# CUDA and Fermi Optimization Techniques

Hyungon Ryu, PSG SA

NVIDIA®

# Agenda

- CUDA 101 (General Overview)
- Toy of FDM example
- Monte Carlo Simulation & Time Series Analysis
- General CUDA Optimization Tips

# CUDA 101

## General Overview

NVIDIA

# CUDA

**Graphics (GPU)**

**Parallel Programming**

# CUDA Parallel programming

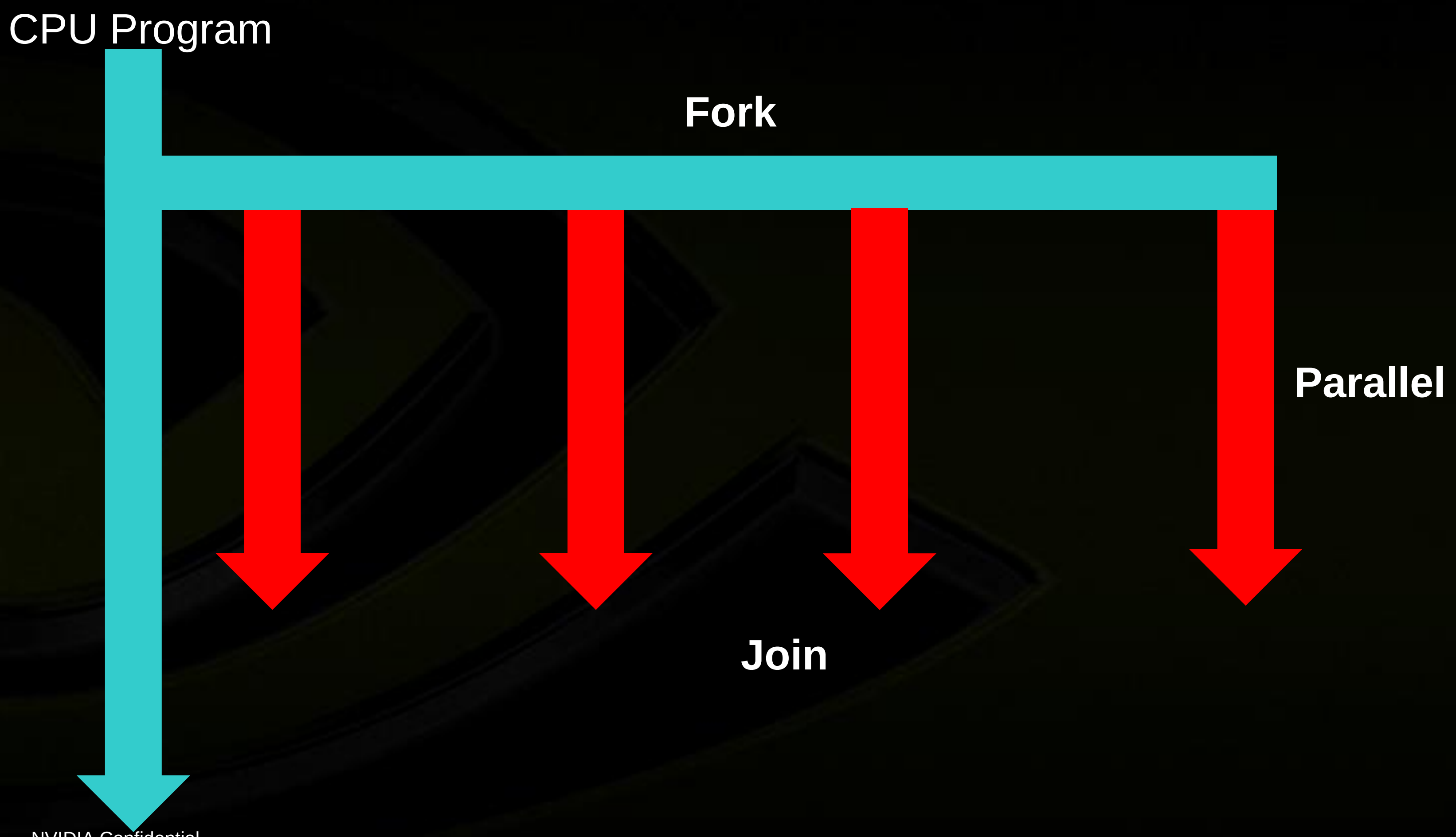
- GPU Knowledge
  - Cg
  - OpenGL
  - DirectX
- Parallel knowledge
  - Pthreads / winthreads
  - MMX, SSE
  - OpenMP
  - PVM / MPI

- Hetereogeneous Knowledge
  - GPU
  - Parallel DSP
  - Parallel ASIC
  - Parallel FPGA
  - Cell BE

