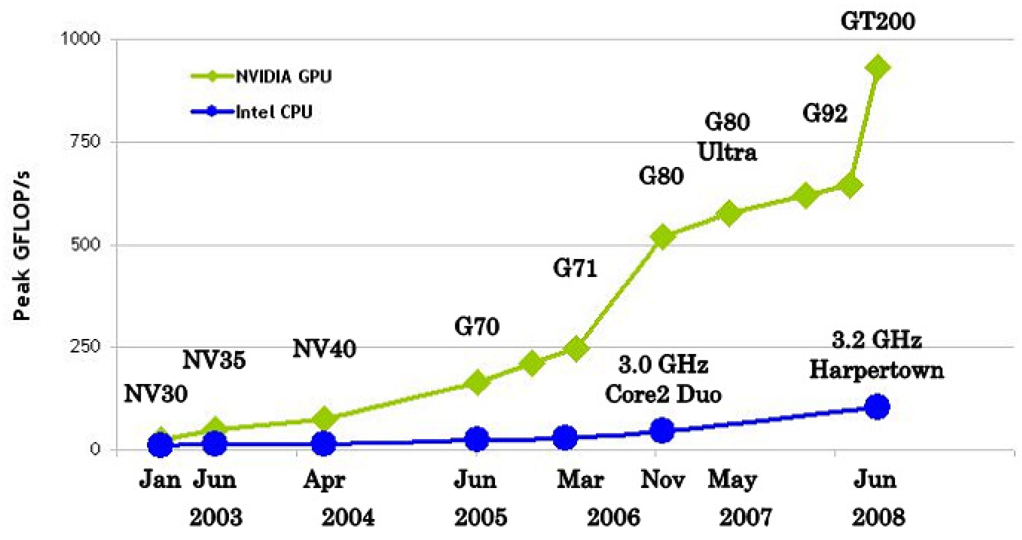
CUDA Independent Study Final Paper

Jeremy Espenshade
Michael Romero
Submitted: 11/12/08

Table of Contents

WHAT IS CUDA

Technology trends and advances in video games and graphics techniques have led to a need for extremely powerful dedicated computational hardware to perform the necessary calculations. Graphics hardware companies such as AMD/ATI and NVIDIA have developed graphics processors capable of massively parallel processing, with large throughput and memory bandwidth typically necessary for displaying high resolution graphics. However, these hardware devices have the potential to be re-purposed and used for other non-graphics-related work. NVIDIA provides a programming interface known as CUDA (Compute Unified Device Architecture) which allows direct programming of the NVIDIA hardware. Using NVIDIA devices to execute massively parallel algorithms will yield a many times speedup over sequential implementations on conventional CPUs.



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

Figure 1: Comparison of Modern GPUs vs. CPUs [2]

CUDA ARCHITECTURE:

Thread Organization

In the CUDA processing paradigm (as well as other paradigms similar to stream processing) there is a notion of a ‘kernel’. A kernel is essentially a mini-program or subroutine. Kernels are the parallel programs to be run on the device (the NVIDIA graphics card inside the host system). A number of primitive ‘threads’ will simultaneously execute a kernel program. Batches of these primitive threads are organized into ‘thread blocks’. A thread block contains a specific number of primitive threads, chosen based on the amount of available shared memory, as well as the memory access latency hiding characteristics desired. The number of threads in a thread block is also limited by the architecture to a total of 512 threads per block. Each thread within a thread block can communicate efficiently using the shared memory scoped to each thread block. Using this shared memory, all threads can also sync within a thread block. Every thread within a thread block has its own thread ID. Thread blocks are conceptually organized into 1D, 2D or 3D arrays of threads for convenience.

A ‘grid’ is a collection of thread blocks of the same thread dimensionality which all execute the same kernel. Grids are useful for computing a large number of threads in parallel since thread blocks are physically limited to only 512 threads per block. However, thread blocks within a grid may not communicate via shared memory, and consequently may not synchronize with one another.

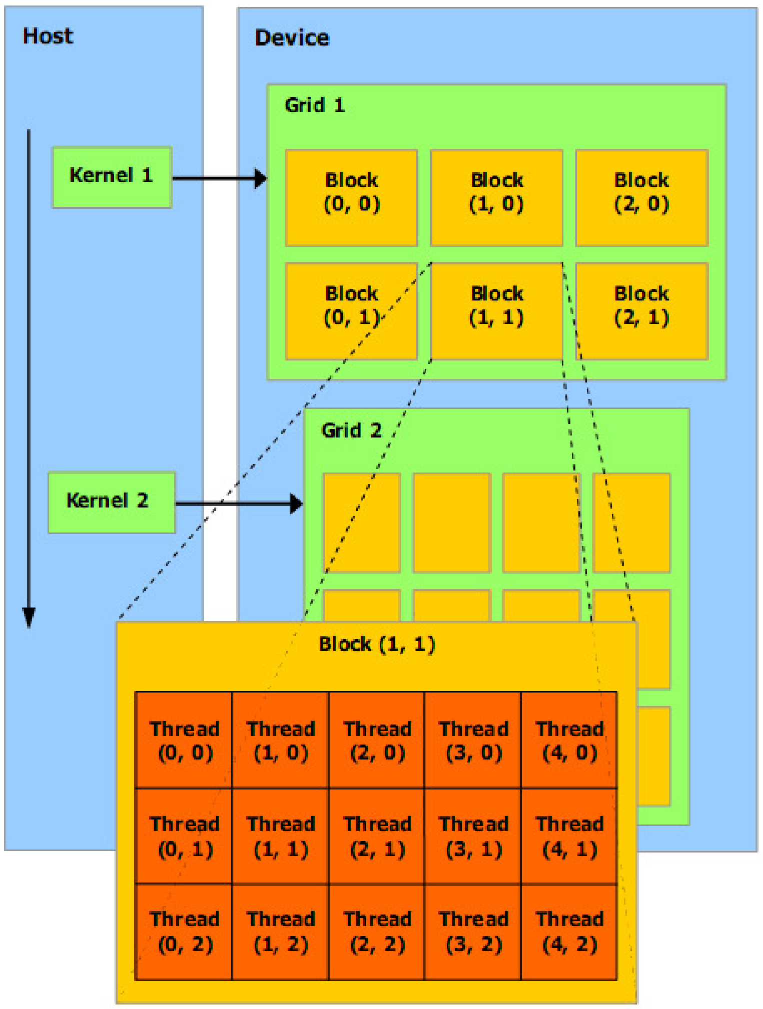


Figure 2: Thread Hierarchy [1]

