CMPE 214
GPU Architecture & Programming

Lecture 2.
GPU Memory Model (3)

Haonan Wang

SAN JOSÉ STATE
UNIVERSITY

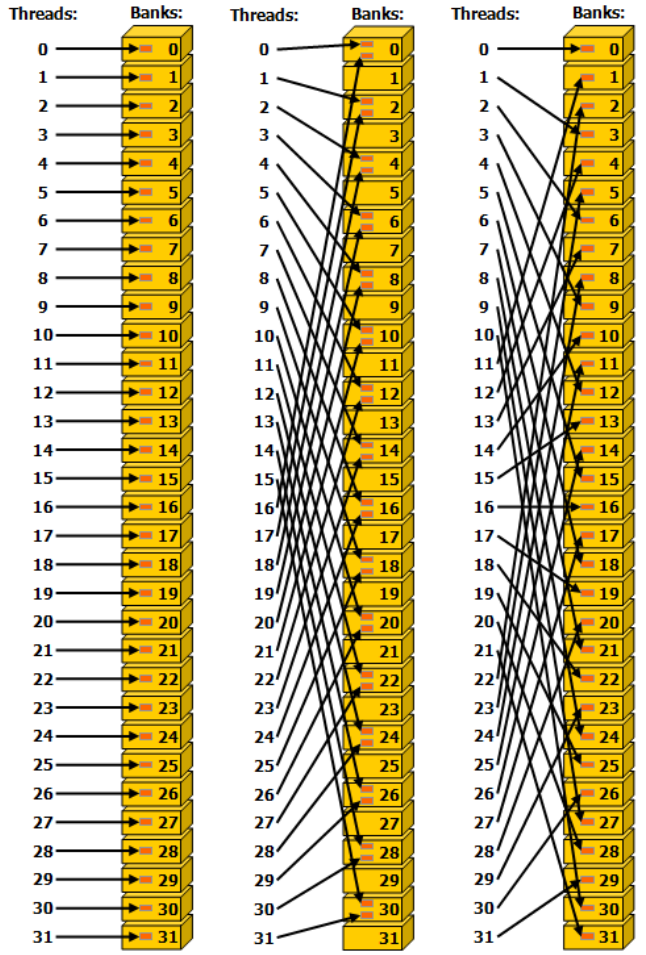
Considerations for Optimization

- **Shared memory**
  - Thread arrangement
  - Bank conflict
- **Context switch overhead**
- **Data movement reduction**
  - Coalescing unit
  - Cache: Miss status holding register
- **SM: Two levels of parallelism**
  - Block scheduler
  - Warp scheduler
- **Memory**
  - Multiple levels of parallelism
  - Row buffer locality

