

GPGPU COURSE USING GPUS FOR GENERAL COMPUTATION



**SDSC, San Diego
March 5 - 6 2009
Wagenbreth**

How to use a GPU to speed up a program

- **NVIDIA GPU**
- **Basic steps - first 50%**
- **How to analyze code**
- **What to put on the GPU**
- **How to put it on GPU**
- **How to use CUDA**



Survey

- **C**
- **GPU**
- **Linux**
- **Laptop putty/ssh**
- **Small algorithm/problem for GPU**



- **wireless network**
- **mega-bug.isi.edu**
- **20 accounts gpu01 – gpu20**
- **NVIDIA 8800 GTS card**
 - **loaner from NVIDIA over a year old**
 - **768 mbyte global memory**
- **NVIDIA CUDA Programming Guide**





Instructor

Gene Wagenbreth

Gene Wagenbreth has been in the field of parallel processing since 1969. I am not an expert on NVIDIA or GPUs. I have been able to use the NVIDIA GPU's without extensive training and without becoming an expert.



Schedule

Thursday - Friday

9 am – 5 pm

lunch 12 -1

breaks every hour and a half or so

Thursday – introduction to GPGPU, NVIDIA, CUDA.

Write, compile and run a program

Threads

Memory

Friday – more detail and example code.

User code ???



What will be covered

CUDA

Linux

projects

compiling

running

host code

kernel code

CUDA scaffolding/macros

timing

emulation

debugging

global/device memory

shared memory

threads

thread blocks

performance

libraries – linear algebra, fft

basic optimizations – first 50%



What will not be covered

Windows

MAC

constant memory

texture memory

coalescing

fast arithmetic instructions

other languages – FORTRAN JAVA C++

advanced optimizations – last 50%

details of specific NVIDIA models

pricing

drivers – installation

OpenGL





History of GPU 1

Historically there has always (1960's - now) been a market for a device to provide specialized high speed processing to either reduce the cost per computation or increase the maximum speed available, or both.

FPS boxes

MAP array processors

SKY vector processors

hardware FFT boxes

signal analyzers

FPGAs

Almost all such products utilized parallelism. Instead of using half the time and half the hardware and half the cost to decode instructions and other functions, use 90-99% of the time, hardware and cost to do calculations.

History of GPU 2

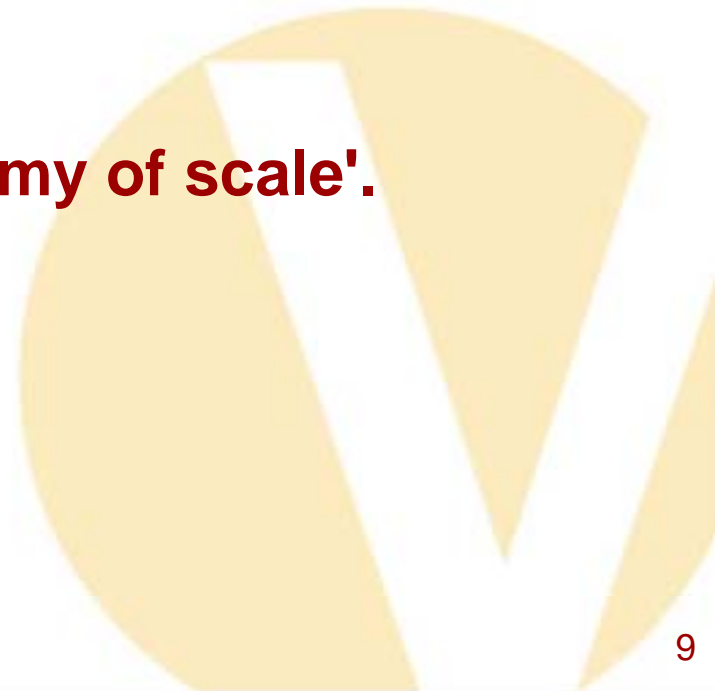
Only applicable to a minority of codes.

More difficult to program than conventional processors.

Primarily used by scientific users.

Expensive due to the lack of 'economy of scale'.

(GPU changes this)





No More Free Lunch

An alternative has been to wait a few years and take advantage of the reduced cost and increased speed (clock rate and FLOPS) available. Very little effort on the part of the programmer. Get a factor of 1000 by waiting 10 years. The free lunch is over. Clock rates and FLOPS are now increasing only marginally. Programs do not automatically speed up by waiting. CPU makers have applied more and more transistors to speeding up serial code by other means such as speculative execution, branch prediction and many other 'tricks'.

GPU Characteristics

GPUs traditionally have special hardware to do graphics calculations quickly

dedicated graphics rendering device

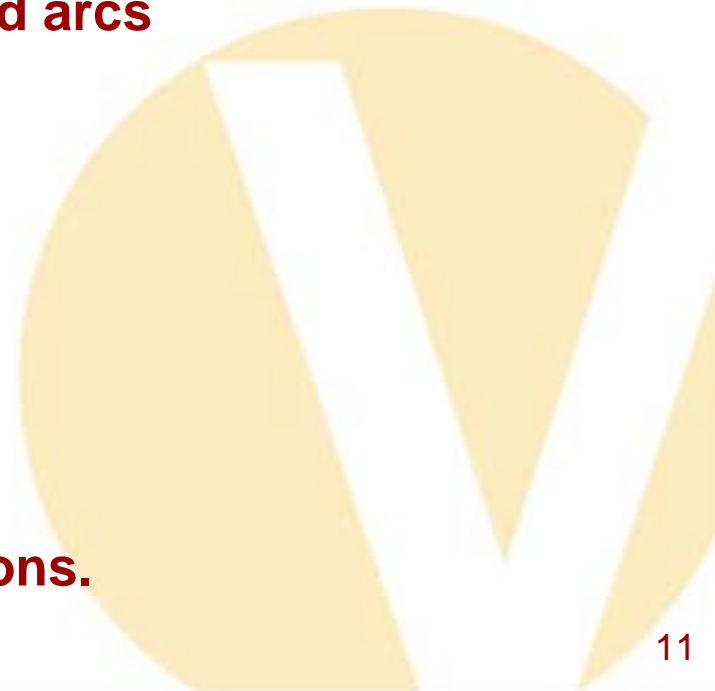
**BitBLT - combines several bitmap patterns using a RasterOp
drawing rectangles, triangles, circles, and arcs**

3D computer graphics

fast memory

parallel specialized functional units

GPU's devote more transistors to calculations.



GPU Choices

NVIDIA

ATI

Cell - microprocessor architecture jointly developed by Sony Computer Entertainment, Toshiba, and IBM, an alliance known as "STI"

New multicore processors MAY render GPU's irrelevant.