Figure 2 demonstrates the thread hierarchy described. Here, kernel 1 contains a 3x2 grid of thread blocks. Each thread block is a 5x3 block of threads, for a total of 90 threads in kernel 1. Kernel 2 may contain a different organization of thread blocks, which in turn may contain an array of threads different than the arrays in the thread blocks of kernel 1.

Memory Hierarchy

There are several levels of memory on the GPU device, each with distinct read and write characteristics. Every primitive thread has access to private ‘local’ memory as well as registers. This ‘local’ memory is really a misnomer; the memory is private to the thread, but is not stored local to the thread’s registers but rather off-chip in the global GDDR memory available on the graphics card. Every thread in a thread block also has access to a unified ‘shared memory’, shared among all threads for the life of that thread block. Finally, all threads have read/write access to ‘global memory’, which is located off-chip on the main GDDR memory module which therefore has the largest capacity but is the most costly to interact with. There also exists a read-only ‘constant’ and ‘texture’ memory, in the same location as the global memory.

The global, constant and texture memory are optimized for different memory usage models. Global memory is not cached, though memory transactions may be ‘coalesced’ to hide the high memory access latency. These coalescence rules and behaviors are dependent on the particular device used. The read-only constant memory resides in the same location as global memory, but this memory may be cached. On a cache hit, regardless of the number of threads reading, the access time is that of a register access for each address being read. The read-only texture memory also resides in the same location as global memory, and is also cached. Texture memory differs from constant memory in that its caching policy specifically exploits 2D spatial locality. This is due to the use of ‘textures’ in 3D graphics; the use of 2D images to ‘texture’ the surface of 3D polygons are frequently read and benefit from caching the texture spatially.

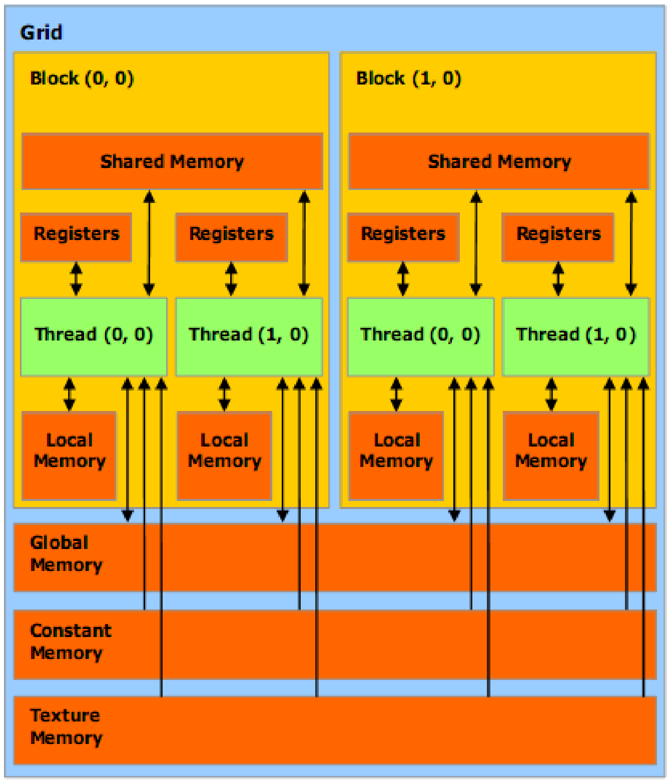


Figure 3: Memory Access Hierarchy [1]

Figure 3 shows the scope of each of the memory segments in the CUDA memory hierarchy. Registers and local memory are unique to a thread, shared memory is unique to a block, and global, constant, and texture memories exist across all blocks.

Multiprocessors

CUDA capable GPUs are constructed with the “Tesla” architecture. CUDA applications may be run on any card which supports this architecture, but each GPU device may have different specifications, and therefore a slightly different set of supported features and a different number of available computational resources. When a kernel is invoked, each thread block executes on a ‘multiprocessor’. This multiprocessor contains the resources to support a certain number of threads. Specifically, each multiprocessor consists of:

- 8 Scalar Processor cores
- 2 special function units for transcendentals
- 1 multithreaded instruction unit
- On-chip shared memory

One or more thread blocks are assigned to a multiprocessor during the execution of a kernel. The CUDA runtime handles the dynamic scheduling of thread blocks on a group of multiprocessors. The scheduler will only assign a thread block to a multiprocessor when enough resources are available to support the thread block. Each block is split into SIMD (Single-Instruction Multiple-Data) groups of threads called ‘warps’. The SIMD unit creates, manages, schedules and executes 32 threads simultaneously to create a warp. Every warp is synchronous, and therefore care must be taken to ensure that certain threads within a warp do not take abnormally longer compared to other threads in that same warp, because the warp will only execute as fast as the slowest thread. There are a number of programming hints provided in the CUDA programming guide to help prevent warp divergence.