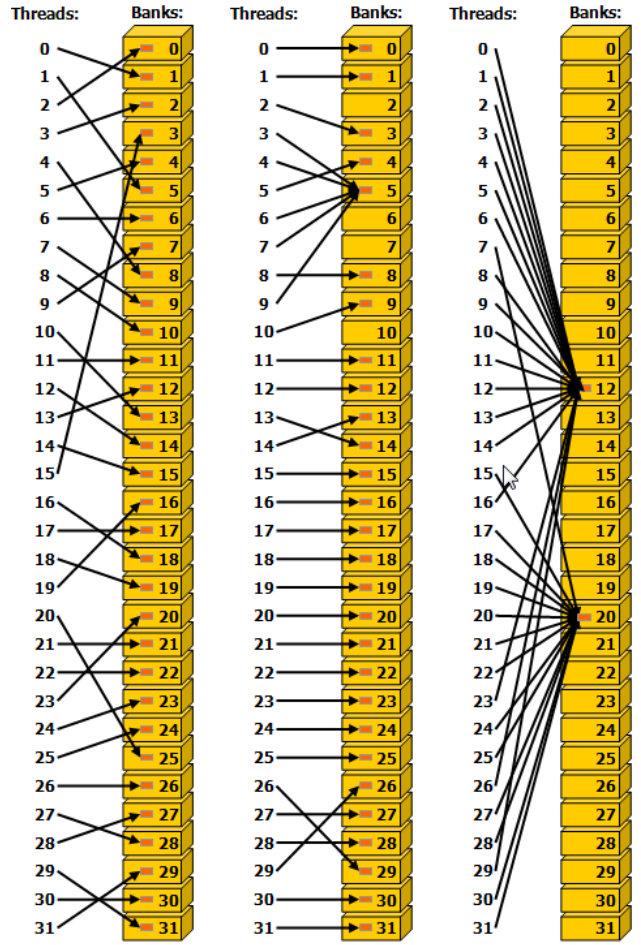
Shared Memory: Bank Conflicts

- **Shared memory consists of multiple banks**
  - Mapping unit = 4B
  - Only one access can be made per bank per cycle
- **Bank conflict: threads in a warp access <u>different word</u> in the same bank**
  - Access are serialized
- **For the same word in the same bank:**
  - Read: broadcast
  - Write: serialized