GPGPU - General Purpose Graphic Processing Unit

'power users' have always been willing to do whatever it takes to get programs to run fast.

GPUs are a 'free' resource

used assembly language, microcode or OpenGL to program GPU

Now GPU makers are accessing the market by making it more convenient to use a GPU for non graphics programming.

NVIDIA GPU

- **need a big power supply – 1000W ?**
- **need lots of cooling**



GPGPU 2

Game players and the more visual world of the internet and interactive use provide 'economy of scale'. The GPGPU user gets a 'free ride'.

(FPGAs do not have economy of scale. More expensive)

A typical GPU will deliver 300 GFLOPS peak performance, vs 3 GFLOPS for a single core CPU or 10 GFLOPS multicore.

(SSE and more cores may increase CPU flops)

ALL new fast hardware suffers from a large imbalance between memory bandwidth and available FLOPS. Efficient cache use, automatic or manual, is required for efficient use.

Amdahls Law

If a calculation consists of a scalar portion S and a parallel portion P , speeding up the parallel portion can only reduce the parallel portion to zero, leaving the scalar portion. If S is $1/10$, you can not get a more than a factor of 10.

code takes 100 seconds. scalar portion is 10 seconds, parallel is 90 seconds. IF GPU speeds up parallel portion by a factor of 90, parallel portion is 1 second. Total time using GPU is 11 seconds and speedup is a factor of 9, even though GPU speeded up parallel portion by 90.

So even if GPU is run at 300 GFLOPS, total code speedup is unlikely to be spectacular. A factor of 2-10 is more realistic.



Amdahls Law - Scaling

Weak and Strong

Increasing the problem size can circumvent Amdahls law. If the example is a 3d problem $N \times N \times N$ and the amount of time spent in the scalar portion is porportional to N and the of time in the parallel portion is proportional to $N \times N \times N$, increasing N by a factor of 2 gives:

without GPU:

original: $S=10 + P=90$	100 seconds
S takes twice as long - $10 \times 2 =$	20
P takes eight times as long - $90 \times 8 =$	720
total time	740

with GPU:

original: $S=10 + P= 1$	11 seconds	9X
S takes twice as long - $10 \times 2 =$	20	
P takes eight times as long - $1 \times 8 =$	9	
total time	29	26X





Amdahls Law - Scaling 2

small problem speedup = $100/11 = 9$

large problem speedup = $740/29 = 25$

Further increase in problem size makes things even better.

IF GPU is 100 times faster than host, getting a better GPU that is 1000 times faster than host often doesn't help.



Course Approach

I am not an NVIDIA/CUDA expert

We will try to teach:

A little about parallel algorithms

Mechanics of using NVIDIA/CUDA

How to implement some example parallel algorithms on NVIDIA GPU

How to do high level tuning for performance.

We will not try to cover:

numbers on clockrate, memory size, bandwidth etc will be approximate. They depend on model and are changing quickly

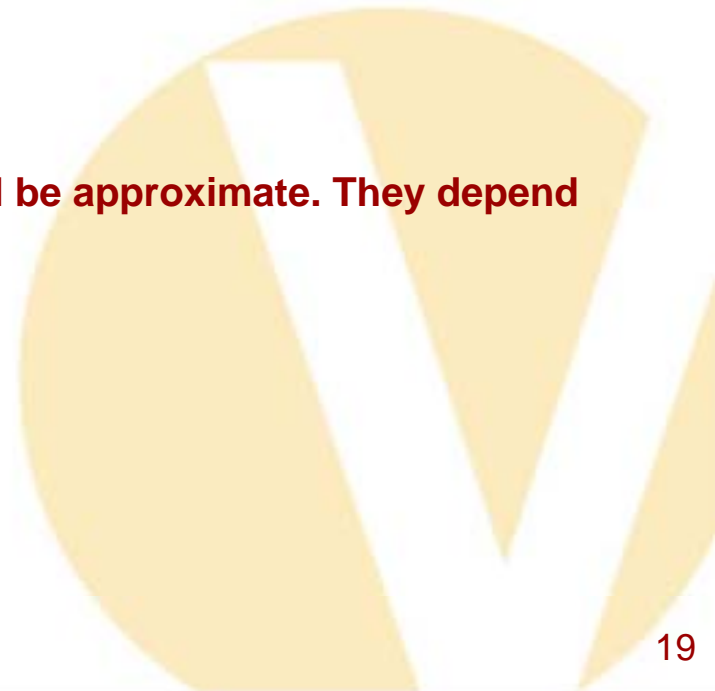
detailed optimization techniques to get the last 10-50%

memory coalescing

constant memory

texture memory

reduced accuracy arithmetic





Course Approach 2

GPU hardware

GPU Software

Code Conversion

Examples

simple cpu example

simple shared memory example

matrix transpose

matrix multiply

route finding

User Examples





How to Convert a Program to Use a GPU

Only a small part of a program should be moved to the GPU

90/10 rule - typically 90% of the time is spent in 10% of the code

Identify cpu intensive part of code

knowledge of the code and algorithm

use timings

tools such as gprof





Data Size vs Computations

Matrix Multiply

Problem must be large enough and cpu intensive enough for GPU

Matrix multiply - $3*N*N$ words of data
- $2*N*N*N$ computations

transfer between host and GPU – 500 mbytes/sec

computation – 100 gflops

N	transfer time	computation time
100	.0002	.00002
1000	.02	.02
10000	2.0	20.0



Data Size vs Computations

FFT

Problem must be large enough and cpu intensive enough for GPU

- FFT - $2*N$ words of data**
- $12*N*\text{LOG}_2(N)$ computations**

transfer between host and GPU – 500 mbytes/sec

computation – 100 gflops

N	transfer time	computation time
1024	.00002	.000001
4096	.00006	.000006
65536	.001	.0001
1000000	.016	.002



Data Size vs Computations General

Problem must be large enough and cpu intensive enough for GPU

500 mbytes/second = 125 mwords/second

100 gflops = 800 calculations for each word transferred





NVIDIA – CUDA

We are not NVIDIA only advocates. At this time NVIDIA and CUDA provide the most convenient and stable environment for GPGPU.

Other software and approaches are available to run on other GPU's, or to run on all GPU's.

In the past programming a GPU meant using a special purpose low level language that was subject to change or replacement as the hardware changed every few years.

CUDA is available for free from nvidia.com

programming manual and extensive training materials are available at nvidia.com

CUDA

CUDA is a high level language. NVIDIA is committed to supporting CUDA as hardware changes. Hardware is projected to change radically in the future. Primarily, the processor count may go from hundreds to tens of thousands. Program algorithm, architecture and source code can remain largely unchanged. Increase problem size to use more processors. Increase a 3D grid by a factor of 5 to go from hundreds to tens of thousands of processors.