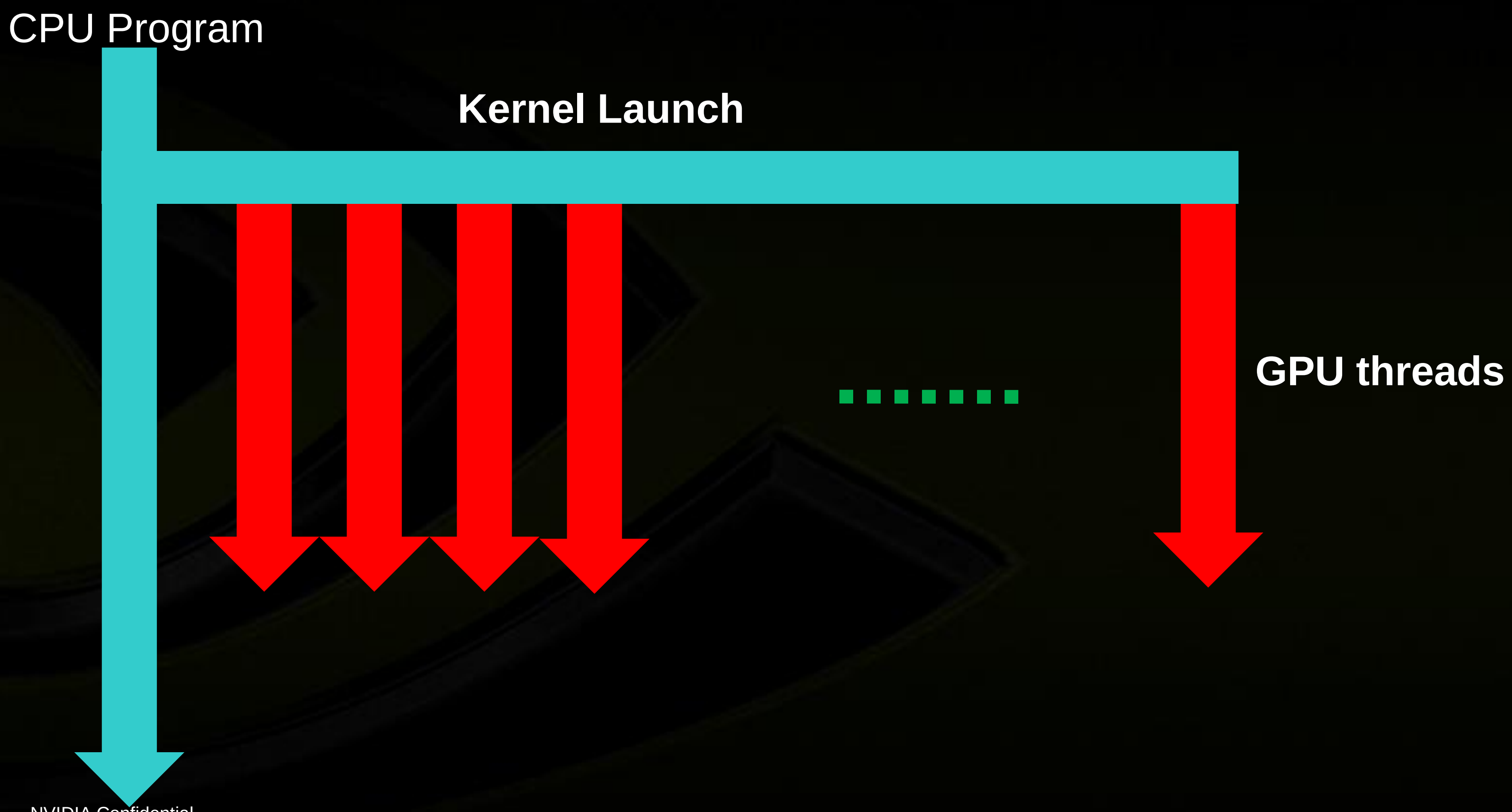
**CUDA**

Parallel Computing with GPU

# Parallel Model in OpenMP

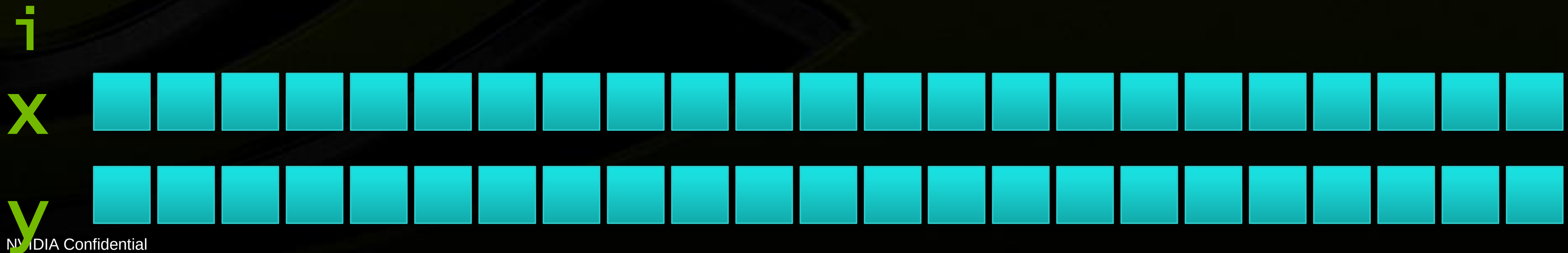


# CUDA parallel Model



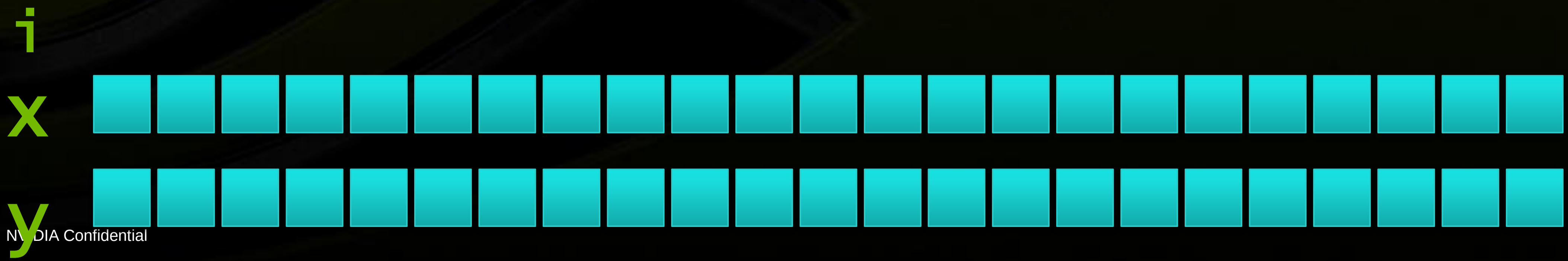
# Saxpy Example : CPU serial

```
for (int i = 0; i < n; ++i) {
        y[i] = a*x[i] + y[i];
}
```



# Example of Saxpy Parallel : OpenMP

```
# pragma omp parallel shared (n,a,x,y) private (i)
# pragma omp for
for (int i = 0; i < n; ++i) {
      y[i] = a*x[i] + y[i];
}
```



# Example of Saxpy Parallel : MPI

```
for ( i = start ; i < end; i++)
{
y[i] = a * x[i] + y[i];
}
```

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);

void para_range(int lowest, int highest,int nprocs, int myrank,
                int *start, int *end) {
    int wk1, wk2;
    wk1 = (highest - lowest + 1) / nprocs;
    wk2 = (highest - lowest + 1) % nprocs;
    *start = myrank * wk1 + lowest + ( (rank<wk2) ? myrank : wk2);
    *end = *start + wk1 - 1;
    if(wk2 > rank) *end = *end + 1;
}
```

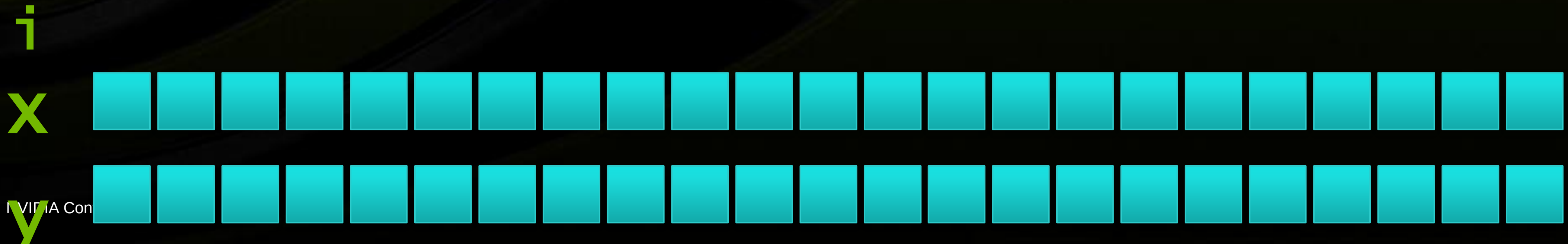
i

x

y

# Example of Saxpy Parallel : SSE

```
void saxpy_vector(short *z, short *x, short *y, short a, unsigned n) {
   __m128i* x_ptr = (__m128i*) x;
   __m128i* y_ptr = (__m128i*) y;
   __m128i* z_ptr = (__m128i*) z;
   __m128i a_vec = _mm_splat_epi16(a);
   int i;
   for (i = 0; i < n/8; ++i) {
     __m128i x_vec = x_ptr[i];
     __m128i y_vec = y_ptr[i];
     __m128i z_vec = _mm_add_epi16( _mm_mullo_epi16 x_vec,a_vec),y_vec);
     z_ptr[i] = z_vec;
   }
 }
```



# Saxpy Parallel : CUDA