Compute Model

Every CUDA-enabled device has a compute compatibility number. This number indicates a standard number of registers, memory size, etc. for all devices of that compatibility number. Compute compatibility numbers are backwards compatible.

	Num Multiprocessors	Compute Compatibility
Tesla C870	16	1.0
GeForce 9800GT	14	1.1
GeForce GTX260	24	1.3

Table 1: Brief Comparison of CUDA Capable Devices [1]

The most recent compute model for the GTX200 has a number of significant improvements over previous compute models, including:

- Double precision support
- Higher memory bandwidth
- Doubled the number of available registers

CUDA Programming Model

API and System Variables

To manage the thread and memory models described above, a set of API commands and system variables are provided by Nvidia. These include the Runtime API, used for managing host/device interfacing, the thread hierarchy identification variables, and several miscellaneous identifiers. The highest level of control is provided by the function type identifiers. Three such identifiers are provided: __host__, __global__, and __device__.

The __host__ identifier denotes a function to be run on the general purpose CPU, or host. This is the default type, and therefore is not typically used explicitly. Host functions can do anything a normal C++ function can do, but they also call the Runtime API functions. These functions are primarily concerned with memory management and include cudaMalloc, cudaFree, cudaMemcpy, and many deviations to work with texture memory and to provide tailored functionality. To use these functions, standard pointers are used where the pointer value is an address on the device memory rather than the host system memory.

Another Runtime API function of particular importance is kernel invocation. The syntax KernelFunctionName <<< GridDim, ThreadBlockDim >>> (@params) is used to specify the dimensions of the thread block and grid of thread blocks as described above. The kernel function is then run on the device and any memory allocated using cudaMalloc can be communicated by passing the device pointer as a parameter.

Kernel functions use the __global__ identifier. This denotes which functions may be called from the host on the device. The final identifier, __device__, denotes functions that run on the device, but may not be called from the host. The most typical use of the __host__ identifier is when a particular function is needed by both the host and device and both the __device__ and __host__ identifiers are used to instruct the compiler to construct both versions.

Within global and device functions, several system variables are used to manage threads and provide unique identification. threadIdx is a dim3 type with the unique identification of a thread within a thread block. Likewise, blockIdx provides unique identification of a thread block within a grid. With these identifiers and the __syncthreads() primitive that syncs all the threads in a thread block, execution can be effectively managed.

Shared Memory

The final important identifier is the __shared__ designator that can be applied to variables declared in device functions. This denotes that the variable should be stored in the shared memory space on the multiprocessor, which is much faster to access than any of the memories located off-chip. While using shared memory is therefore often a good idea, there are several pitfalls which can adversely affect performance when shared memory is not used effectively.

The most direct way that shared memory affects performance is that the number of concurrent thread blocks is limited by the available shared memory on each multiprocessor. The 16 kb of shared memory is split among each thread block, so if the shared memory used by each thread block is 4 kb, 4 thread blocks may run at a time. If that increases to 8 kb, only 2 thread blocks may run. This can be a mute point if the other limiting factors of threads/thread block and registers/thread already limit the number of concurrent blocks however.

More obscurely, memory access patterns can have a large impact on performance. To ensure fast execution, the shared memory is organized into 16 memory banks that can be accessed in parallel. Since each concurrently executing group of threads, or warp, contains 32 threads, these banks are accessed in two phases, each time by a half-warp of 16 threads. When each of these threads accesses a separate bank or all threads access the same element, the access is as fast as using registers, however when multiple threads attempt to access different values in the same bank, conflicts occur. The result of conflicts is that accesses must be serialized. Therefore if each thread accesses every fourth 32-bit value, all accesses will occur on 4 of the 16 banks, and 4 levels of serialization will be required, effectively decreasing performance by a factor of 4. This can be avoided by ensuring the step size between memory accesses by threads within a half-warp does not divide evenly into 16. This can be accomplished with any odd step size.