CUDA Advantages

NVIDIA promises to support CUDA for the foreseeable future

CUDA encapsulates hardware model

you dont have to worry about hardware model changes

all the conveniences of C vs assembly

CUDA emulation mode allows debugging on the host with print statements or dbg

you can experiment !!!!!

CUDA Conversion

Learning the hardware and developing parallel algorithms is still difficult. But the infrastructure for writing, developing, debugging and maintaining source code is straight forward and similar to conventional serial programming.

DEVELOPING PARALLEL ALGORITHMS IS DIFFICULT. NEITHER THIS COURSE NOR CUDA MAKES IT EASY !!!!

Software, manuals, examples available for free at NVIDIA web site.

**Linux and Windows - we will be using Linux
MAC too I think**



Considerations for Efficient Use of GPU

Lots of calculations

Parallelism for 100s or 1000s of threads

SIMD - few conditionals

Memory





Lots of Calculations

Lots of Calculations

300 GFLOPS for a minute = 18 trillion operations

300 GFLOPS for an hour = one quadrillion operations

NVIDIA GPU is completely or partially single precision -
32 bit

if all calculations must be double precision - 64 bit -
FLOP rate is much lower

arithmetic is not IEEE standard - Small differences

Parallelism

100s or 1000s of threads

Data parallelism

2D, 3D or higher dimensionality arrays



SIMD

Single Instruction Multiple Data

All processing elements in a processor execute
same instructions in lockstep

Conditionals are executed by all PE's. Results
are not stored for some PE's

`IF(A[I] != 0.0)A[i]=1.0/A[I]`

Too many conditionals kill performance



Memory

GPU has (currently) approximately 1 gbyte memory

completely random memory access greatly lowers efficiency

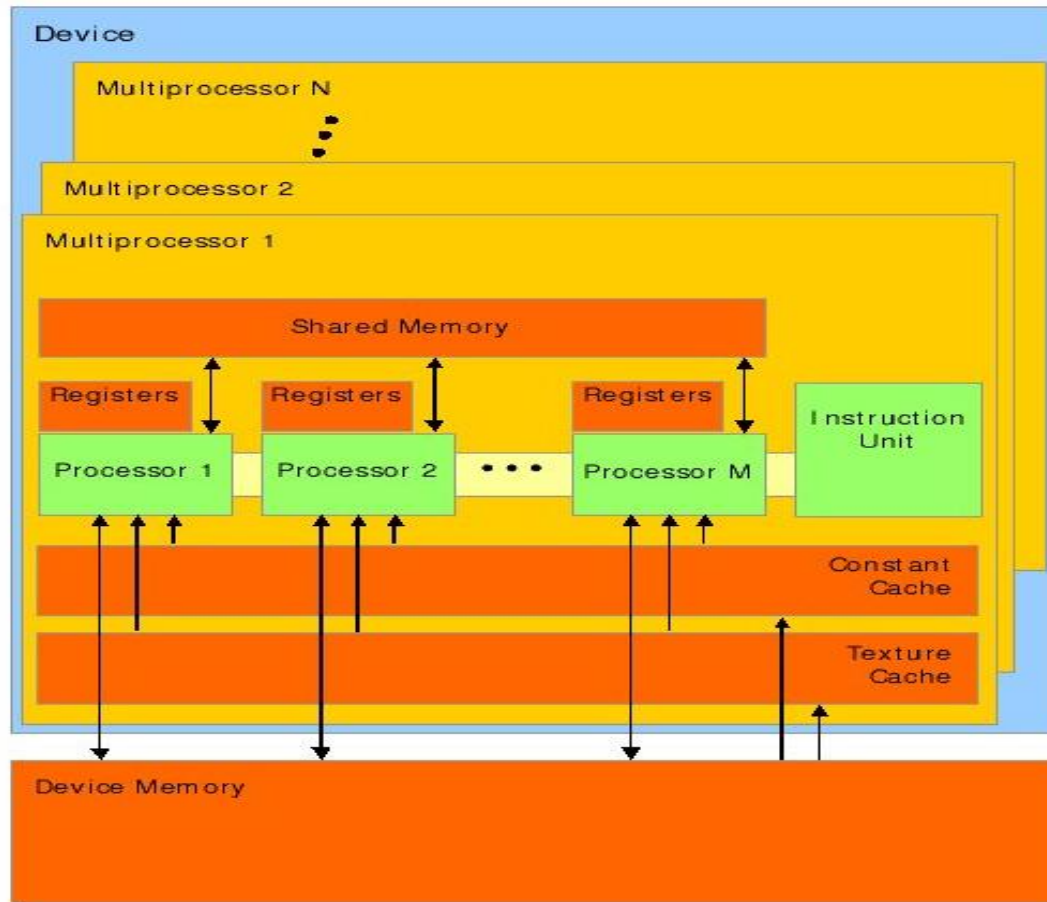
use of shared memory as cache for global memory can dramatically increase efficiency

Page 27 of NVIDIA CUDA Programming Guide 1.1

3.1 A Set of SIMD Multiprocessors with On-Chip Shared Memory

The device is implemented as a set of multiprocessors as illustrated in Figure 3-1. Each multiprocessor has a Single Instruction, Multiple Data architecture (SIMD): At any given clock cycle, each processor of the multiprocessor executes the same instruction, but operates on different data. Each multiprocessor has on-chip memory of the four following types: One set of local 32-bit registers per processor, A parallel data cache or shared memory that is shared by all the processors and implements the shared memory space, A read-only constant cache that is shared by all the processors and speeds up reads from the constant memory space, which is implemented as a read-only region of device memory, A read-only texture cache that is shared by all the processors and speeds up reads from the texture memory space, which is implemented as a read-only region of device memory. The local and global memory spaces are implemented as read-write regions of device memory and are not cached.

Hardware Architecture



A set of SIMD multiprocessors with on-chip shared memory.

Figure 3-1. Hardware Model

Hardware – CPU

128 processors - 16 multiprocessors, each with 8 SIMD processing elements

relatively slow clock (less than a ghertz)

arithmetic is 32 bit - no double precision or restricted double precision. this may improve with time.

Hardware – Memory

Three levels of memory

local memory - very fast. registers, i,j,k. not shared by PE's

shared memory - shared by PE's in a processor. Not shared by processors

fast if accessed reasonably correctly.

relatively small - 32 kbytes per multiprocessor

global memory - shared by all processors. Slow - long (device memory) latency. Can be masked by efficient access pattern and multi-threading.