```
{
x[i] = a * x[i] + t * y[i];
}
```

```
Saxpy <<<N ,M >>> (n, 2.0, x, y);
```

# CUDA C extension

**Launch the kernel**

Function <<< Grid, Block >>> ( parameter);

**Additional C standard API for mem control**

cudaXXX : cudaMalloc, cudaMemcpy,

cuXXX : cuMalloc, cuMemcpy

cutXXX : cutXXX

**For Function**

__global__, __device__, __host__, __device__ __host__

**For memory**

__shared__, __device__, reg/loc

**pre-defined variables**

blockDim, blockIdx, threadIdx, cudaMemcpyHostToDevice

**Pre-defined function**

__syncthreads(), __mul24(); etc

# Process of CUDA developing

**Serial**

Algorithm

serial Programming

Compile

Debugging

Release

↔

**CUDA parallel**

Algorithm

serial Programming

Compile

CUDA convert

Profile

Parallelize

Compile

Debugging

Optimize/profile

Release

# CUDA is Parallel Computing !!!

**Serial**

Algorithm

serial Programming

Compile

Debugging

Release

**MPI parallel**

Algorithm

serial Programming

Compile

parallel Programming

Profile

Parallelize

Compile

Debugging [totalview]

Optimize

Release

MPIrun

**CUDA parallel**

Algorithm

serial Programming

Compile

CUDA convert

Profile

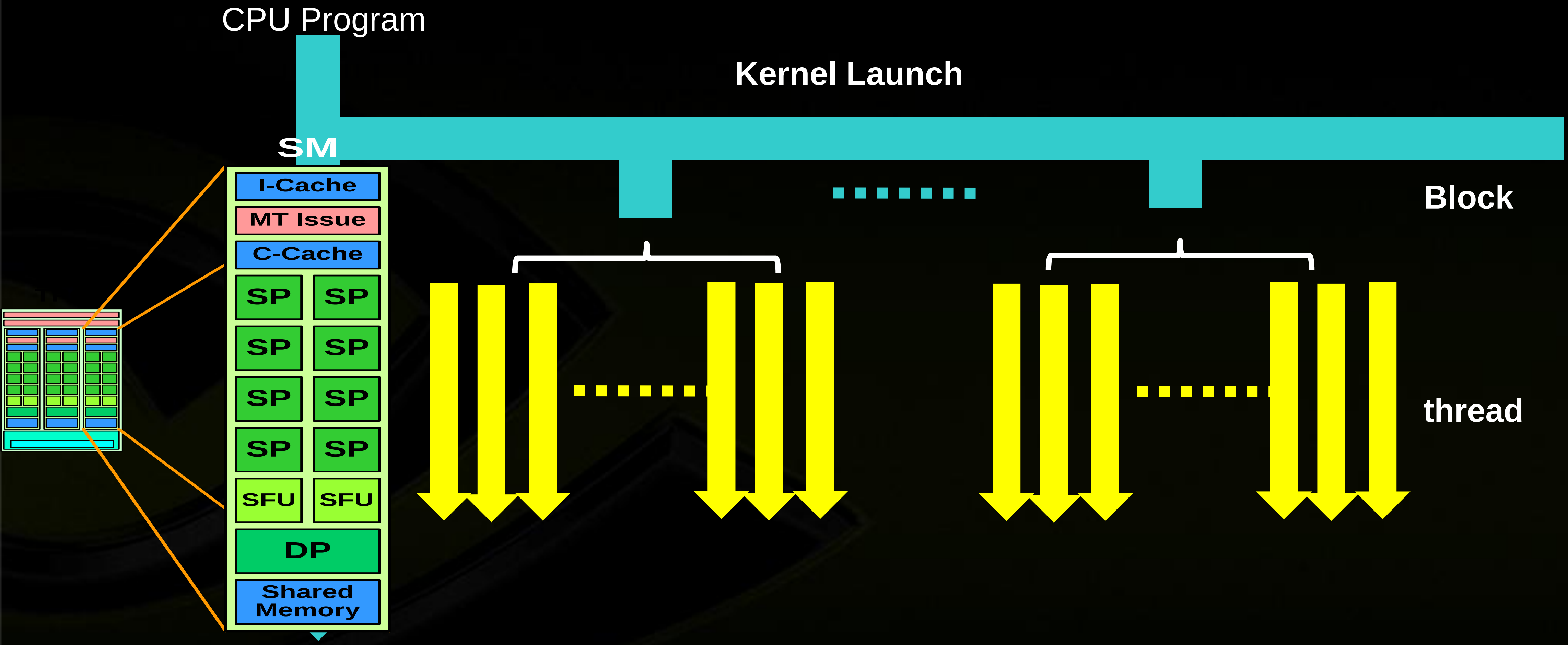
Parallelize

Compile

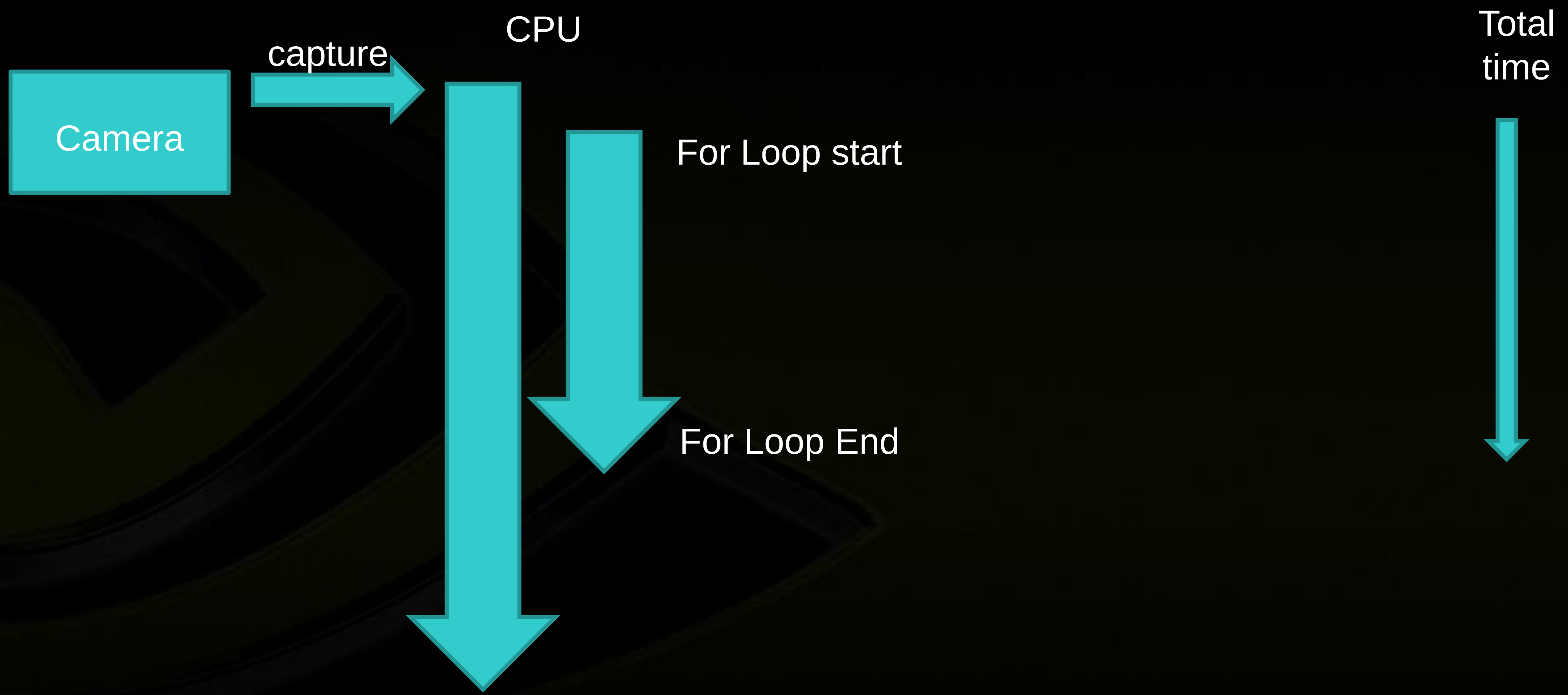
Debugging

Optimize/profile