Bank conflict example

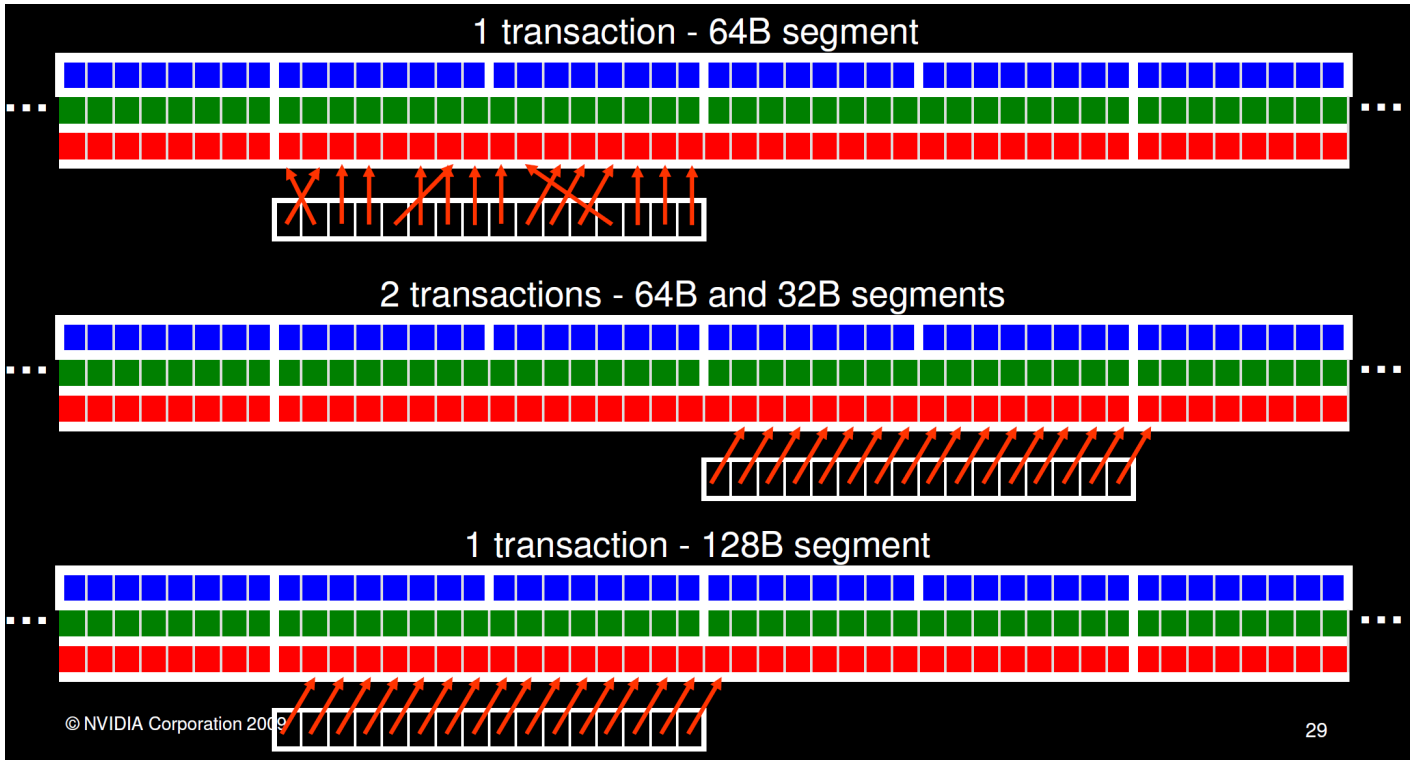


Broadcast example



Memory Coalescing

- **Global memory accesses are coalesced in 32, 64, and 128 bytes**
  - Coalesced accesses are serviced by one memory transaction
  - Changed at CC 1.2, CC 2.0, CC 6.0?

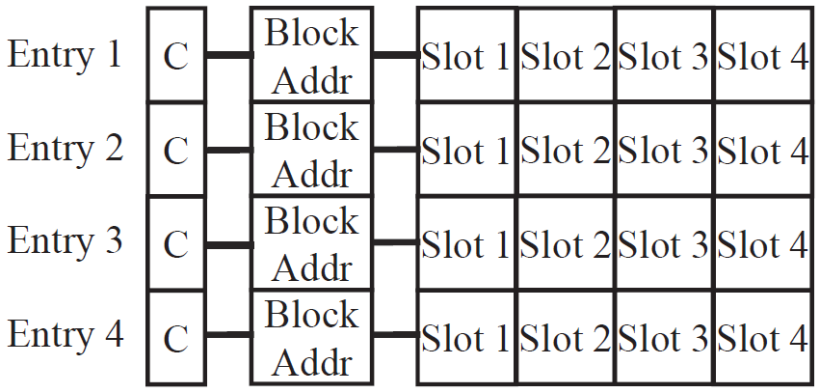


Caches & MSHR

- **L1 cache is disabled by default**
  - Enable: nvcc -Xptxas -dlcm=ca main.cu
  - Disbable: nvcc -Xptxas -dlcm=cg main.cu
  - L2 cache is always enabled

- **L1 and shared memory size limit:**
  - cudaDeviceSetCacheConfig(), limited at 48KB before CC 7.0
  - Use dynamic allocation to use beyond 48KB after CC 7.0
    - MatrixMul<<<1024, 1024, N * sizeof(flaot)>>>(...);
    - cudaFuncSetAttribute(my_kernel, cudaFuncAttributeMaxDynamicSharedMemorySize, 98304);