Relatively large - 1 gbyte

shared memory is a user controlled cache

Multi-Threading

important feature for speed

switch contexts between clocks/instructions

threads are switched in blocks

can mask memory latency

pipeline concept

WARPS – 4 threads per processing element



Multi-Threading 2

**efficient multi-threading is the key to efficiency on the
NVIDIA GPU**

each multiprocessor has 8-16 processing elements

**on most modern CPUs memory is much slower than
arithmetic**

fast cache is the solution to slow memory

it is still difficult to keep CPU busy

processing is free (??!!)

Multi-Threading 3

Multi-Threading with Fast Context Switch

THIS IS THE SOURCE OF NVIDIA GPU EFFICIENCY

NVIDIA GPU provides hardware support for multithreading

**when processing elements stall thread scheduler can
switch context to another thread**

**each batch of threads has its own set of registers
and its own location counter**

can switch thread context BETWEEN clocks

Multi-Threading 4

give 128 processing elements 1024 threads

each of the 16 multiprocessors has 64 threads to execute on its 8 processing elements

8 groups of 8 threads

initiate an instruction for 8 threads. If it stalls, initiate an instruction for a different 8 threads

the result is that if memory is not used too badly, each of the 128 processing elements completes one instruction per clock (2 per clock)

Memory Speed

memory has two important specifications

latency - how long after a memory read is issued does the data reach the CPU. Typically many clocks

bandwidth - what is the maximum rate that data can flow between memory and the CPU

fast memory is expensive

CPU and GPU - memory is the bottleneck.

multithreading masks latency, not bandwidth.

Memory and Cache

the solution to the memory bottleneck on modern systems is to have a small amount of very fast memory. 512 kbyte of cache vs a gbyte of main memory (2000X)

multiple levels of cache on most CPU's

access memory with good patterns and a small amount of fast cache makes all memory appear to be as fast as cache

Memory and Cache 2

**this description is generally true on general CPU –
AMD,INTEL**

**cache is organized in cache lines of various sizes. assume
32 byte (8 word) cache size.**

**if a byte of a cache line is read, all 32 bytes are copied
from memory to cache.**

**if a byte is already in cache, transfer to the CPU is very
fast.**



Memory and Cache Access Patterns

Access memory contiguously

need not be absolute. 1-4-2-3 is as good as 1-2-3-4

Reuse already accessed memory in 'small' blocks

Software optimization – blocking





NVIDIA GPU Shared Memory

NVIDIA GPU shared memory is a user controlled cache.

There is no automatic cache.

Makes coding more difficult because you have to pay attention.

Makes it possible to optimize access

Software technique called blocking



Software and Programming Model

CUDA is C with a few straight forward extensions.

The extensions to the C programming language are four-fold:

Function type qualifiers to specify whether a function executes on the host or on the device and whether it is callable from the host or from the device `__global__`

Variable type qualifiers to specify the memory location of a variable `__device__`, `__shared__`

A new directive to specify how a kernel is executed on the device from the host

Four built-in variables that specify the grid and block dimensions and the block and thread indices

`gridDim` `blockIdx` `blockDim` `threadIdx`



NVIDIA SDK comes with many sample projects

I use the structure, makefile, macros, etc of the sample projects when writing my own code.

Fairly simple and straight forward.

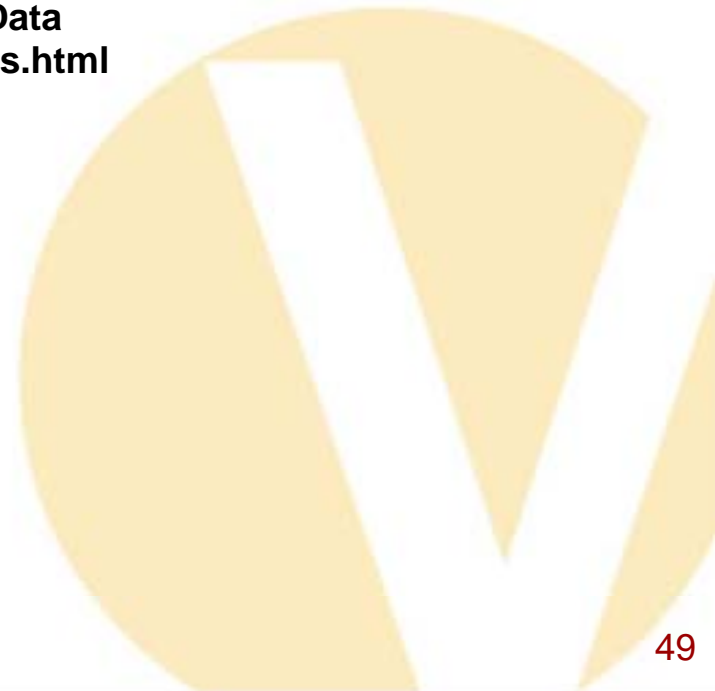
Easiest way to get your first code running





Directory Structure of SDK

```
[genew@mega-bug ~]$ pwd
/home/genew
[genew@mega-bug ~]$ ls -lt NVIDIA_CUDA_SDK
total 88
drwxrwxr-x 6 genew users 4096 2009-02-17 18:15 common
drwxrwxr-x 64 genew users 4096 2009-01-23 15:32 projects
drwxrwxr-x 2 genew users 60 2009-01-23 15:31 lib
drwxrwxr-x 2 genew users 42 2008-04-18 12:02 tools
drwxrwxr-x 3 genew users 18 2008-04-18 12:01 bin
drwxrwxr-x 2 genew users 90 2008-04-18 11:27 releaseNotesData
-rw-rw-r-- 1 genew users 71115 2008-04-18 11:27 ReleaseNotes.html
drwxrwxr-x 2 genew users 4096 2008-04-18 11:27 doc
-rw-rw-r-- 1 genew users 675 2008-04-18 11:27 Makefile
drwxr-xr-x 7 genew users 63 2008-04-18 11:24 cuda
[genew@mega-bug ~]$ cd NVIDIA_CUDA_SDK
[genew@mega-bug NVIDIA_CUDA_SDK]$ ls cuda
bin doc include lib open64
```





Directory Structure of SDK

```
[genew@mega-bug NVIDIA_CUDA_SDK]$ ls cuda  
bin doc include lib open64
```

```
[genew@mega-bug NVIDIA_CUDA_SDK]$ ls lib  
libcutil.a libcutilD.a libparamgl.a
```

```
[genew@mega-bug NVIDIA_CUDA_SDK]$ ls bin/linux  
debug emudebug emurelease release
```

```
[genew@mega-bug NVIDIA_CUDA_SDK]$ ls common  
common.mk cutil_readme.txt inc lib Makefile Makefile_paramgl obj src
```

```
[genew@mega-bug NVIDIA_CUDA_SDK]$ ls common/inc  
bank_checker.h  cutil.h          GL                stopwatch_base.h
```

```
cmd_arg_reader.h  cutil_math.h    multithreading.h  stopwatch_base.inl  
cudpp            error_checker.h  paramgl.h         stopwatch.h  
cutil_gl_error.h  exception.h     param.h           stopwatch_linux.h
```

```
[genew@mega-bug NVIDIA_CUDA_SDK]$ ls common/src  
bank_checker.cpp  cutil.cpp       param.cpp         stopwatch.cpp  
cmd_arg_reader.cpp  multithreading.cpp  paramgl.cpp     stopwatch_linux.cpp
```