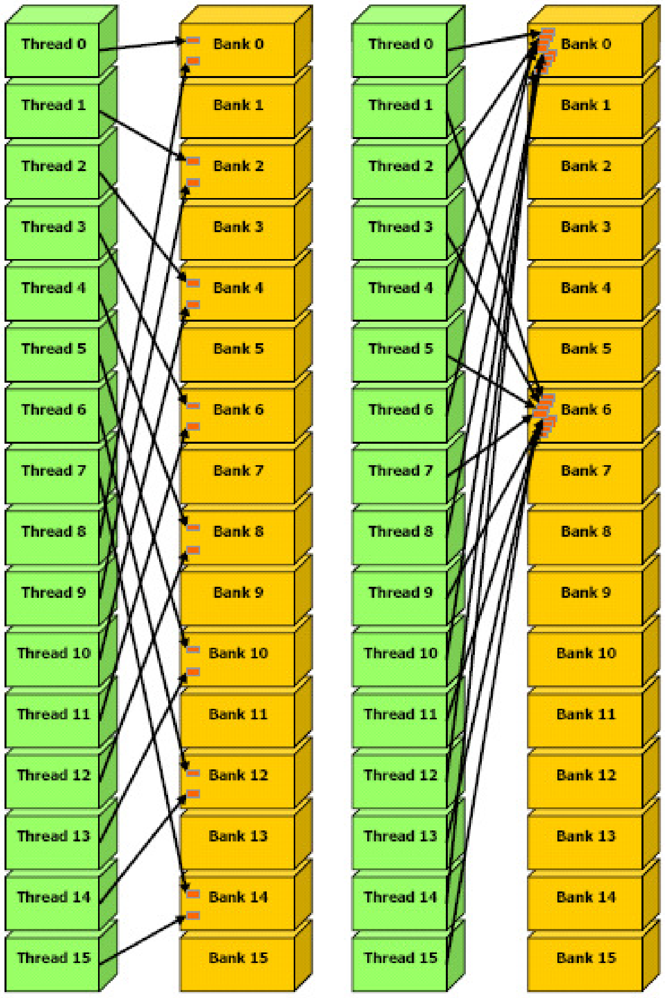


Figure 4: Bank Conflict Examples 2-way and 8-way respectively

Design Rule Checking

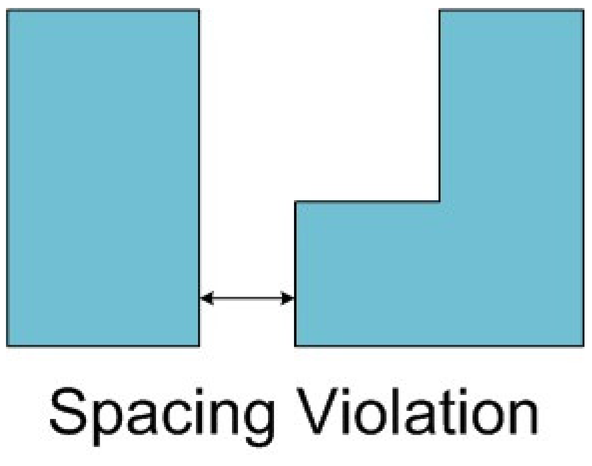
DRC Overview

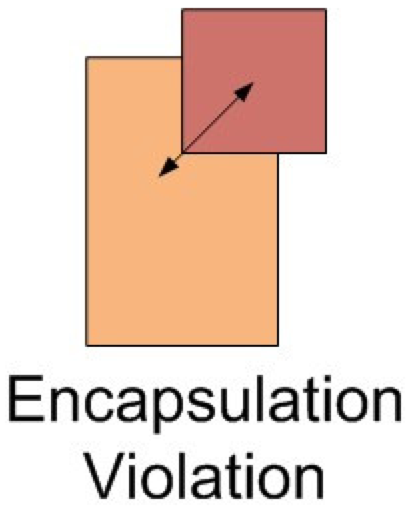
In the micro electrical engineering field, layout CAD tools are used to draw the metal, oxide and semiconductor layers that are etched into silicon to create transistors and integrated circuits. Geometric shapes are created using the layout tools, with different layers representing different metals, oxides and semiconductor materials. Layouts take place on a 2D plan, which often has a grid applied. The unit grid is based on the lambda value of the design process being utilized. The lambda value and design process are based on the gate length of the transistors created using that process. The geometric shapes created by these layout tools must adhere to a series of strict rules to ensure both proper electrical characteristics of the design, as well as the feasibility of etching the design with a given set of equipment. Design Rule Checking, or DRC is used to check the layout design against these rules.

Suitability of DRC for CUDA

Design Rule Checking was identified as an application that could benefit by a CUDA implementation because of the inherent parallelism that exists with the rules to be checked. The most common design rules used for IC layout have been identified as:

- Minimum Spacing
- Minimum Width
- Minimum Edge Length
- Encapsulation of Layer





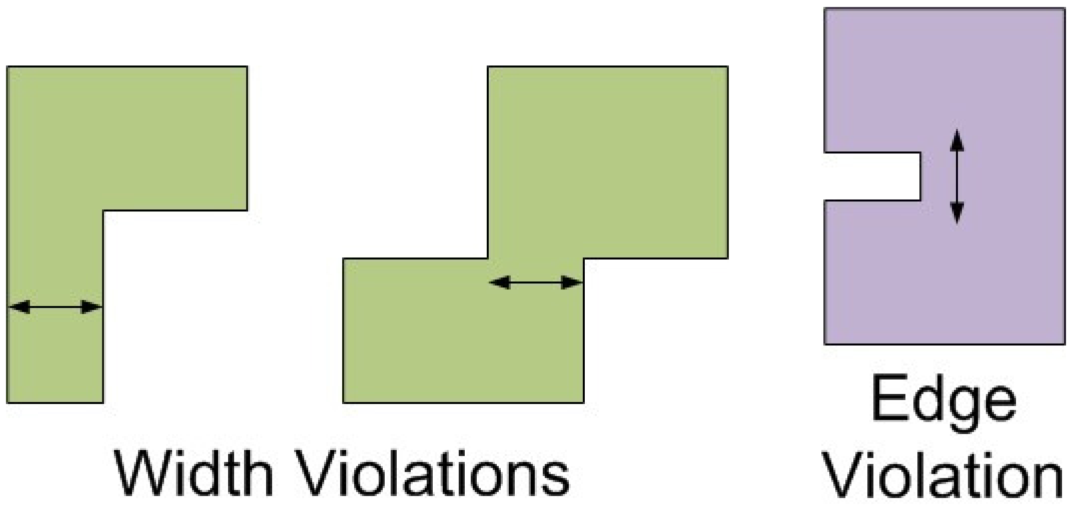


Figure 5: DRC Violation Examples

Figure 5 is a graphical depiction of the rule violations identified with certain geometry. The minimum spacing rule verifies that one layer at its nearest point is at least a minimum distance away from another particular layer. The minimum width rule verifies that the narrowest point of the given layer is at least a minimum width. The minimum edge length rule verifies that every edge of the particular geometric shape is a minimum length, ensuring that the design is not too 'detailed' for the available fabrication tools to accurately reproduce. The encapsulation rule verifies that a particular layer is encapsulated within another layer. The minimum distance and encapsulation rules are implemented by individually verifying a specific geometric shape of a particular layer with every other geometric shape of a particular layer. The minimum width and edge length rules are verified by independently analyzing every geometric shape for one particular layer. It can be observed that every rule performs a check on one or more layers independently, and therefore each check of each geometric shape can be performed in parallel

A number of industry-standard IC layout applications exist, from companies such as Mentor Graphics and Cadence Design Systems. However, each of these applications has proprietary algorithms to perform DRC checking, and proprietary internal representations of their layouts. In an effort to use open software with open DRC algorithms for comparison and open layout representations for ease of use, the Magic Layout Tool was chosen. Magic was written in the 1980's at Berkeley by John Ousterhout, and is a commonly used layout tool in academia.

