mentioned before, memory access latencies and bandwidth are significantly better when the data is placed local to the processor accessing it. Therefore, data should be suitably migrated to take advantage of lower latencies and higher bandwidth. ‣ Memory thrashing should be prevented: If data is frequently accessed by multiple processors and has to be constantly migrated around to achieve data locality, then the overhead of migration may exceed the benefits of locality. Memory thrashing should be prevented to the extent possible. If it cannot be prevented, it must be detected and resolved appropriately. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 325 ----------------------- Page 342----------------------- Unified Memory Programming To achieve the same level of performance as what's possible without using Unified Memory, the application has to guide the Unified Memory driver subsystem into avoiding the aforementioned pitfalls. It is worthy to note that the Unified Memory driver subsystem can detect common data access patterns and achieve some of these objectives automatically without application participation. But when the data access patterns are non-obvious, explicit guidance from the application is crucial. CUDA 8.0 introduces useful APIs for providing the runtime with memory usage hints (cudaMemAdvise()) and for explicit prefetching (cudaMemPrefetchAsync()). These tools allow the same capabilities as explicit memory copy and pinning APIs without reverting to the limitations of explicit GPU memory allocation. cudaMemPrefetchAsync() is not supported on Tegra devices. K.3.1. Data Prefetching Data prefetching means migrating data to a processor’s memory and mapping it in that processor’s page tables before the processor begins accessing that data. The intent of data prefetching is to avoid faults while also establishing data locality. This is most valuable for applications that access data primarily from a single processor at any given time. As the accessing processor changes during the lifetime of the application, the data can be prefetched accordingly to follow the execution flow of the application. Since work is launched in streams in CUDA, it is expected of data prefetching to also be a streamed operation as shown in the following API: cudaError\_t cudaMemPrefetchAsync(const void \*devPtr, size\_t count, int dstDevice, cudaStream\_t stream); where the memory region specified by devPtr pointer and count number of bytes, with ptr rounded down to the nearest page boundary and count rounded up to the nearest page boundary, is migrated to the dstDevice by enqueueing a migration operation in stream. Passing in cudaCpuDeviceId for dstDevice will cause data to be migrated to CPU memory. Consider a simple code example below: void foo(cudaStream\_t s) { char \*data; cudaMallocManaged(&data, N); init\_data(data, N); // execute on CPU cudaMemPrefetchAsync(data, N, myGpuId, s); // prefetch to GPU mykernel<<<..., s>>> (data, N, 1, compare); // execute on GPU cudaMemPrefetchAsync(data, N, cudaCpuDeviceId, s); // prefetch to CPU cudaStreamSynchronize(s); use\_data(data, N); cudaFree(data); } Without performance hints the kernel mykernel will fault on first access to data which creates additional overhead of the fault processing and generally slows down the application. By prefetching data in advance it is possible to avoid page faults and achieve better performance. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 326 ----------------------- Page 343----------------------- Unified Memory Programming This API follows stream ordering semantics, i.e. the migration does not begin until all prior operations in the stream have completed, and any subsequent operation in the stream does not begin until the migration has completed. K.3.2. Data Usage Hints Data prefetching alone is insufficient when multiple processors need to simultaneously access the same data. In such scenarios, it's useful for the application to provide hints on how the data will actually be used. The following advisory API can be used to specify data usage: cudaError\_t cudaMemAdvise(const void \*devPtr, size\_t count, enum cudaMemoryAdvise advice, int device); where advice, specified for data contained in region starting from devPtr address and with the length of count bytes, rounded to the nearest page boundary, can take the following values: ‣ cudaMemAdviseSetReadMostly: This implies that the data is mostly going to be read from and only occasionally written to. This allows the driver to create read- only copies of the data in a processor's memory when that processor accesses it. Similarly, if cudaMemPrefetchAsync is called on this region, it will create a read- only copy of the data on the destination processor. When a processor writes to this data, all copies of the corresponding page are invalidated except for the one where the write occurred. The device argument is ignored for this advice. This advice allows multiple processors to simultaneously access the same data at maximal bandwidth as illustrated in the following code snippet: char \*dataPtr; size\_t dataSize = 4096; // Allocate memory using malloc or cudaMallocManaged dataPtr = (char \*)malloc(dataSize); // Set the advice on the memory region cudaMemAdvise(dataPtr, dataSize, cudaMemAdviseSetReadMostly, 0); int outerLoopIter = 0; while (outerLoopIter < maxOuterLoopIter) { // The data is written to in the outer loop on the CPU initializeData(dataPtr, dataSize); // The data is made available to all GPUs by prefetching. // Prefetching here causes read duplication of data instead // of data migration for (int device = 0; device < maxDevices; device++) { cudaMemPrefetchAsync(dataPtr, dataSize, device, stream); } // The kernel only reads this data in the inner loop int innerLoopIter = 0; while (innerLoopIter < maxInnerLoopIter) { kernel<<<32,32>>> ((const char \*)dataPtr); innerLoopIter++; } outerLoopIter++; } ‣ cudaMemAdviseSetPreferredLocation: This advice sets the preferred location for the data to be the memory belonging to device. Passing in a value of cudaCpuDeviceId for device sets the preferred location as CPU memory. Setting the preferred location does not cause data to migrate to that location immediately. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 327 ----------------------- Page 344----------------------- Unified Memory Programming Instead, it guides the migration policy when a fault occurs on that memory region. If the data is already in its preferred location and the faulting processor can establish a mapping without requiring the data to be migrated, then the migration will be avoided. On the other hand, if the data is not in its preferred location or if a direct mapping cannot be established, then it will be migrated to the processor accessing it. It is important to note that setting the preferred location does not prevent data prefetching done using cudaMemPrefetchAsync. ‣ cudaMemAdviseSetAccessedBy: This advice implies that the data will be accessed by device. This does not cause data migration and has no impact on the location of the data per se. Instead, it causes the data to always be mapped in the specified processor’s page tables, as long as the location of the data permits a mapping to be established. If the data gets migrated for any reason, the mappings are updated accordingly. This advice is useful in scenarios where data locality is not important, but avoiding faults is. Consider for example a system containing multiple GPUs with peer-to-peer access enabled, where the data located on one GPU is occasionally accessed by other GPUs. In such scenarios, migrating data over to the other GPUs is not as important because the accesses are infrequent and the overhead of migration may be too high. But preventing faults can still help improve performance, and so having a mapping set up in advance is useful. Note that on CPU access of this data, the data may be migrated to CPU memory because the CPU cannot access GPU memory directly. Any GPU that had the cudaMemAdviceSetAccessedBy flag set for this data will now have its mapping updated to point to the page in CPU memory. Each advice can be also unset by using one of the following values: cudaMemAdviseUnsetReadMostly, cudaMemAdviseUnsetPreferredLocation and cudaMemAdviseUnsetAccessedBy. K.3.3. Querying Usage Attributes A program can query memory range attributes assigned through cudaMemAdvise or cudaMemPrefetchAsync by using the following API: cudaMemRangeGetAttribute(void \*data, size\_t dataSize, enum cudaMemRangeAttribute attribute, const void \*devPtr, size\_t count); This function queries an attribute of the memory range starting at devPtr with a size of count bytes. The memory range must refer to managed memory allocated via cudaMallocManaged or declared via \_\_managed\_\_ variables. It is possible to query the following attributes: ‣ cudaMemRangeAttributeReadMostly: the result returned will be 1 if all pages in the given memory range have read-duplication enabled, or 0 otherwise. ‣ cudaMemRangeAttributePreferredLocation: the result returned will be a GPU device id or cudaCpuDeviceId if all pages in the memory range have the corresponding processor as their preferred location, otherwise cudaInvalidDeviceId will be returned. An application can use this query API to make decision about staging data through CPU or GPU depending on the preferred www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 328 ----------------------- Page 345----------------------- Unified Memory Programming location attribute of the managed pointer. Note that the actual location of the pages in the memory range at the time of the query may be different from the preferred location. ‣ cudaMemRangeAttributeAccessedBy: will return the list of devices that have that advise set for that memory range. ‣ cudaMemRangeAttributeLastPrefetchLocation: will return the last location to which all pages in the memory range were prefetched explicitly using cudaMemPrefetchAsync. Note that this simply returns the last location that the application requested to prefetch the memory range to. It gives no indication as to whether the prefetch operation to that location has completed or even begun. Additionally, multiple attributes can be queried by using corresponding cudaMemRangeGetAttributes function. www.nvidia.com CUDA C++ Programming Guide PG-02829-001\_v10.2 | 329 ----------------------- Page 346----------------------- 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 of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation. Trademarks NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated. Copyright © 2007-2019 NVIDIA Corporation. All rights reserved. www.nvidia.com