SDK CUTIL MACROS and LIBRARY

```
[genew@mega-bug inc]$ pwd
/home/genew/NVIDIA_CUDA_SDK/common/inc
[genew@mega-bug inc]$ ls
bank_checker.h  cutil.h      GL          stopwatch_base.h
cmd_arg_reader.h  cutil_math.h  multithreading.h
stopwatch_base.inl
cudpp          error_checker.h  paramgl.h   stopwatch.h
cutil_gl_error.h  exception.h    param.h
stopwatch_linux.h
[genew@mega-bug inc]$
```

cutil.h CUT macros - Cuda Utilities

I use CUT macros

LOOK AT MACROS ONLINE





Template Project

Template project sets result array equal to input times the array length.

```
[genew@mega-bug template]$ pwd
/home/genew/NVIDIA_CUDA_SDK/projects/template
[genew@mega-bug template]$ ls -lt
total 20
drwxrwxr-x 4 genew users 32 2009-01-23 15:27 obj
drwxrwxr-x 2 genew users 26 2008-04-18 11:27 doc
-rw-rw-r-- 1 genew users 2115 2008-04-18 11:27 Makefile
-rw-rw-r-- 1 genew users 5313 2008-04-18 11:27 template.cu
-rw-rw-r-- 1 genew users 2695 2008-04-18 11:27 template_gold.cpp
-rw-rw-r-- 1 genew users 3104 2008-04-18 11:27 template_kernel.cu
```

.cpp – host c routine

.cu - host/gpu cuda routine





Template Project – Makefile

```
# Add source files here  
EXECUTABLE := template  
# CUDA source files (compiled with cudacc)  
CFILES := template.cu  
# CUDA dependency files  
CU_DEPS := \  
    template_kernel.cu \  
  
# C/C++ source files (compiled with gcc / c++)  
CCFILES := \  
    template_gold.cpp \  
  
# Rules and targets  
include ../../common/common.mk
```





Template Project – CUDA files

**CUDA does not allow separate compilation of GPU (non host code).
template.cu includes template_kernel.cu as an include file.**

template.cu:





Template_gw - template with more timing

Processing time total: 90.523003 (ms)
Without alloc : 0.179000 (ms)
copy to gpu : 0.047000 (ms)
copy from gpu : 0.038000 (ms)
kernel : 0.084000 (ms)
Test PASSED

Press ENTER to exit...

Data length is 32
This kernel does almost nothing.
These times are almost all overhead.

200 micro seconds = 40 million operations at 200 gflops

Do not perform small calculations on GPU





CUDA Examples

/home/genew/NVIDIA_CUDA_SDK/projects:

alignedTypes	fluidsGL	reduction
asyncAPI	histogram256	rfAllToAll
bandwidthTest	histogram64	scalarProd
binomialOptions	imageDenoising	scan
bitonic	lineOfSight	scanLargeArray
BlackScholes	Mandelbrot	simpleAtomics
boxFilter	marchingCubes	simpleCUBLAS
clock	matrixMul	simpleCUFFT
convolutionFFT2D	matrixMulDrv	simpleGL
convolutionSeparable	MersenneTwister	simpleStreams
convolutionTexture	MonteCarlo	simpleTemplates
cppIntegration	MonteCarloMultiGPU	simpleTexture
deviceQuery	multiGPU	simpleTextureDrv
dwtHaar1D	nbody	SobelFilter
dxtc	oceanFFT	template
eigenvalues	particles	tglobal
fastWalshTransform	postProCL	transpose



Structure of NVIDIA Example Projects

```
[genew@mega-bug matrixMul]$ ls -lt
/home/genew/NVIDIA_CUDA_SDK/projects/matrixMul
total 28
2054 2008-04-18 11:27 Makefile
2008-04-18 11:27      matrixMul.cu      - host program - calls GPU kernel
2962 2008-04-18 11:27 matrixMul_gold.cpp - host solution to compare results
2272 2008-04-18 11:27 matrixMul.h        - include file; problem size, etc
4791 2008-04-18 11:27 matrixMul_kernel.cu - GPU code
```

matrixMul.cu “includes” matrixMul_kernel.cu





Compiling an Example

```
[genew@mega-bug bandwidthTest]$ pwd
/home/genew/NVIDIA_CUDA_SDK/projects/bandwidthTest
[genew@mega-bug bandwidthTest]$ verbose=1 make clean
rm -f obj/release/bandwidthTest.cu_o
rm -f
rm -f ../../bin/linux/release/bandwidthTest
rm -f
[genew@mega-bug bandwidthTest]$ verbose=1 make
nvcc -o obj/release/bandwidthTest.cu_o -c bandwidthTest.cu --compiler-options -fno-strict-aliasing -I. -I/usr/local/cuda/include -I../../common/inc -DUNIX -O3
g++ -fPIC -o ../../bin/linux/release/bandwidthTest obj/release/bandwidthTest.cu_o -
L/usr/local/cuda/lib -L../../lib -L../../common/lib -lcudart -IGL -IGLU -lglut -
L/usr/local/cuda/lib -L../../lib -L../../common/lib -lcutil
[genew@mega-bug bandwidthTest]$
```

Compiling an Example

`dbg=1 make` compiles debug version
`emu=1 make` compiles emulation version
`verbose=1 make` make runs in verbose mode

default is release

CUT macros do not check for failure in
release mode

for testing initialize device memory !!!



Running an Example

```
[genew@mega-bug bandwidthTest]$ ../../bin/linux/release/bandwidthTest
```

Quick Mode

Host to Device Bandwidth for Pageable memory

.

Transfer Size (Bytes) Bandwidth(MB/s)

33554432 568.2

Quick Mode

Device to Host Bandwidth for Pageable memory

.

Transfer Size (Bytes) Bandwidth(MB/s)

33554432 527.6

Quick Mode

Device to Device Bandwidth

.

Transfer Size (Bytes) Bandwidth(MB/s)

33554432 57291.2 = 57 gbytes/sec = 15 gwords/sec

&&&& Test PASSED

Press ENTER to exit...