The Magic Layout Tool represents layouts using rectangles. Every geometric shape is split into a series of rectangles, where each rectangle is split when the horizontal width changes. These rectangles are listed in a text file, where a layer name is followed by a series of rectangles specified by the lower left and upper right corners, where an x, y coordinate pairs describes a corner. Rules are also located in a text file, and were an inspiration for the format of rules in the CUDA DRC implementation.

Implementation

Data Structures and Memory Management

A Magic layout file is described by a list of layers, with each layer followed by the rectangles that comprise the geometric shapes for that layer. Each rectangle is specified using an x, y pair for the lower left and upper right corner of the rectangle. A rectangle struct was created as follows:

```
typedef struct{
  int x, y;
} Coordinate

typedef struct{
  Coordinate ll, ur;
  int offset;
} Rectangle;
```

The offset integer is not used in the algorithm, and is never referenced. As mentioned in the CUDA Programming Model section of this document, memory access patterns must be taken into consideration when defining data structures. If the Rectangle struct contained exactly 4 integers, only 4 of the 16 memory banks in shared memory would be used, resulting in 4 levels of serialization when each thread accesses the same member value of separate contiguous rectangles. By adding an offset, the step size is increased to five, removing the bank conflicts under typical access paterns.

It was necessary to create a data structure capable of holding all of these rectangles, with easy access to every rectangle within a particular layer. It was also advantageous to load a layout dynamically, without statically defining the number of rectangles to a layer or the number of layers in a layout. A C STL vector was used for a dynamically growing array of Rectangles. Each vector represents a layer, and all of the Rectangles in that layer are entries in the vector. A second vector was used to contain each layer vector. Ultimately, the entire layout is represented in a vector of vectors of Rectangles.

```
typedef vector<Rectangle> RectVect;

vector<RectVect> layers;
```

Once this was constructed, the problem of data locality presents itself. DRC is inherently a problem which is concerned with local data, as rectangle-defined shapes must only be checked against neighboring shapes. Therefore, checking every rectangle against every other rectangle is exceedingly inefficient and local groupings of rectangles become a valuable construct. This was implemented through a partitioning mechanism that accepts the number of regions and builds a two dimensional vector of RectVect's for each layer where the RectVect contains the rectangles in a particular spatial region.

Now that the design is represented in spatially local regions, it must be placed on the device. This proved to be a more cumbersome procedure than one would expect. Since STL vectors were used to ease the dynamic size handling of the design data, the final result was contained in a non-contiguous block of memory, which cannot be copied to the device using cudaMemcpy. To alleviate this problem, the data structure had to be flattened into a simple Rectangle array.

While simple to implement, this introduced the problem of how to index into the array such that given a layer and region, one would be able to identify both the position and length of the associated sub-array. The solution was to construct an index table that could be used to accomplish both goals. The index table was constructed by recording the number of rectangles in each region for each layer. An exclusive prefix sum was then performed on the index table so that each member contained the appropriate offset to the first rectangle in the given region and layer. By accessing both the required index and the next index, the number of rectangles could also be determined. The flattened Rectangle array and the index table could then be copied to the device memory and accessed by the execution kernels.

All rules are capable of being defined by a few characteristics. All rules have a type, such as minimum distance or minimum width. A numerical value represents the minimum distance or width. All rules act either on one layer or between two layers. Finally, all rules have a comment related to the rule in the event that the rule is violated to give the user of the layout tool some feedback on why the rule failed. The rule structure is and vector of rules are as follows:

```
typedef struct{
  string type;
  int value;
  int layerA;
  int layerB;
  string comment;
} Rule;

vector<RectVect> layers;
```

Similar to the vectors used to store a dynamic number of rectangles, a vector was constructed to hold a dynamic number of rules. Again, the rule list was required to be dynamic, since the rules file is read at load time, and may vary greatly from one design rule check to another. Fortunately, it was not necessary to send the vector of rules to the device. The vector is simply iterated through by a checkRules() function, which parses the rules and calls their respective kernels with the appropriate parameters.

Parallel Algorithms

Once the data structures had been defined and loaded into the device memory, the design rule checks could be performed. In all the following algorithms, each thread is responsible for a single rectangle at a time. This allows all threads to follow the same memory access patterns and execution steps, which supports the shared memory access patterns and warp-level SIMD execution model.

Several levels of parallelism are exhibited by the problem and exploited by the algorithm. First, task parallelism is available by way of the independent spatial regions. Because there is no need for synchronization or data sharing between regions, separate thread blocks can be assigned to each region. This results in a grid of thread blocks analogous to the grid of regions in the design. An added benefit of constructing separate thread blocks for each region is the inherent dynamic scheduling that results. Because different numbers of rectangles will exist in each region, thread blocks will finish execution at different times resulting in an unbalanced work load. However, with a larger number of thread blocks than can be executed concurrently, completed regions are simply replaced by yet to be computed regions for much of the execution time. A reasonably large design should have hundreds or thousands of regions, and typically only 32 thread blocks will be executing concurrently, resulting in effective use of this feature.

The second level of parallelism is data level parallelism, which is naturally exploited by the threads in each thread block. Each thread computes the same instructions on different data and requires no synchronization. DRC could be considered an embarrassingly parallel problem due to the lack of data dependencies anywhere in the program flow. This is also well suited to the CUDA SIMT model, as it makes use of the base execution case without concern for thread synchronization.