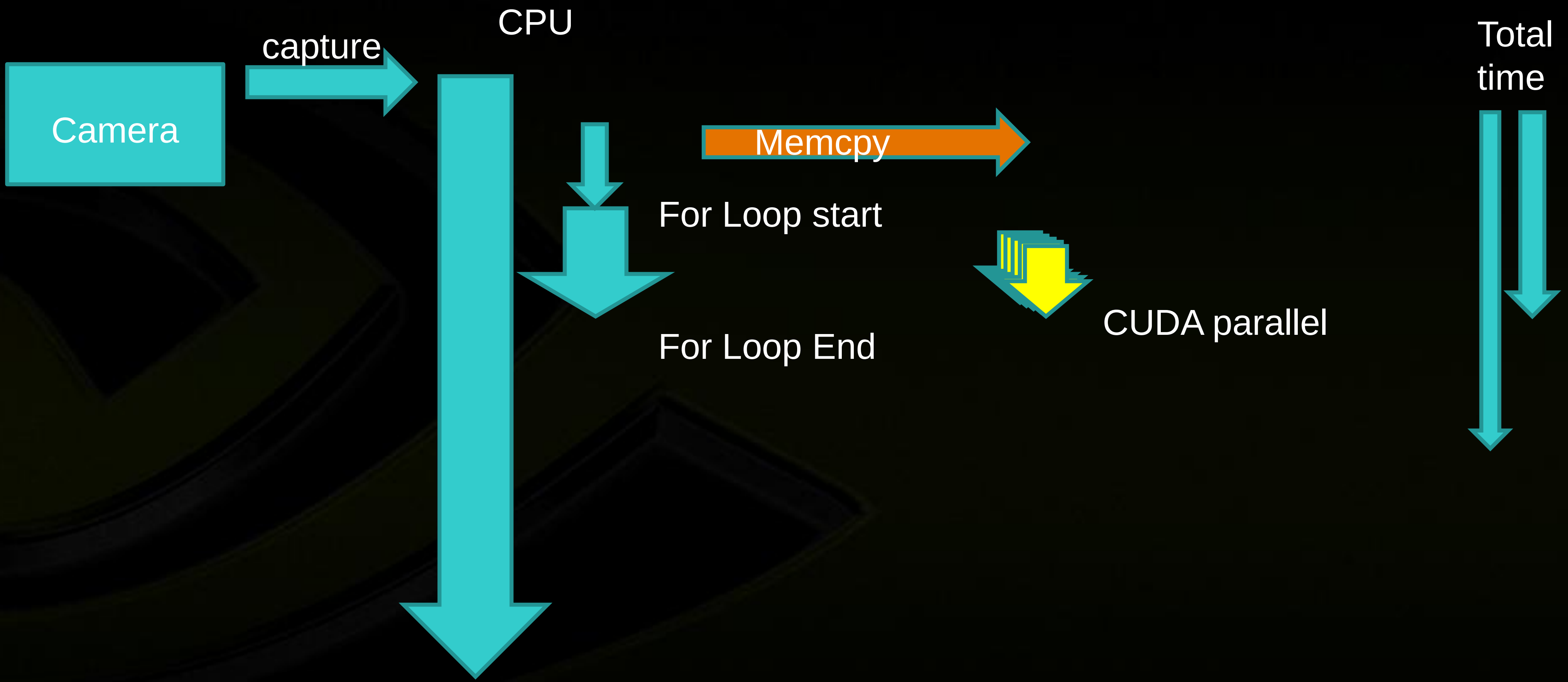
Release


CPU Program
Kernel Launch
SM
I-Cache
MT Issue
C-Cache
SP
SP
SP
SP
SP
SP
SP
SP
SFU
SFU
DP
Shared Memory
Block
thread

# Image Processing Diagram without CUDA



# Image Processing Diagram with CUDA



# 1D Heat Equation

## CUDA Toy
## for undergraduate student

NVIDIA

# 1D Heat Equation

# 1D Heat Equation

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

1D Heat Equation

$$u(0,t) = u(1,t) = 0$$

Boundary condition

$$u(x,0) = u_0$$

Initial condition

# Discretization

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

descretization

$$\frac{u_{j,i+1} - u_{j,i}}{\Delta t} = \alpha \frac{u_{j+1,i} - 2u_{j,i} + u_{j-1,i}}{\Delta x^2}$$

Forward difference with second order
i(time), j(space)

relation

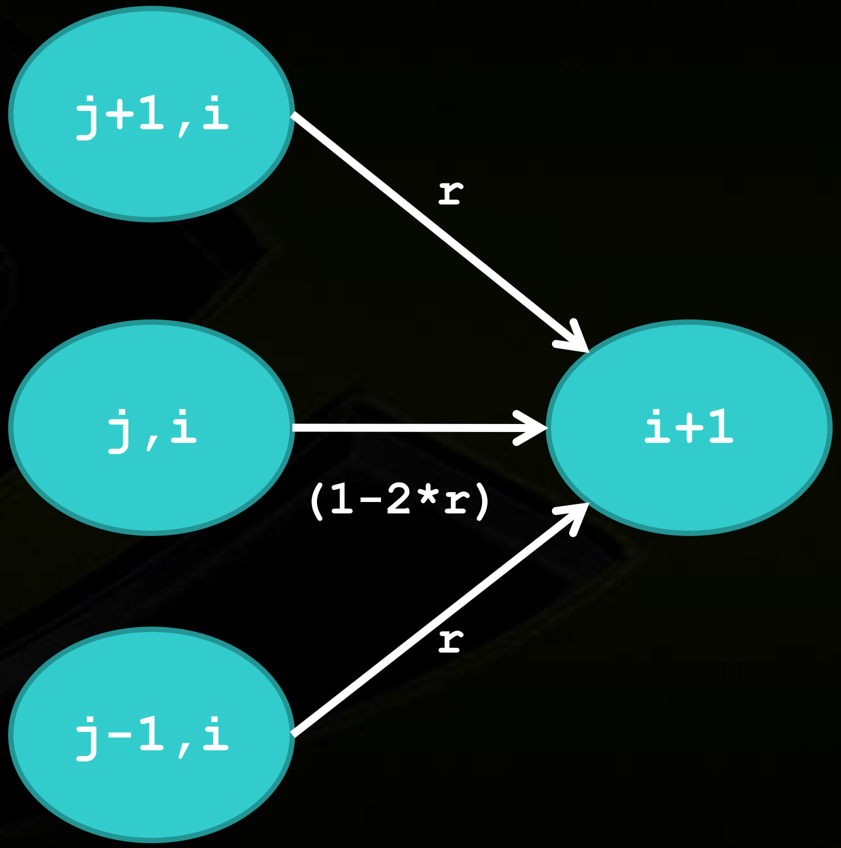
$$u_{j,i+1} = ru_{j-1,i} + (1-2r)u_{j,i} + ru_{j+1,i}$$
$$r = \alpha \Delta t / \Delta x^2$$

Explicit method

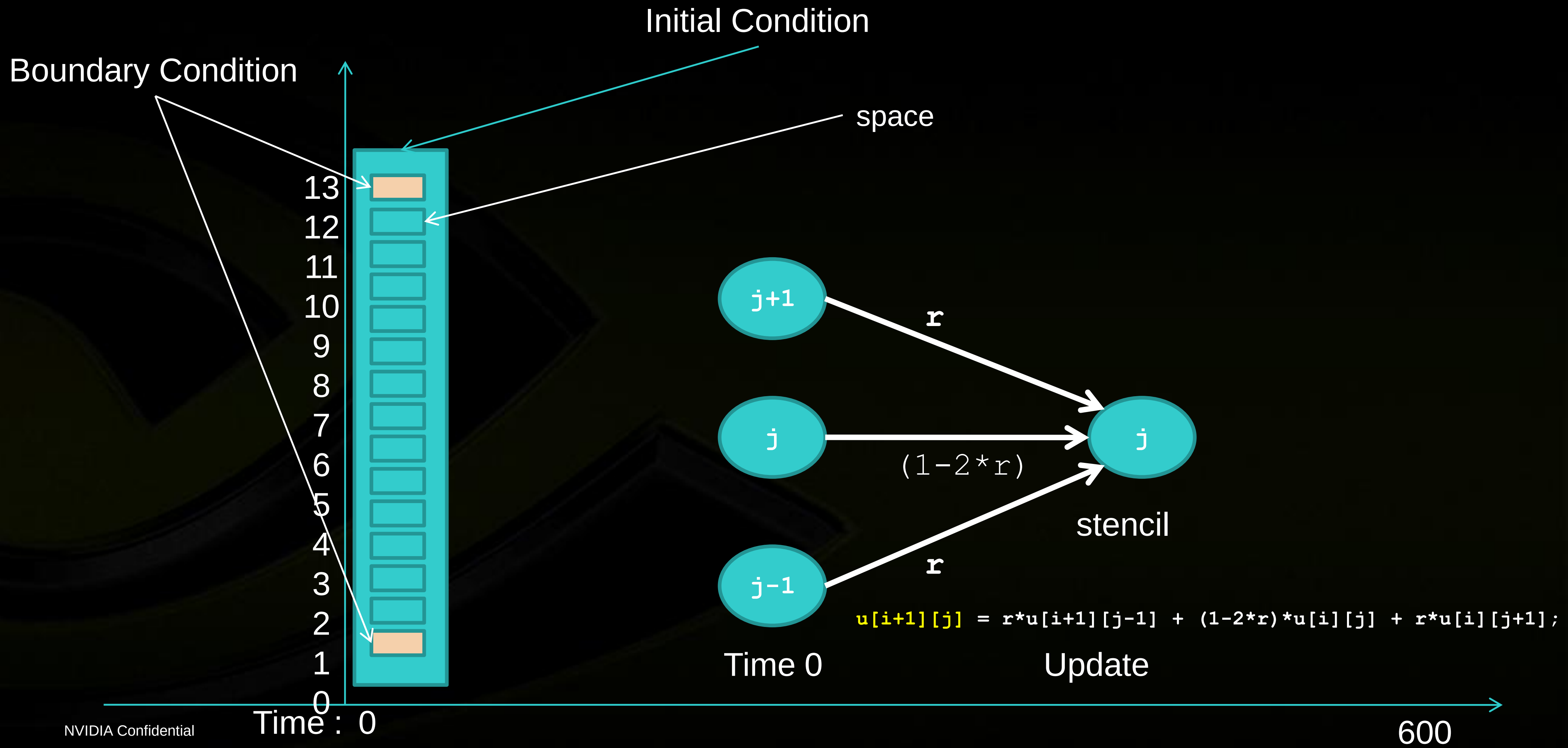
```
u[i+1][j] = r*u[i+1][j-1] + (1-2*r)*u[i][j] + r*u[i][j+1];
```

# Discretization (stencil)

$$u_{j,i+1} = ru_{j-1,i} + (1-2r)u_{j,i} + ru_{j+1,i}$$



# Conceptual Diagram



# Explicit Pseudo Code

- Parameter and data Initialization
  - Stencil, boundary/initial condition,

- FOR LOOP (time, i)

  FOR LOOP (stencil, j)

  Update the stencil relation