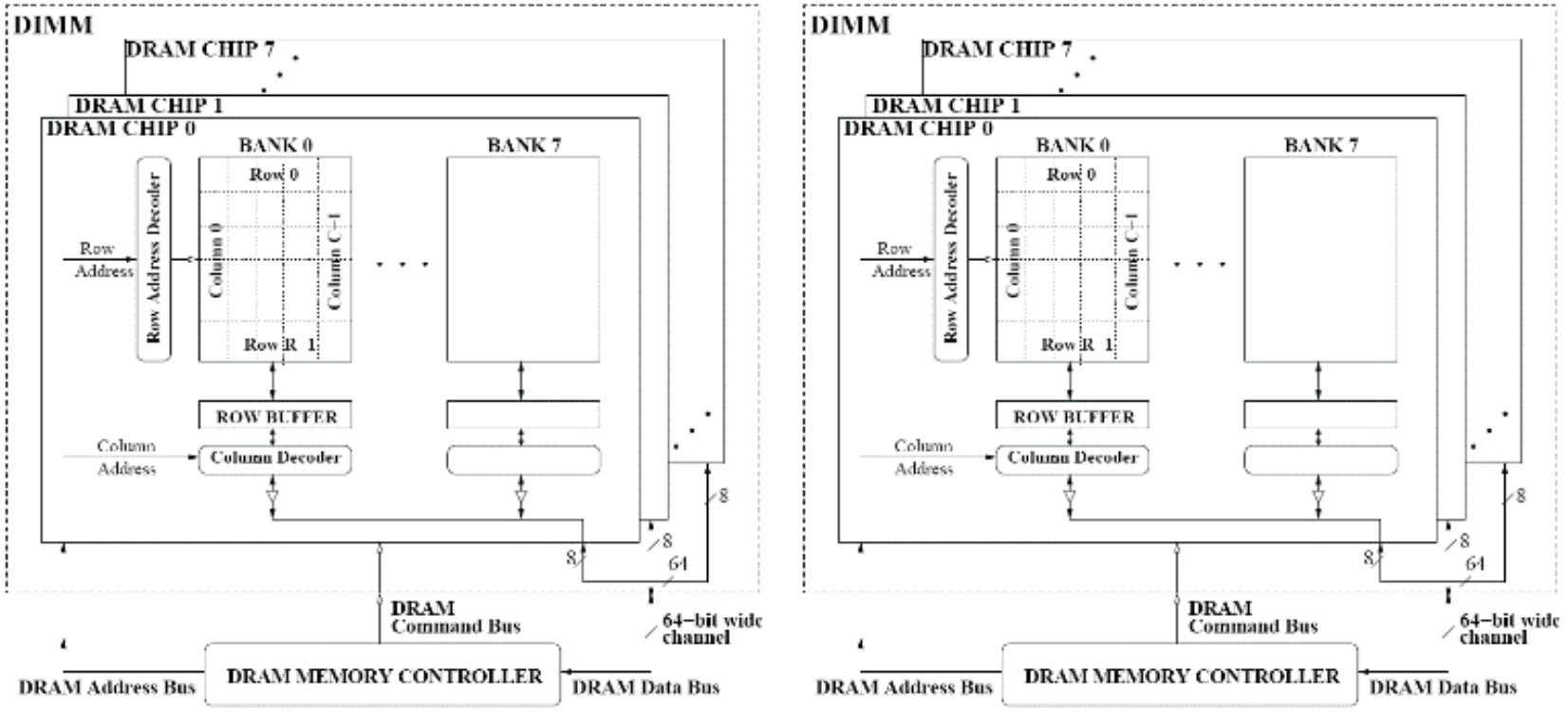
- **MSHR: Miss status holding registers**
  - Coalescing at the cache level
  - Reservation fail:
    - Limited slots for each address



Memory: Levels of Parallelism

DRAM Organization:

- Channel
- DIMM
- Rank
- Chip
- Bank
- Row/Column



- **Row size = 4KB**
- **Channel level parallelism & Bank level parallelism:**
  - Map consecutive row requests into different channel and banks

Peformance Analysis

- Run time estimation
- Data reuse analysis
- Real experiment
- Profiling
- Simulation

Data Reuse Analysis (1)

How to maximize reuse?