Performance Analysis

How Data was Collected

The purpose of this investigation was to demonstrate the performance improvement possible with GPGPU computing for a common electronic design automation application, DRC. At the same time, it is understood that implementation is at a pre-prototype level and certain assumptions must be made to extract meaningful results.

The primary concern in this implementation is the exceedingly long time required to build the complex set of dynamically sized vectors representing the design. This process takes much longer than the actual design rule checks and would overshadow the differentiating performance data if included in both CPU and GPU timing statistics. Based on the assumption that any DRC tool would have an internal memory representation and such a representation would not have to be dynamically built, that portion of execution was ignored. The portion of execution that was included was the actual design rule checks and looping structures.

Performance data was gathered from reference designs in MAG file format. To vary the design size, a design duplication script was used. This allowed relatively consistent increases in complexity and computation without the additional variability of changing designs. Often the DRC checks were so fast (especially for the GPU) that the 10 ms unit of measurement was too large, so checks were performed 100 times and the total time was recorded. For each design, the DRC was performed on both the CPU and GPU and performance results were compared.

Results

Many variations are possible including the number of regions, size of design, and number of threads/thread block. To provide some direction, the first performance data to be gathered used a set number of 100 regions. With that constant, the size of the design was then changed to vary the number of rectangles in each region and the number of threads per thread block was varied to identify an optimal configuration.

Maximum Rectangles/Region	**cpu**	**gpu - 384**	**gpu - 256**	**gpu - 192**	**gpu - 128**	**gpu - 64**
(Rects)	**(ms)**	**(ms)**	**(ms)**	**(ms)**	**(ms)**	**(ms)**
6	110	1100	1110	500	370	240
15	410	1220	1210	580	440	310
32	940	1500	1500	720	600	500
86	9240	2570	2100	1380	1250	1290
293	120280	6720	6630	5580	5920	9020

Table 2: Execution Time Results

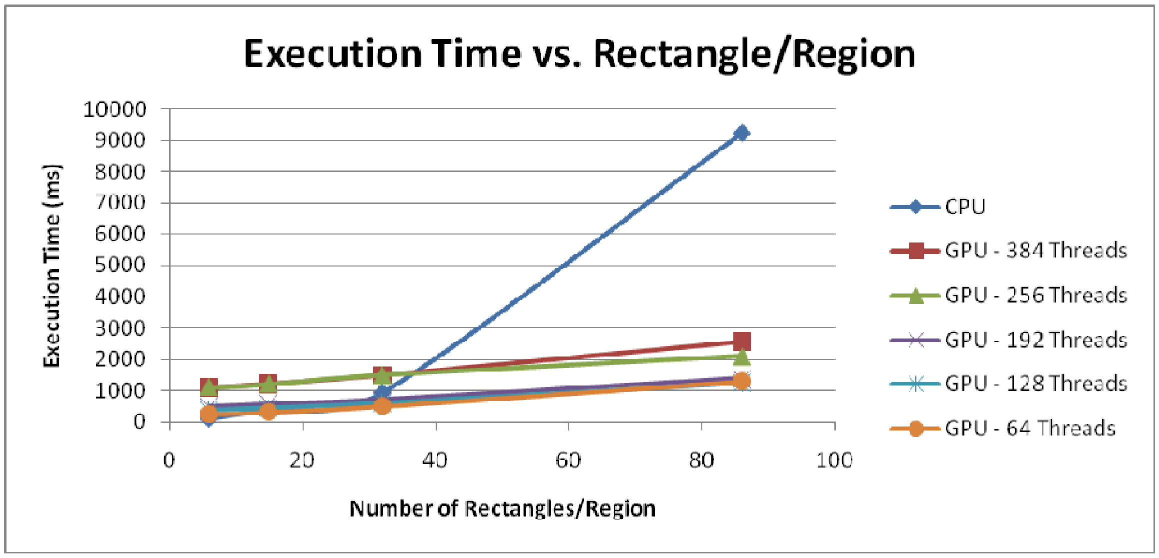


Figure 6: Execution Time Trend

From Table 2 and Figure 6, an important characteristic of GPU computing is demonstrated. Because of the overhead of thread creation and the large computational capability of the GPU, a design size must be sufficiently large to fully utilize the hardware and demonstrate speedup. Since each thread is responsible for a single rectangle, the cases with fewer than 36 rectangles per region result in many threads running idle and the CPU outperforming the GPU. Once 36 rectangles are used, enough threads are active to overcome that deficiency, but only when many thread blocks can be run in parallel, favoring smaller numbers of threads/thread block. However, beyond 36, enough threads become active to exceed CPU performance in all cases.

Maximum Rectangles/Region	gpu - 384	gpu - 256	gpu - 192	gpu - 128	gpu - 64
(Rects)					
6	0.1	0.099099	0.22	0.297297	0.458333
15	0.336066	0.338843	0.706897	0.931818	1.322581
32	0.626667	0.626667	1.305556	1.566667	1.88
86	3.595331	4.4	6.695652	7.392	7.162791
293	17.89881	18.14178	21.55556	20.31757	13.33481