Steps in CUDA code

Host Code

Initialize/acquire device (GPU)

Allocate memory on GPU

Copy data from host to GPU

Execute kernel on GPU

Copy data from GPU to host

Deallocate memory on GPU

Release device

Run “GOLD” version on host

Compare results



Steps in CUDA code Initialize/acquire Device

```
void  
runTest( int argc, char** argv)  
{  
  
    CUT_DEVICE_INIT();  
    ....  
}
```



Allocate memory

```
// allocate host memory
float* h_idata = (float*) malloc( mem_size);
// initialize the memory
for( unsigned int i = 0; i < num_threads; ++i)
{
    h_idata[i] = (float) i;
}

// allocate device memory
float* d_idata;
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_idata, mem_size));
// copy host memory to device
```

Copy Data from host to GPU

// copy host memory to device

```
CUDA_SAFE_CALL( cudaMemcpy( d_idata, h_idata, mem_size,  
                           cudaMemcpyHostToDevice) );
```



Execute Kernel on GPU

```
// setup execution parameters
dim3  grid( 1, 1, 1);
dim3  threads( num_threads, 1, 1);

// execute the kernel
testKernel<<< grid, threads, mem_size >>>(
d_idata, d_odata);

// check if kernel execution generated and error
CUT_CHECK_ERROR("Kernel execution failed");
```

Copy Data from GPU to Host

```
// copy result from device to host  
CUDA_SAFE_CALL( cudaMemcpy( h_odata, d_odata,  
sizeof( float) * num_threads,  
cudaMemcpyDeviceToHost) );
```



Deallocate Memory on GPU

```
// cleanup memory  
free( h_idata);  
free( h_odata);  
free( reference);  
CUDA_SAFE_CALL(cudaFree(d_idata));  
CUDA_SAFE_CALL(cudaFree(d_odata));
```



Timing Code

```
unsigned int timer = 0;  
CUT_SAFE_CALL( cutCreateTimer( &timer));  
CUT_SAFE_CALL( cutStartTimer( timer));  
...  
CUT_SAFE_CALL( cutStopTimer( timer));  
printf( "Processing time: %f (ms)\n",  
        cutGetTimerValue( timer));  
CUT_SAFE_CALL( cutDeleteTimer( timer));
```

Kernel Code

```
__global__ void testKernel( float* g_idata, float* g_odata)
{
    // shared memory
    // the size is determined by the host application
    extern __shared__ float sdata[];

    // access thread id
    const unsigned int tid = threadIdx.x;
    // access number of threads in this block
    const unsigned int num_threads = blockDim.x;

    // read in input data from global memory
    // use the bank checker macro to check for bank conflicts during host
    // emulation
    SDATA(tid) = g_idata[tid];
    __syncthreads();

    // perform some computations
    SDATA(tid) = (float) num_threads * SDATA( tid);
    __syncthreads();

    // write data to global memory
    g_odata[tid] = SDATA(tid);
}
```

Kernel Code Implied Thread Loop

```
// access thread id
const unsigned int tid = threadIdx.x;
// access number of threads in this block
const unsigned int num_threads = blockDim.x;

// read in input data from global memory
// use the bank checker macro to check for bank conflicts
// during host
// emulation
SDATA(tid) = g_idata[tid];
__syncthreads();

// perform some computations
SDATA(tid) = (float) num_threads * SDATA( tid);
__syncthreads();
```



The Important Features For Efficiency

Threads/Blocks

Shared Memory

Few conditionals

NVIDIA libraries

Stay on GPU





Setting Number of Threads and Blocks

```
dim3  grid( num_blocks,1, 1);  
dim3  threads( num_threads, 1, 1);  
  
// total number of threads = num_threads*num_blocks  
  
// execute the kernel  
testKernel<<< grid, threads >>>( d_idata, d_odata);
```



How Many Threads

How many Blocks

- Each block assigned to a multiprocessor
- Each block executes independently of all other blocks
- Need at least 32 threads per block for cpu efficiency (keep 8 processors busy)
- Maximum 512 threads per block
- At least 16 blocks so each multiprocessor is busy
- Problem 'Vector Length' *****
- Number of blocks = $VL / \text{threads_per_block}$
- Ignoring shared memory
 - Threads = 512 (maximum)
 - Blocks = $VL / \text{Threads}$



Convert Template to New Project

```
cp -r template newtest
```

```
change 'template' to 'newtest' in Makefile
```

```
mv template.cu newtest.cu
```

```
mv template_kernel.cu newtest_kernel.cu
```

```
mv template_gold.cpp newtest_gold.cpp
```

```
In template.cu
```

```
include template_kernel.cu => include newtest_kernel.cu
```

```
In template_kernel.cu
```

```
change TEMPLATE_KERNEL_H => NEWTEST_KERNEL_H
```

```
make clean
```

```
make
```

```
../../bin/linux/release/newtest
```



cpu_10 – CPU only

Time Blocks and Threads

cpu_10/cpu_10.cu

```
unsigned int length = 4096*4096/8;  
int threads_per_blocka[10] = {1,2,4,8,16,32,64,128,256,512};  
int threadsa[25] = {1,2,4,8,16,32,64,128,256,512,1024,2048,  
4096,8192,16384,32768,65536,131072,  
262144,524288,1048576,2097152,4194304,  
8388608,16777216};  
int ithreads_per_block,ithreads;  
int nthreads,nthreads_per_block,blocks_per_grid;
```

Threads and Blocks

- Each block contains a group of threads ≤ 512
- The grid contains a group of blocks ≤ 32768
- Each block executes on a single multiprocessor.
- Each block executes independently.
- Different blocks can not communicate or synchronize with each other.
- Need at least 64 threads per block for efficiency. Maybe more.

cpu_10

Time Blocks and Threads

cpu_10/cpu_10.cu

```
for(ithreads=4 ; ithreads<25 ; ithreads++){
    nthreads=threadsa[ithreads];
    for(ithreads_per_block=0 ; ithreads_per_block<10 ;ithreads_per_block++){
        nthreads_per_block = threads_per_blocka[ithreads_per_block];
        times[ithreads][ithreads_per_block]=0.0;
        blocks_per_grid = nthreads/nthreads_per_block;
        printf("\nthreads=%d nthreads_per_block=%d blocks_per_grid=%d\n",
            nthreads, nthreads_per_block,blocks_per_grid);
        if(blocks_per_grid>32768){
            printf("blocks_per_grid=%d > 32768. skipping\n",blocks_per_grid);
            continue;
        }
        if(nthreads_per_block>nthreads){
            printf("nthreads_per_block=%d > nthreads=%d skipping\n",
                nthreads_per_block,nthreads);
            continue;
        }
    }
}
```

cpu_10

Time Blocks and Threads

```
cpu_10/cpu_10.cu
```

```
// setup execution parameters
    dim3  grid( blocks_per_grid, 1, 1);
    dim3  threads( nthreads_per_block, 1, 1);

    CUT_SAFE_CALL( cutStartTimer( timer));
// execute the kernel
    testKernel<<< grid, threads >>>(length,d_odata);

// check if kernel execution generated and error
    CUT_CHECK_ERROR("Kernel execution failed");
```