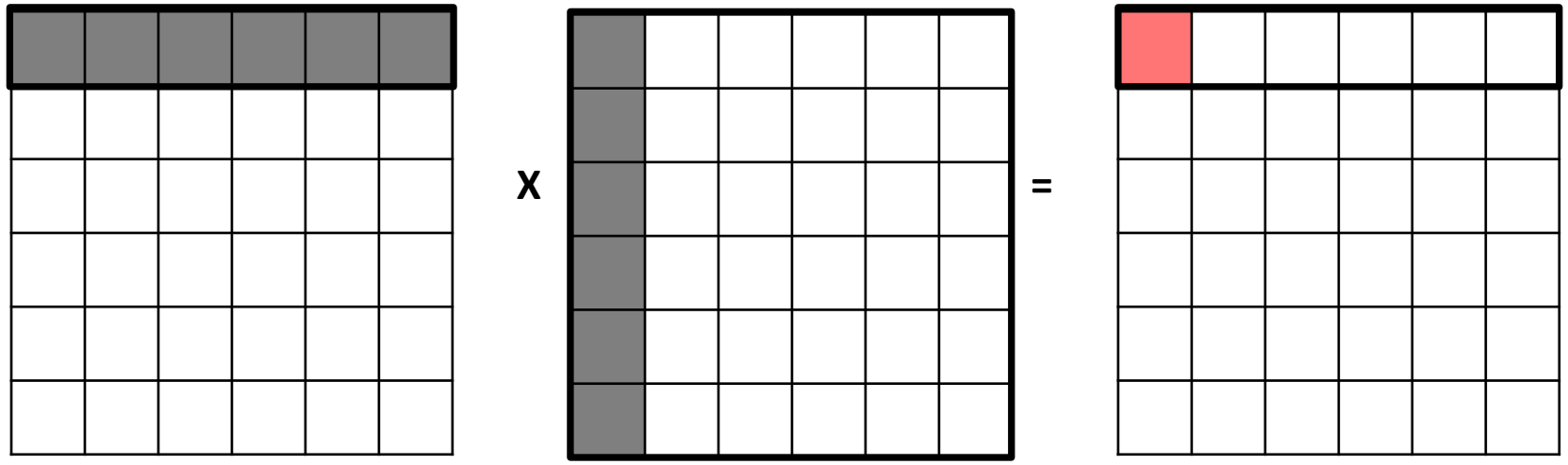
- Assume shared memory capacity = 8

Reuse per load:

M = 6 / 1 = 6

N = 6 / 6 = 1

AVG = 12 / 7



Data Reuse Analysis (2)

How to maximize reuse?

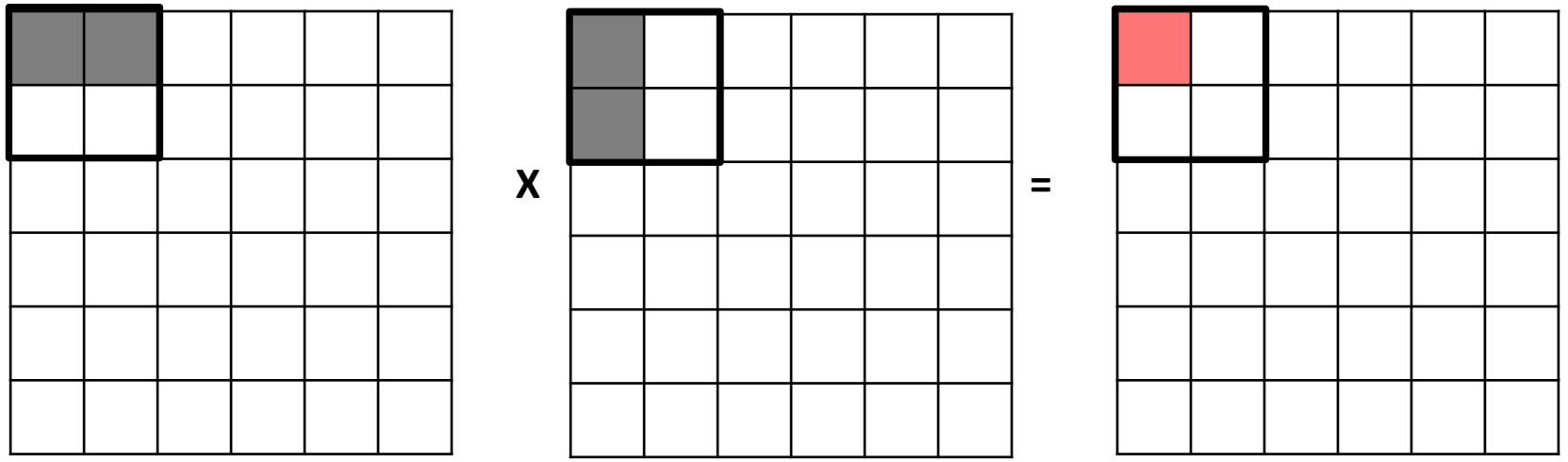
- Assume shared memory capacity = 8

Reuse per load:

M = 2 / 1 = 2

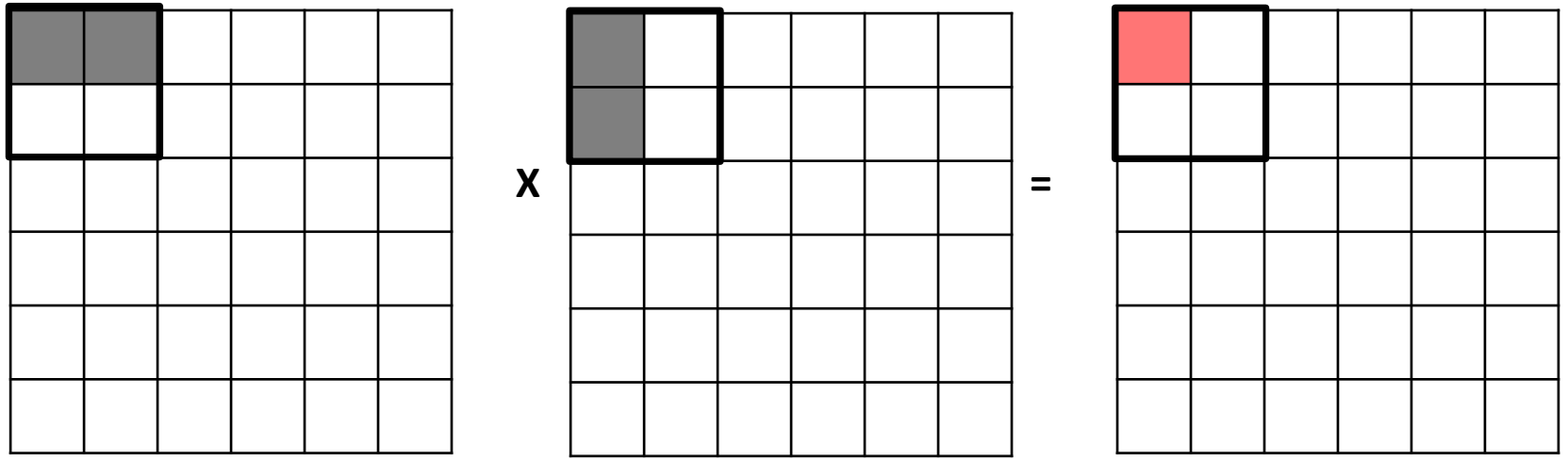
N = 2 / 1 = 2

AVG = 4 / 2 = 2



Data Reuse Analysis (3)

- Assume shared memory capacity = 18
  - Matrix width = 6, tile width = 3
- Assume shared memory capacity = 32
  - Matrix width = 12, tile width = 4



Context Switches (1)

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {

  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;

    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
     Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }

    P[Row*Width+Col] = Pvalue;
  }

}
```

Context Switches (2)

```
__global__ void MatrixMul(float* A, float* B, float* C) {
…
 int tile_row = threadIdx.x/TILE_WIDTH;
 int tile_col = threadIdx.x%TILE_WIDTH;
 __shared__ float As[TILE_WIDTH * TILE_WIDTH];
 __shared__ float Bs[TILE_WIDTH * TILE_WIDTH];
 float Cl = 0;

 for (int p = 0; p < WIDTH / TILE_WIDTH; ++p) {
    As[threadIdx.x] = A[row * WIDTH + p * TILE_WIDTH + tile_col];
    Bs[threadIdx.x] = B[(p * TILE_WIDTH + tile_row)* Width + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)
    Cl += As[tile_row * WIDTH + i] * Bs[i * WIDTH + tile_col];

    __synchthreads();
  }

  C[tid] = Cl;
}
```

```
int tid= blockIdx.x * blockDim.x + threadIdx.x;
int row = tid/WIDTH;
int col = tid%WIDTH;
```

Block and Warp Schedulers

- **Stalls: warp is context switched out to wait for long running operations**
  - Which warp will run next dependents on the warp Scheduler
  - Context switch overhead

- **Block scheduler: Naïve approach**

- **Warp schedulers:**
  - LRR: Lose round robin
  - GTO: Greedy then oldest
  - SWL: static warp limit
  - Two-level

Profiling (1)

- CPU timer