Table 3: Speedup Results

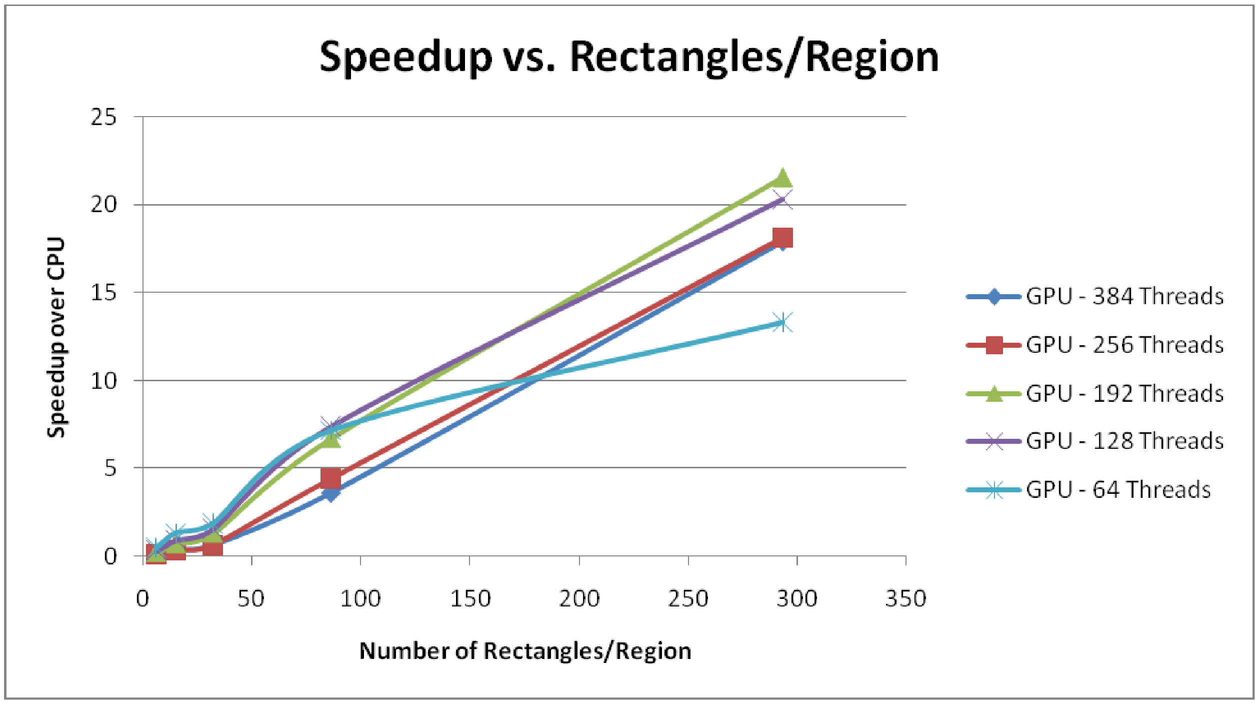


Figure 7: Speedup Trends

Beyond demonstrating the scale required to make full use of the GPU hardware and execution model, this set of performance data was useful in determining the optimal number of threads. Figure 7 shows the change in speedup as the number of rectangles increases and identifies 192 threads per thread block as an optimal configuration. This is an interesting result, as one might expect that having only a single rectangle for each thread would be preferable to having each thread be responsible for multiple rectangles. However, it appears that multiple rectangles actually provides a better load balance between threads and along with decreased thread management overhead causes an increase in performance.

Regions	Rectangles	CPU Time (ms)	GPU Time (ms)	Speedup
4	376	10150	1910	5.314136
16	384	40840	1980	20.62626
36	381	80560	4340	18.56221
81	352	152070	6440	23.61335
196	335	284950	10780	26.43321
324	354	604050	21170	28.5333

Table 4: Design Scaling Performance

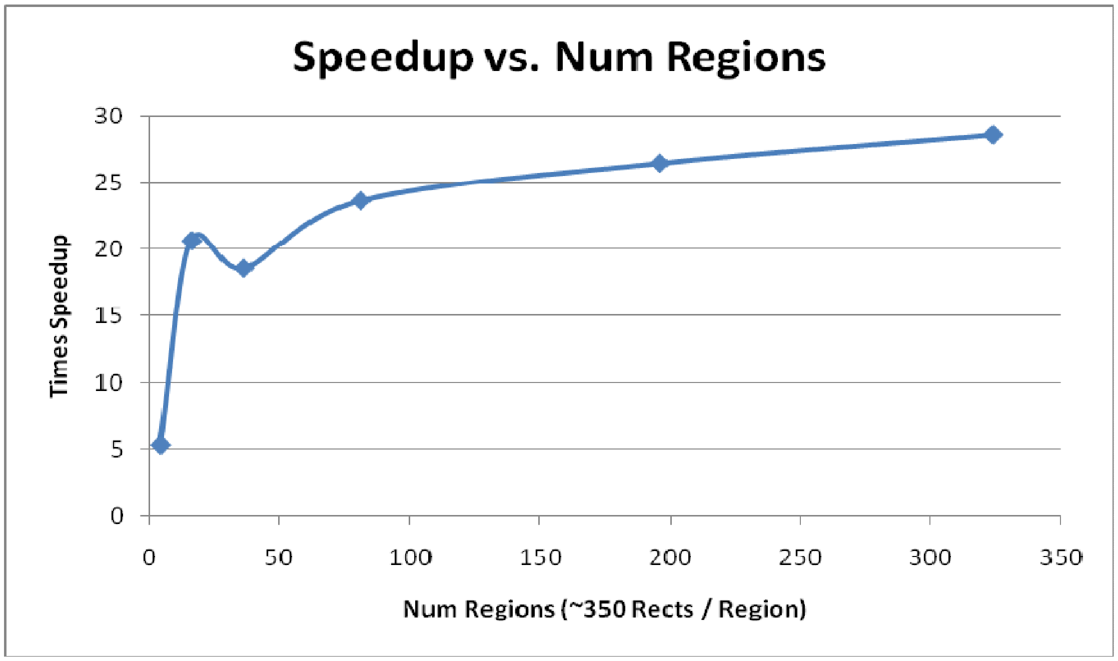


Figure 8: Speedup vs. Number of Regions (~350 Rectangles / Region)