```
u[i+1][j] = r*u[i+1][j-1] + (1-2*r)*u[i][j] + r*u[i][j+1];
```

- Results

# CPU code

- u[i][j] vs. u[j]
  - u[i][j]  easy to develop
    - Possible to visualize the process
  - u[j]  efficient to use memory
    - Get the last result

# Main algorithm

```
for( i =0; i < N; i++) {

    for( j =0; j < M; j++){

        u[i+1][j] =         r * u[i+1][j-1]
                  + (1-2*r) * u[i  ][j]
                  + r       * u[i  ][j+1];

    }
}
```

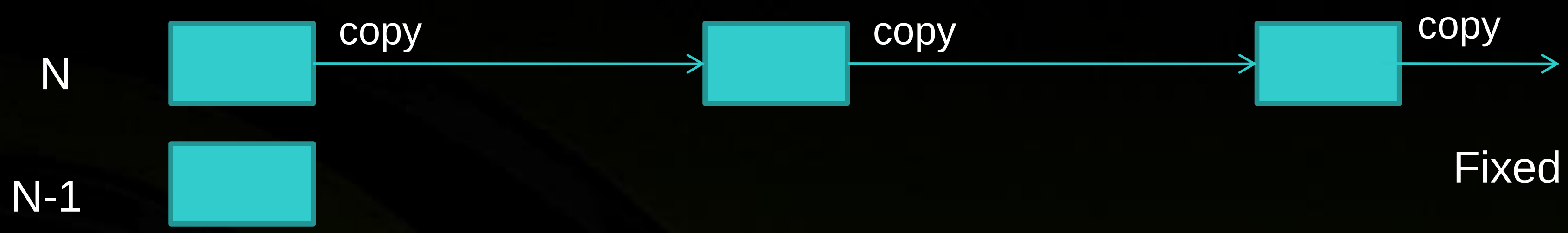
Time Iteration

Space Iteration

Heat relation

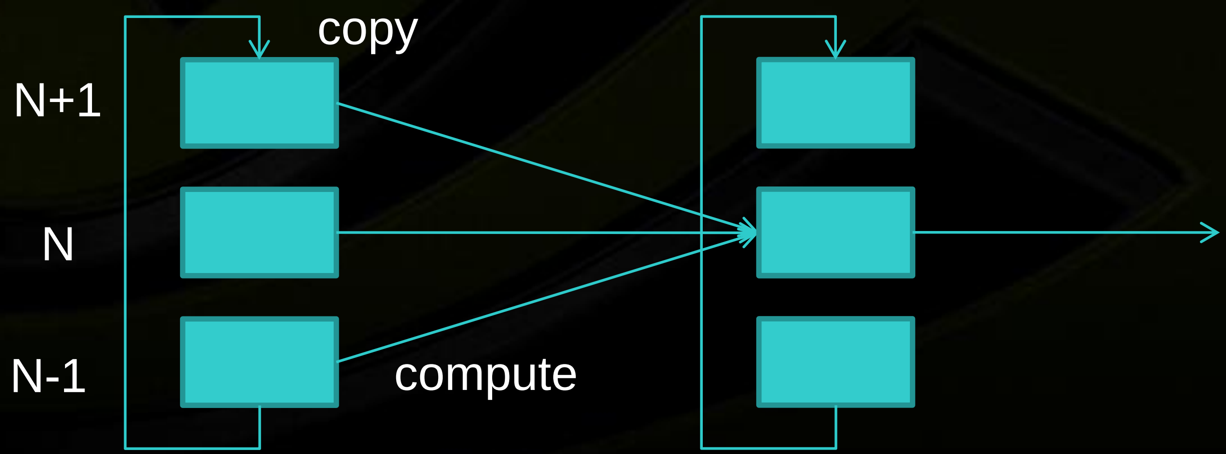