```
t1 = myCPUTimer();
vecadd<<<256, 256>>>(…);
cudaDeviceSynchronize();
t2 = myCPUTimer();
Total = t2 – t1;
```

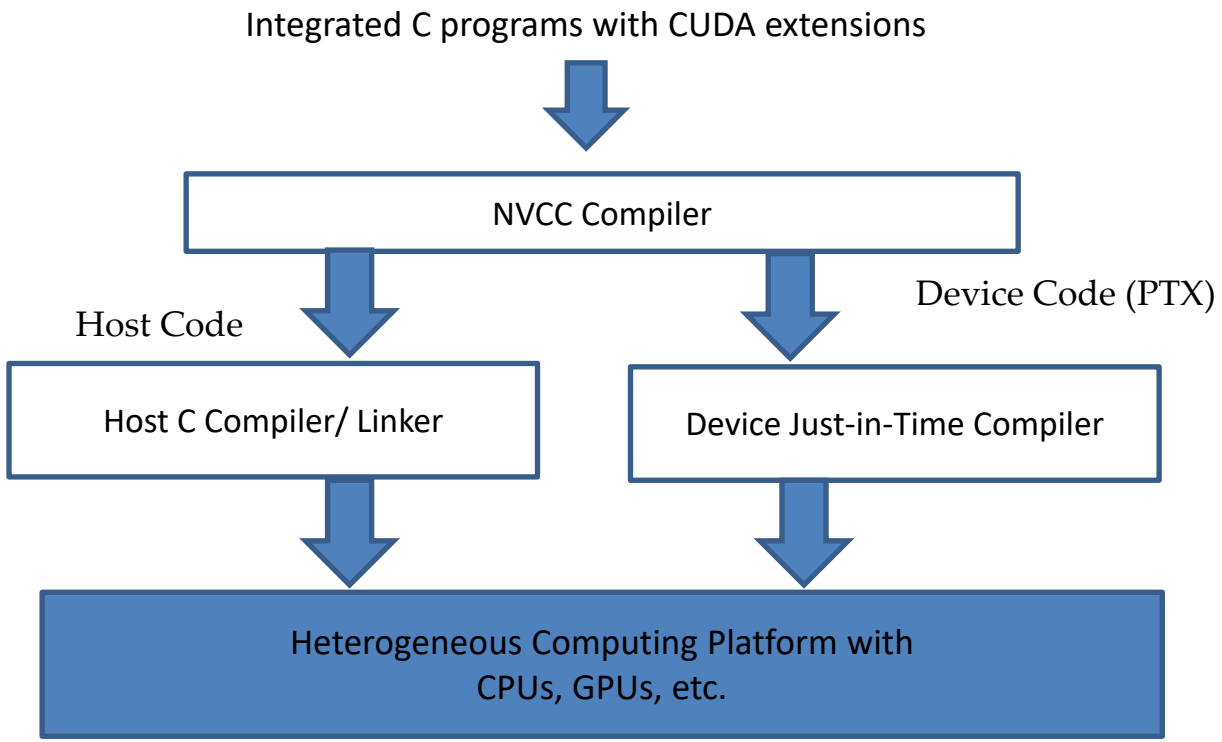
- CUDA Event

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
vecadd<<<256, 256>>>(…);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

Profiling (2)

- NVPROF
  - nvprof --print-gpu-trace ./vecadd
  - nvprof --metrics ipc ./vecadd
- Timer in the kernel
  - clock_t clock();
  - long long int clock64();

CUDA Tools


Integrated C programs with CUDA extensions
NVCC Compiler
Host Code
Device Code (PTX)
Host C Compiler/ Linker
Device Just-in-Time Compiler
Heterogeneous Computing Platform with
CPUs, GPUs, etc.

CUDA Tools

- Compiler: NVCC
- Debugging tool: cuda-gdb, cuda-memcheck
- Profiling tool: NVPROF, NVVP
- IDE: NSIGHT
- Libs

SAN JOSÉ STATE UNIVERSITY *powering* SILICON VALLEY