Once this was determined, the number of threads per thread block could be held constant and the size of the design changed to demonstrate how performance scales. To scale the design size, the reference design was duplicated spatially and divided into increasing numbers of region such that approximately 350 rectangles are included in each region. This makes sure that the shared memory space capable of holding 384 rectangles has high utilization and there is sufficient work to be done by each thread. Figure 8 shows how the increasing number of regions will, in general, increase speedup logarithmically with an asymptote near 30 times speedup. There is a dip in speedup when 36 regions were chosen. This is because with 36 regions, 2 thread blocks completely consume the resources of each of the 16 microprocessors, leaving 4 regions remaining to be computed once the resources on the microprocessors are freed. Since the penalty of waiting for these free resources is significant enough, the amount of remaining work to be performed (only 4 remaining regions), is not large enough to provide comparably the same trend in speedup as the same benchmark with 16 regions or 81 regions. The highest demonstrated speedup was 28.53 times.

Conclusion

The objective of this investigation was to investigate the Nvidia CUDA framework for GPU computing and demonstrate the potential for large performance gains using the massively parallel execution model enabled. To accomplish this, a common Electronic Design Automation application called Design Rule Checking (DRC), which verifies that a design can be fabricated with a given technology, was implemented. DRC proved to be well suited, exhibiting task and data level parallelism which was exploited via the effective use of multiple thread blocks containing multiple threads. An optimal configuration using 192 threads in each thread block with a separate thread block for each spatial region of the design was identified and a speedup of 28.53 times was obtained.

Future Work

Testing:

The DRC algorithms have been verified using a number of rules on a particular layout file. To create a larger layout, the initial layout was copied and pasted to replicate the existing design, and spread across a larger design space to create more work for the design rule checker. Validation of both the GPU parallel algorithms as well as the CPU sequential algorithms would benefit greatly from further testing using other layouts, against additional rules.

Performance Comparison:

For the purposes of showing accurate results, essentially the same algorithm logic was used for both the GPU parallel algorithms and the CPU sequential algorithms. While this provided a good basis for producing speedup results, it would also be beneficial to observe a real-world performance comparison against other industry standard DRC applications.

Algorithms:

Each thread was given an entire rectangle to check in each of the rule checking kernels. Some kernels, especially the corner spacing and corner overlap checks, further parallelism is available by dividing each rectangle into independent corners. If each thread were able to check a corner rather than a whole rectangle, fewer registers would be required and additional work could be done in parallel. Special attention would have to be paid to warp organization for this model however. As the instructions required for each corner check vary slightly, simply letting the first four threads check the four corner of the first rectangle would cause greatly decreased performance by splitting the warp into four distinct SIMD groups.

Data Structure Creation:

The main limiting factor of design size and program flexibility is the data structure creation, especially the partitioning step. As mentioned, this takes a very long time for large designs, limiting the design sizes that can be checked in a timely fashion. More importantly, the number of regions must be varied along with the design size to ensure the number of rectangles in any given region does not exceed some set amount. There is assuredly a method for automatically handling this restriction that would be beneficial for future work.

Appendices

Programmer Guide

The project is contained in the NVIDIA_CUDA_SDK/projects/drc directory. The following files make up the project:

• Drc.cpp	– Host functions and main program entry point
• Drc.cu	– Functions that make CUDA calls to the device
• Drc.h	– Type definitions, method signatures
• Partition.cpp	– Functions used for partitioning of data before sending to device

The project can be compiled by typing “make” into the command line in the drc directory. This will place the drc binary in the NVIDIA_CUDA_SDK/bin/linux/release directory. It is also possible to compile the project using the emu and debug flags. Compiling the project using emulation mode, where the device behavior is emulated entirely on the host is performed by typing “make emu=1”. This places the binary executable in the NVIDIA_CUDA_SDK/bin/linux/emurelease directory.

A number of flags have been built into the program. Flags are located in Drc.h, and the program must be recompiled once the flags are changed. The following lists details the flags:

• NUM_TB	– NUM_TB x NUM_TB number of thread blocks
• NUM_THREADS	– Number of threads executing per thread block
• MAX_RECTANGLES	– Maximum number of rectangles per thread block
• CPU_ONLY	– Execute only on the CPU (1 for yes, 0 for no)
• GPU_ONLY	– Execute only on the GPU (1 for yes, 0 for no)
• LOOP_ITERATIONS	– Number of times to perform drc for timing

User Guide

Visit **www.nvidia.com/cuda** and download and install the CUDA driver and the CUDA toolkit for the given operating system.
Navigate to NVIDIA_CUDA_SDK/bin/linux/release and execute the DRC program using the following command:

```
./drc[version] <layout_file.mag> <rules_file.tech>
```

Three demos have been prepared to show functionality and speedup. A DRC clean rules/design combination can be viewed by executing:

```
./drc1 layouts/tut11d.mag rules/rules_demo_pass.tech
```

A failing rules/design combination can be viewed by executing:

```
./drc1 layouts/tut11d.mag rules/rules_demo_fail.tech
```

Finally, a representative speedup with 81 regions can be viewed by executing:

```
./drc2 layouts/tut3d5.mag rules/rules2.tech
```

Bibliography and References

Bibliography

[1] NVIDIA, NVIDIA CUDA Programming Guide, 1.0 edition, June 2007. Available: www.nvidia.com/cuda

[2] NVIDIA, NVIDIA CUDA Programming Guide, 2.0 edition, June 2008. Available: www.nvidia.com/cuda

References

Further CUDA Resources:

www.nvidia.com/cuda
www.gpgpu.org
http://courses.ece.uiuc.edu/ece498/al1/
http://grid.rit.edu/seminar/docu.php/grid:tesla#learning_cuda

Magic Layout Tool and DRC Resources:

http://opencircuitdesign.com/magic/