# Boundary Condition

Method1



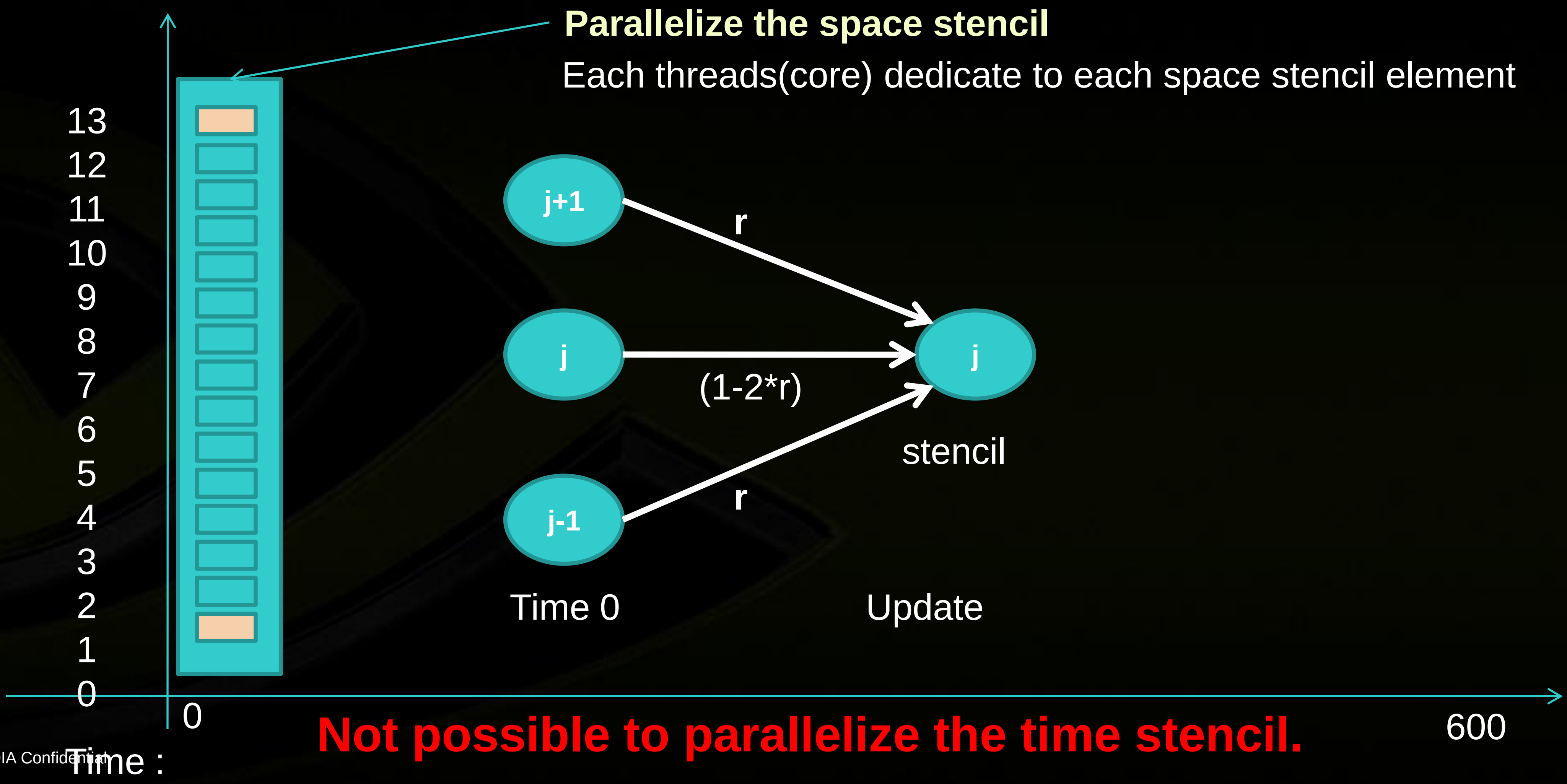

Fixed boundary

Method2




Free boundary

# How to Parallelize



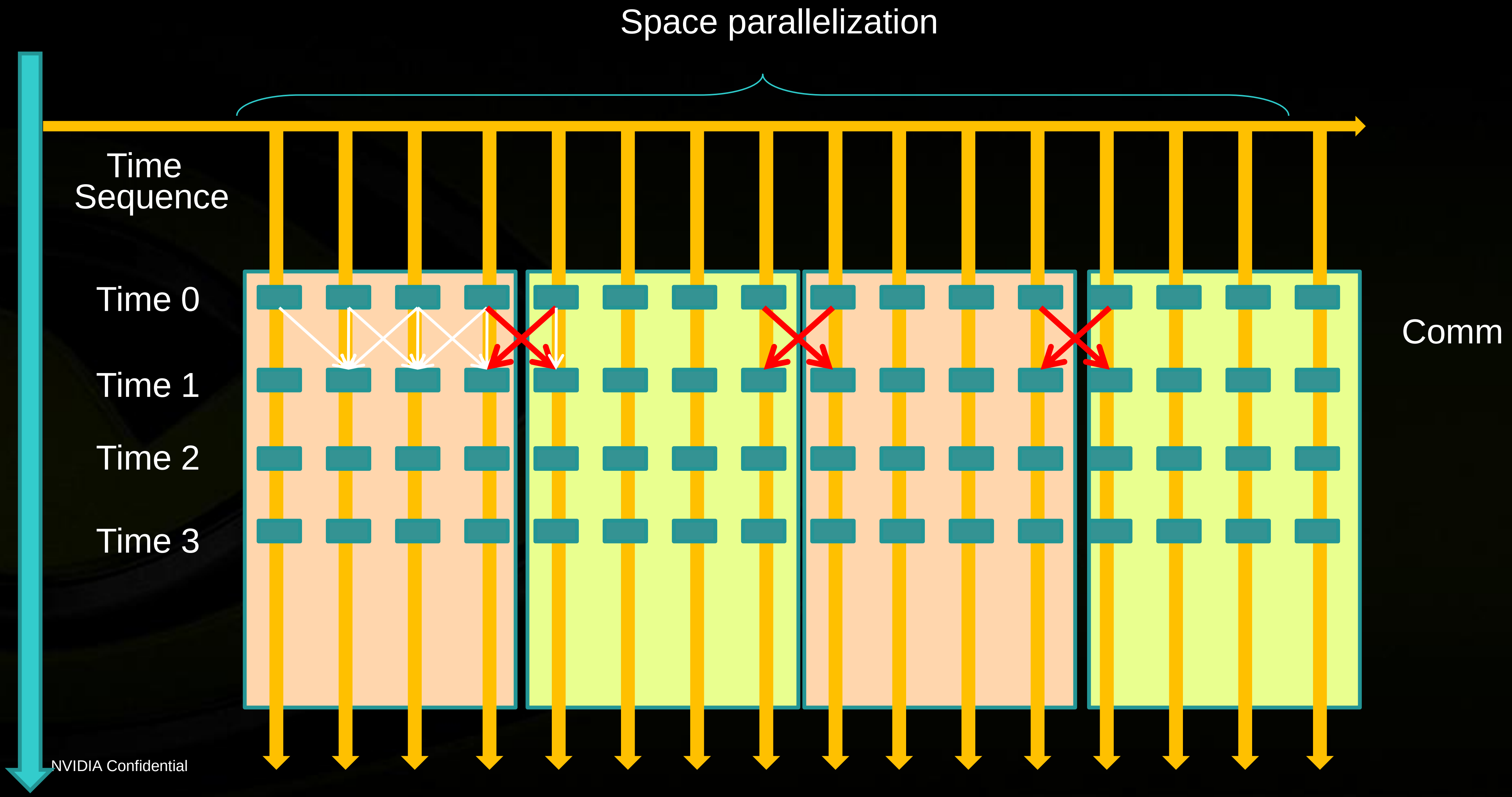

**Not possible to parallelize the time stencil.**

# How to parallelize

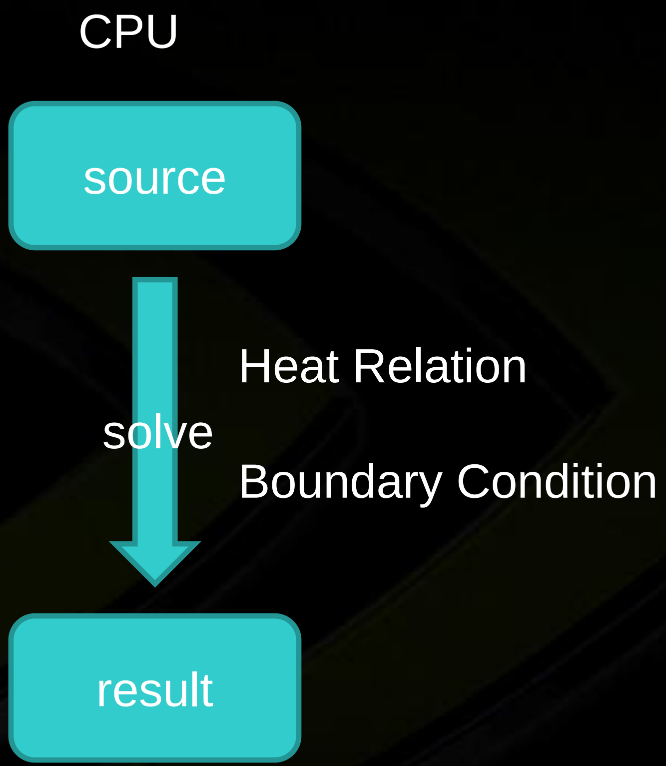
- Parameter and data Initialization
  - Stencil, boundary/initial condition,
- FOR LOOP (time, i)

  FOR LOOP (stencil, j)

  Update the stencil relation
- Results


Space parallelization
Time Sequence
Time 0
Time 1
Time 2
Time 3
Comm

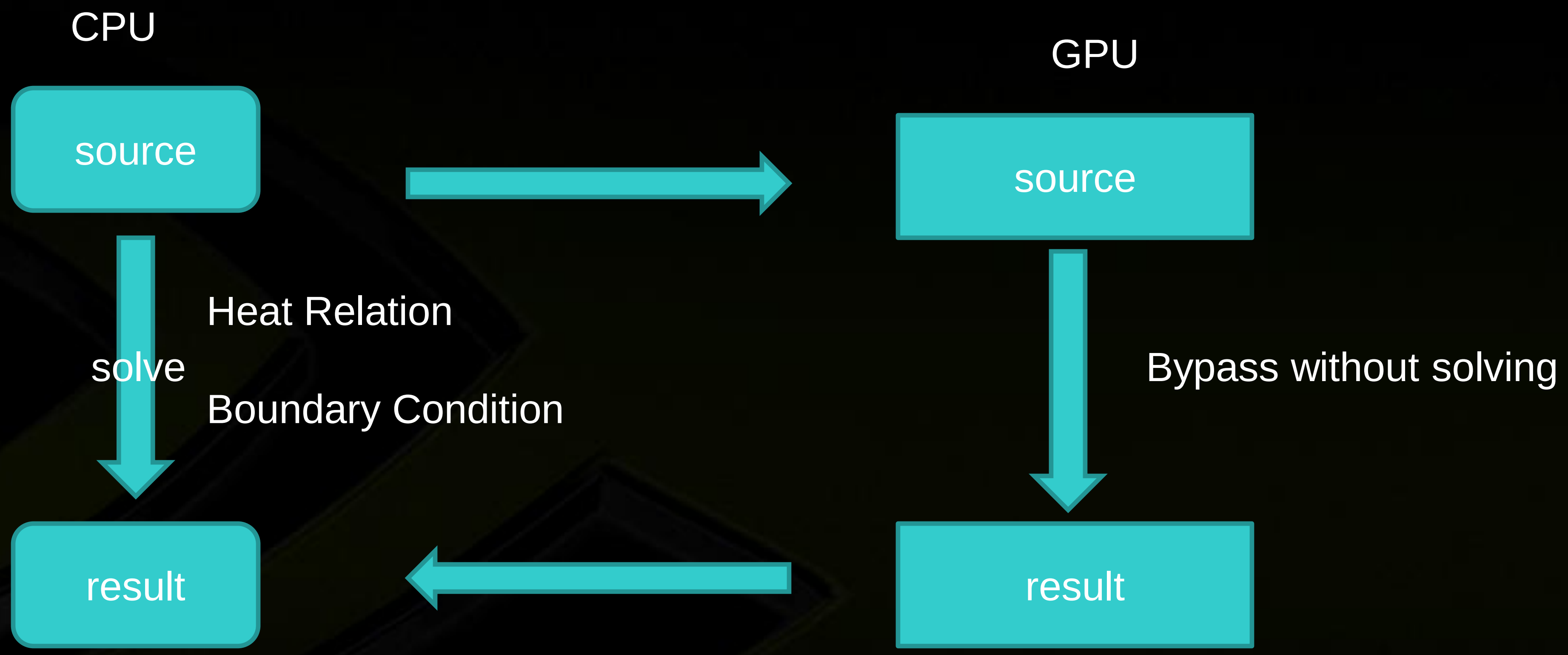
# CPUcode



# CPU code