cpu_10 Kernel Code

```
cpu_10/cpu_10_kernel.cu
__global__ void testKernel(int length, float* g_odata)
{
    // access thread id
    const unsigned int tid = threadIdx.x + blockIdx.x*blockDim.x;
    const unsigned int num_threads = blockDim.x * gridDim.x ;
    // perform some computations
    unsigned int i,ix;

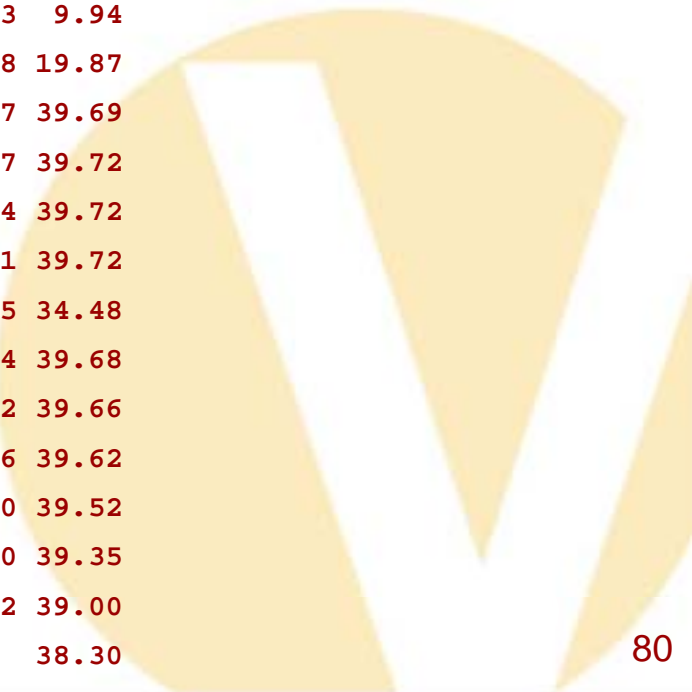
    for(ix=0 ; ix<length ; ix += num_threads){
        i=ix+tid;
        if(i<length){
            int j;
            float x;
            x=i;
            for(j=0;j<250;j++){
                x=x*3.0+9.0;
                x=x*.3333333-3.0;
            }
            g_odata[i] = x;
        }
    }
}
```





Threads/Blocks Timings Not Perfectly Predictable

threads	threads per block									
	1	2	4	8	16	32	64	128	256	512
16	0.27	0.27	0.27	0.27	0.27					
32	0.54	0.54	0.54	0.54	0.54	0.54				
64	1.07	1.07	1.07	1.07	1.07	1.07	1.07			
128	1.76	2.15	2.15	2.15	2.15	2.15	2.15	1.95		
256	1.27	3.53	4.29	4.29	4.29	4.29	4.29	3.90	2.46	
512	1.48	3.53	7.05	8.59	8.59	8.59	8.58	7.81	4.92	2.49
1024	1.60	3.53	7.05	14.10	17.17	17.16	17.16	9.80	9.83	4.93
2048	1.68	3.53	7.05	14.10	28.17	31.20	31.20	30.53	19.63	9.94
4096	1.73	3.53	7.06	14.11	28.18	39.15	39.18	26.47	19.88	19.87
8192	1.75	3.53	7.06	14.12	28.22	39.21	39.70	31.81	26.17	39.69
16384	1.76	3.54	7.07	14.14	28.28	39.24	39.72	26.52	31.77	39.72
32768	1.77	3.55	7.09	14.18	28.31	39.24	39.73	35.34	35.34	39.72
65536		3.56	7.12	14.23	28.44	39.25	39.74	36.35	37.41	39.72
131072			7.12	14.25	28.45	39.27	39.74	38.55	38.55	34.48
262144				14.23	28.36	39.30	39.73	39.13	39.14	39.68
524288					28.44	39.35	39.73	39.58	39.42	39.66
1048576						39.37	39.72	39.48	39.56	39.62
2097152							39.68	39.63	39.60	39.52
4194304								39.52	39.50	39.35
8388608									39.22	39.00
16777216										38.30





How Many Threads How Many Blocks

use 16 blocks

each block has thread count = vector length/16

more blocks if thread count > 512

consider shared memory – more later

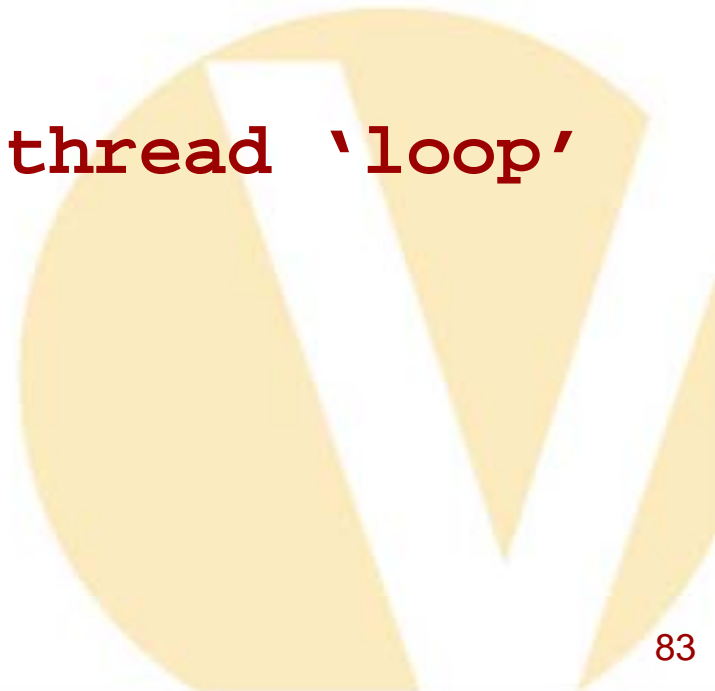
registers and local memory per multiprocessor are limited. May limit the threads per block to less than 512

- **Use of shared memory**
- **Use shared memory when:**
 - **same elements are used repeatedly**
 - **access pattern is not regular**
- **ISI project devicemem**
- **If not memory intensive, don't have to use shared memory**
- **If memory intensive, can get some speed without shared memory**
- **limited to 4088 words per block**

Devicemem Kernel Code

(See editor screen)

- `copy index array (2048) from global`
- `to shared memory for each block`
- `syncthreads`
- `use shared index array`
- `two different forms of thread 'loop'`
 - `Block only loop`
 - `Multiblock loop`