```
do{
        time += dt; printf("aaaaaa %f\n",time);
        for(i=1; i < mesh+1; i++){
                temper[i].new_1 = (double) (1-2*r)*temper[i].old + r*(temper[i-1].old + temper[i+1].old);
                printf("\n processing \t %d %f %f \n",i, temper[i].new_1, temper[i].old);
        }
        temper[mesh+1].new_1 = temper[mesh-1].new_1;
         printf("\t  print results %d %f %f \n",mesh+1, temper[mesh+1].new_1,temper[mesh-1].new_1 );

        for(i=1; 1 < mesh+2; i++)
        temper[i].old = temper[i].new_1; printf("aa\t\t  %d %f %f \n",i, temper[i].new_1,temper[i].new_1 );
        if((++count % print_step)==0){
        printf("hh \t\t\t %10.5lf", time);
        for(i=0; i<mesh; i+=2)
                printf("df 8.4lf", temper[i].new_1);
        if(!(i%2))
                printf("fd %8.4lf\n", temper[mesh].new_1);
        }
                 printf("\n\n time print %f\n", time); getchar();
}while(time < end_time);
if((count % print_step)){
        printf("bghj %10.5lf", time);
        for(i=0; i<mesh; i+=2)
                printf("ahg 8.4lf", temper[i].new_1);
        printf("nhgf %8.4lf\n", temper[mesh].new_1);
}
```

# GPUcode-01 Memory Map



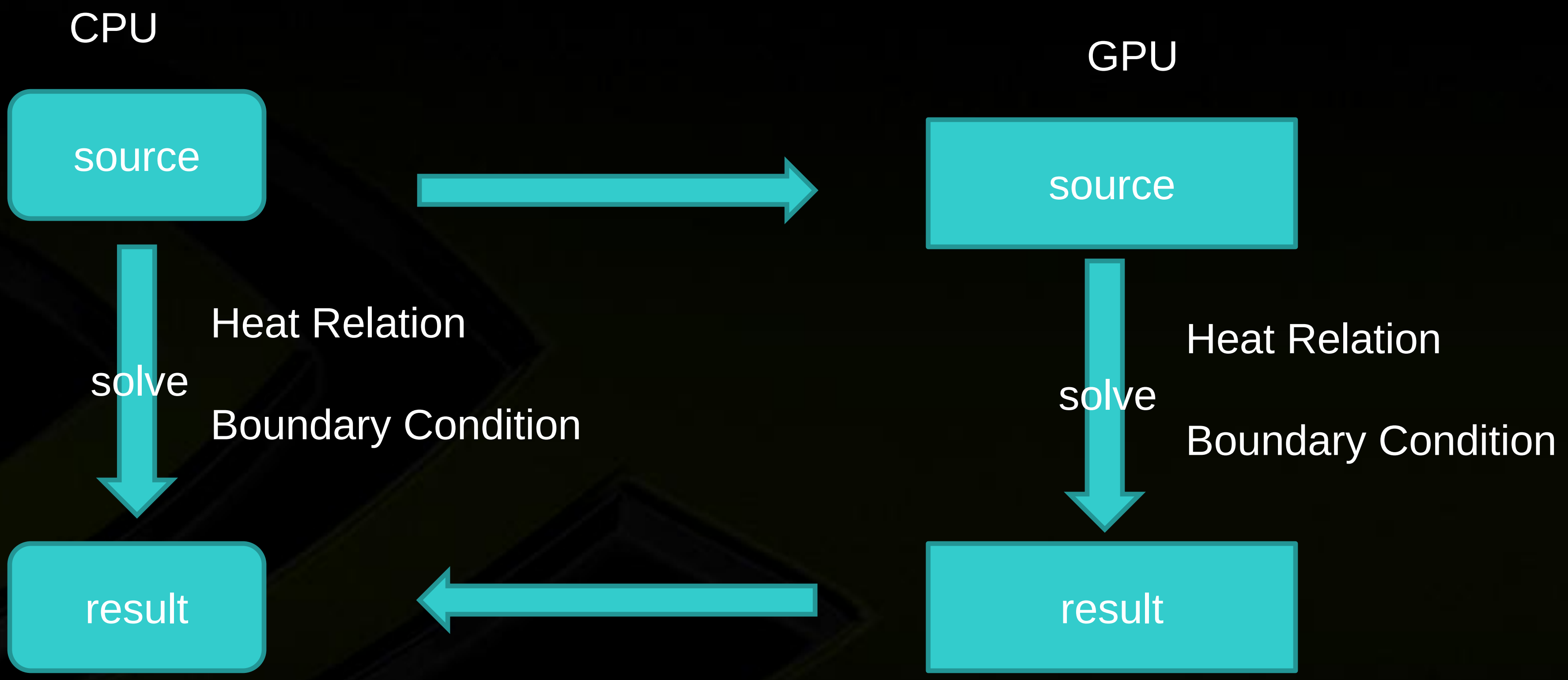
# GPUcode-01 Malloc Template

```
double* init_GPU_data(struct flow * temper, int mesh)
{
          double *u_dev; // for GPU data upload
          size_t gpuMemsize=sizeof(double)*(mesh+2) ;
          double *u_host; // temperal value
          cudaMalloc( (void**)&u_dev, gpuMemsize);cudaErr("malloc u_dev");
          u_host = (double *) malloc( gpuMemsize);
          for(int i=0;i<mesh;i++){
                    u_host[i]= temper[i].old;
                    printf("before %d : data initial :u_host[%d]= %f temper[%d].old  =%f\n", i, i, u_host[i], i,
temper[i].old);
          }
          cudaMemcpy(u_dev, u_host, gpuMemsize, cudaMemcpyHostToDevice);cudaErr("memcpy u_dev u_host");
          cudaMemcpy(u_host, u_dev, gpuMemsize, cudaMemcpyDeviceToHost);cudaErr("memcpy u_dev u_host");
          for(int i=0;i<mesh;i++){
                    printf("after %d : data initial :u_host[%d]= %f temper[%d].old  =%f\n", i, i, u_host[i], i,
temper[i].old);
          }

          free(u_host);

          return (double *)u_dev;
}
```

# GPUcode-01 Launch Template

```
void __global__ functionG(double *u_dev, int meshsize, double r, double bound )
{
          int idx = blockIdx.x*blockDim.x+threadIdx.x;
          int i = idx+1;
          if(idx <meshsize +1 ) {

                    u_dev[i]= i*0.01;
          }
          if(idx == 6){
          u_dev[idx+1]=u_dev[idx-1]=11;
          }
}

void compute_GPU(   double * u_dev,      double * u_host, double dt,  double dx,         double r, int mesh,
                    int print_step,     int count,           double time,        double end_time, double bound )
{
          size_t gpuMemsize = sizeof(double)*(mesh+2);
          //for( int i=0; i < 6000 ; i++){ //time step
                    functionG<<<4,5>>>(u_dev, mesh,r, bound);cudaErr2("kernel launch",1,0);
                    cudaMemcpy(u_host, u_dev, gpuMemsize, cudaMemcpyDeviceToHost);cudaErr("memcpy u_dev to u_host");
           for(int i=0; i<mesh+1; i++){
                    printf( " in kernel - GPU : temper[%d] ==> %f \n", i, u_host[i]);
          }
          //}
          return;
}
```

# GPUcode-02 Solving



# GPUcode-02 Malloc part

```
double* init_GPU_data(struct flow * temper, int mesh)
{
          double *u_dev; // for GPU data upload
          size_t gpuMemsize=sizeof(double)*(mesh+2) ;
          double *u_host; // temperal value
          cudaMalloc( (void**)&u_dev, gpuMemsize);cudaErr("malloc u_dev");
          u_host = (double *) malloc( gpuMemsize);
          for(int i=0;i<mesh;i++){
                    u_host[i]= temper[i].old;
                    printf("before %d : data initial :u_host[%d]= %f temper[%d].old  =%f\n", i, i, u_host[i], i,
temper[i].old);
          }
          cudaMemcpy(u_dev, u_host, gpuMemsize, cudaMemcpyHostToDevice);cudaErr("memcpy u_dev u_host");
          cudaMemcpy(u_host, u_dev, gpuMemsize, cudaMemcpyDeviceToHost);cudaErr("memcpy u_dev u_host");
          for(int i=0;i<mesh;i++){
                    printf("after %d : data initial :u_host[%d]= %f temper[%d].old  =%f\n", i, i, u_host[i], i,
temper[i].old);
          }

          free(u_host);

          return (double *)u_dev;
}
```

# GPUcode-02 Launch part

```
void __global__ functionG(double *u_dev, int meshsize, double r, double bound )
{
        int idx = blockIdx.x*blockDim.x+threadIdx.x;
        int i = idx+1;
        if(idx <meshsize +1 ) {
                u_dev[i] = (double) (1-2*r)*u_dev[i] + r*(u_dev[i-1]  + u_dev[i+1] );
        //      u_dev[i]= i*0.01;
        }
        if(idx == 6){
        u_dev[idx+1]=u_dev[idx-1]=11;
        }
}