Shared Memory Timing

	global	shared
0%	1479	596
25%	2554	1597
50%	2897	2068
75%	3632	2715
100%	4308	3437



CUDA Libraries

Very efficient CUDA libraries are supplied

Linear algebra and FFT

User can call these routines from kernel without writing any parallel code

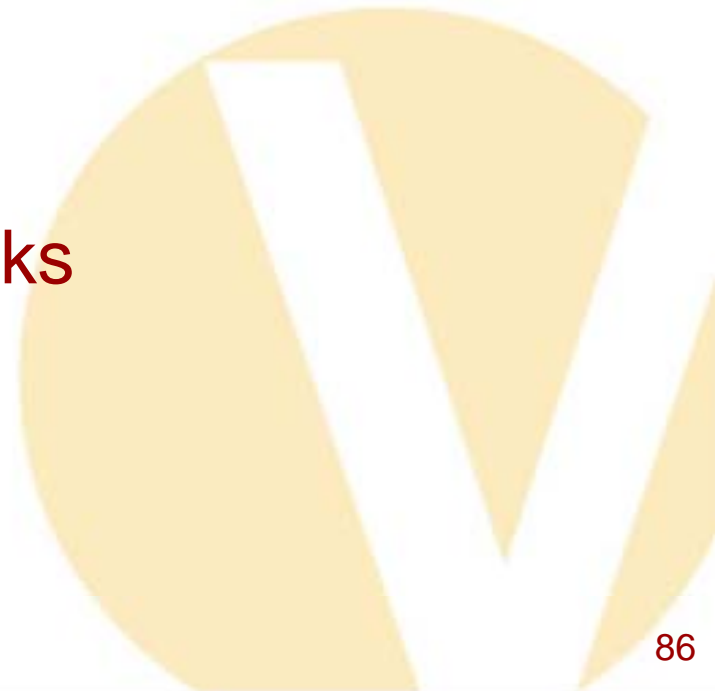
Still write allocates, transfers etc



That's It !!!

- threads to keep processors busy
- shared memory used efficiently
- library routines
- no synchronization across blocks

You have the mechanics.
You have the general rules



Some Hints

- Use chunks big enough to avoid start/stop overhead
- Enough calculations to cover copy in/out overhead
- Need problem with vector length at least $32 \times 16 = 512$. Bigger better
- Use library routines when possible
- Stay on the GPU. Do intermediate steps on the GPU even if inefficient, but not too inefficient

CUT_DEVICE_INIT

```
/home/genew/NVIDIA_CUDA_SDK/common/inc/cutil.h
# define CUT_DEVICE_INIT() do {
    int deviceCount;
    CUDA_SAFE_CALL_NO_SYNC(cudaGetDeviceCount(&deviceCount));
    if (deviceCount == 0) {
        fprintf(stderr, "There is no device.\n");
        exit(EXIT_FAILURE);
    }
    int dev;
    for (dev = 0; dev < deviceCount; ++dev) {
        cudaDeviceProp deviceProp;
        CUDA_SAFE_CALL_NO_SYNC(cudaGetDeviceProperties(&deviceProp, dev));
        if (deviceProp.major >= 1)
            break;
    }
    if (dev == deviceCount) {
        fprintf(stderr, "There is no device supporting CUDA.\n");
        exit(EXIT_FAILURE);
    }
    else
        CUDA_SAFE_CALL(cudaSetDevice(dev));
} while (0)
```

```
/home/genew/NVIDIA_CUDA_SDK/common/inc/cutil.h
```

```
#ifdef _DEBUG
```

```
...
```

```
# define CUDA_SAFE_CALL_NO_SYNC( call) do {                                \
    cudaError err = call;                                                  \
    if( cudaSuccess != err) {                                              \
        fprintf(stderr, "Cuda error in file '%s' in line %i : %s.\n",      \
            __FILE__, __LINE__, cudaGetErrorString( err) );              \
        exit(EXIT_FAILURE);                                               \
    } } while (0)
```

```
...
```

```
#else
```

```
...
```

```
# define CUDA_SAFE_CALL_NO_SYNC( call) call
```

```
...
```

```
#endif
```



Memory – Matrix Transpose

On CPU (or GPU)

Too memory intensive for GPU unless intermediate step

Transpose a matrix – flip it across a diagonal

```
for(j=0;j<n;j++){  
    for(i=0;i<n;i++){  
        AT[i][j] = A[j][i];  
    }  
}
```

memory access is good for A, bad for AT

if we switch the nesting of the loops, memory access is good for AT, bad for A.

Memory – Matrix Transpose A Solution

Consider matrix broken up into blocks:

01	02	03	04
05	06	07	08
09	10	11	12
13	14	15	16

transpose blocks:

01	05	09	13
02	06	10	14
03	07	11	15
04	08	12	16

copy each block to new location while transposing

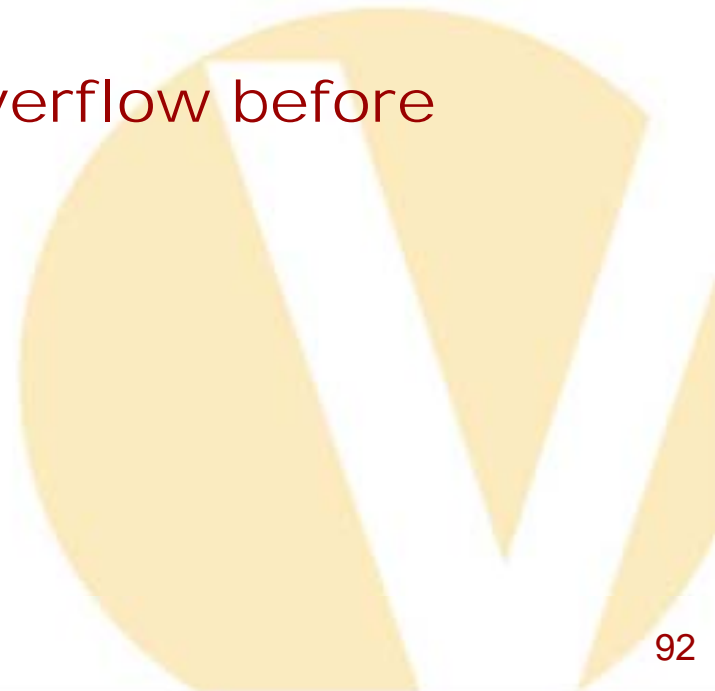


Memory – Matrix Transpose A Solution

Efficient cache use does not require contiguous sequential access

The blocked transpose results in the use of all data transferred to cache.

If blocks are too big, cache will overflow before reuse.





Write Your Own Parallel Code

User can write their own CUDA code

More difficult

Not too difficult to write efficient code

Simple code in matrix multiply example is half as fast as the matrix multiply library