void compute_GPU(   double * u_dev,      double * u_host, double dt,  double dx,          double r, int mesh,
                    int print_step,     int count,          double time,        double end_time, double bound )
{
        size_t gpuMemsize = sizeof(double)*(mesh+2);
        //for( int i=0; i < 6000 ; i++){ //time step
                functionG<<<4,5>>>(u_dev, mesh,r, bound);cudaErr2("kernel launch",1,0);
                cudaMemcpy(u_host, u_dev, gpuMemsize, cudaMemcpyDeviceToHost);cudaErr("memcpy u_dev to u_host");
         for(int i=0; i<mesh+1; i++){
                printf( " in kernel - GPU : temper[%d] ==> %f \n", i, u_host[i]);
        }
        //}
        return;
}
```

# How to Optimize?

# Monte Carlo Simulation

# Computation Time

Malliavin MC results

| type | Black-Sholes | 5000 | 10000 | 50000 | 100000 | 200000 | 300000 |
|---|---|---|---|---|---|---|---|
| Price | 12.8216 | 12.8706 | 12.8721 | 12.8413 | 12.8525 | 12.8559 | 12.8480 |
| Err | 0.000% | 0.38% | 0.39% | 0.15% | 0.24% | 0.27% | 0.21% |
| Delta | 0.5858 | 0.5724 | 0.5887 | 0.5826 | 0.5826 | 0.5829 | 0.5824 |
| Err | 0 | 2.28% | -0.51% | 0.53% | 0.53% | 0.49% | 0.58% |
| Gamma | 0.0130 | 0.0112 | 0.0134 | 0.0127 | 0.0127 | 0.0127 | 0.0127 |
| Err | 0.00% | 13.39% | -3.34% | 2.07% | 2.26% | 2.21% | 2.47% |
| Total Time | 0:00 | 00:03 | 00:05 | 00:25 | 00:50 | 01:44 | 02:33 |

time : 100 sec
target1 : > 1 sec (100X)
Target2 : >0.001 sec (2000X)

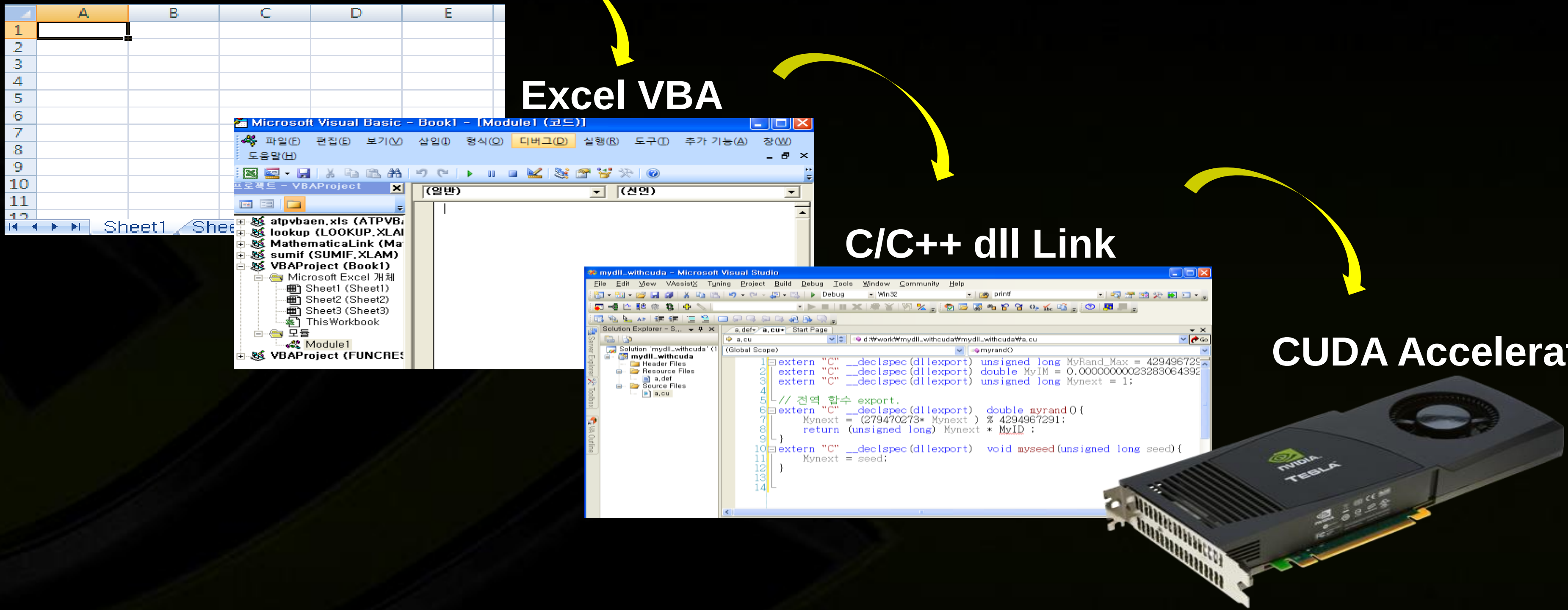
# Monte Carlo Simulation for Finance

Excel Sheet

Excel VBA

C/C++ dll Link

CUDA Acceleration



# Monte Carlo Code with VBA

```
function  MC( S As Double, X As Double, T As Double, R As Double, _
                                Vol As Double, Q As Double, No As Double, rtype As String) As Double
    Simul_No = No
     dt = 1   'dt = 1 / 365
        For K = 0 To Simul_No - 1
           Juga = S
             For i = 0 To MaxStep - 1
                   Juga = Juga * Exp( (R - Q - Vol ^ 2 / 2) * dt + Vol * Sqr(dt) * MakeNorsD() )
              Next
             price = Exp(-R * T) * Max(Juga - X, 0)
            sum_price = sum_price + price
        Next
    MC = sum_price / Simul_No
End function
```

# *Malliavin* Greeks

- **Greek computation for Monte Carlo simulation**

$$\frac{\partial}{\partial s} E[f(S)] \approx \frac{E[f(S_0 + \Delta S)] - E[f(S_0)]}{\Delta S}$$

- ***Malliavin* approach**

$$\frac{\partial}{\partial s} E[f(S)] \approx E[f(S) \cdot \underline{W(S)}]$$

***Malliavin*** **weights**

**With Malliavin approach, we can save the computation time.**

# Problem

To compare the accuracy, we compute the Price, Delta and Gamma of Vanilla Call option.

Approach

1. Closed Form solution (VBA,C)
2. Monte (VBA, C)
3. Malliavin (VBA,C, CUDA v1, v2)

# Malliavin Monte Carlo Code with VBA

```
function  Malliavin( S As Double, X As Double, T As Double, R As Double, _
                    Vol As Double, Q As Double, No As Double, rtype As String) As Double
  Simul_No = No
   dt = 1   'dt = 1 / 365
     For K = 0 To Simul_No - 1
        Juga = S
         For i = 0 To MaxStep - 1
             Juga = Juga * Exp( (R - Q - Vol ^ 2 / 2) * dt + Vol * Sqr(dt) * MakeNorsD() )
         Next
         WT = (Log(Juga) - Log(S) - (R - Q - 1 / 2 * Vol ^ 2) * T) / Vol
         WT_delta = (WT / (S * Vol * T))
         WT_gamma = (1 / (Vol * T * S ^ 2)) * (WT ^ 2 / (Vol * T) - WT - 1 / Vol)

        price = Exp(-R * T) * Max(Juga - X, 0)
        delta = Exp(-R * T) * Max(Juga - X, 0) * WT_delta
        gamma = Exp(-R * T) * Max(Juga - X, 0) * WT_gamma

     sum_price = sum_price + price
     sum_delta = sum_delta + delta
     sum_gamma = sum_gamma + Gamma
     Next

    Malliavin = sum_delta / Simul_No
```



```
End function
```

# Step1 Malliavin Monte Carlo C language sketch

```
void Malliavin( double S , double X , double T ,  double R, double Vol, double Q, long No){
    long Simul_No = No;
    double dt = 1;   // dt = 1 / 365
        for ( int K = 0; K< Simul_No - 1    ; K++){
            Juga = S ;
             for (int i = 0; i< MaxStep - 1 ki++){
                      Juga = Juga  *  exp( (R - Q - Vol ^ 2 / 2)  *  dt + Vol  *  sqrt(dt)  *  norm() ); // rand with box muller
              }

            WT = (Log(Juga) - Log(S) - (R - Q - 1 / 2  *  Vol ^ 2)  *  T) / Vol;
            WT_delta = (WT / (S  *  Vol  *  T));
            WT_gamma = (1 / (Vol  *  T  *  S ^ 2))  *  (WT ^ 2 / (Vol  *  T) - WT - 1 / Vol);

            price = Exp(-R  *  T)  *  max(Juga - X, 0);
            delta = Exp(-R  *  T)  *  max(Juga - X, 0)  *  WT_delta;
            gamma = Exp(-R  *  T)  *  max(Juga - X, 0)  *  WT_gamma;
           sum.price = sum.price + price;
           sum.delta = sum.delta + delta;
           sum.gamma = sum.gamma + Gamma ;
        }
     r.price = sum.delta / Simul_No
     r.delta = sum.delta / Simul_No
     r.gamma = sum.delta / Simul_No
     return 0;
}
```

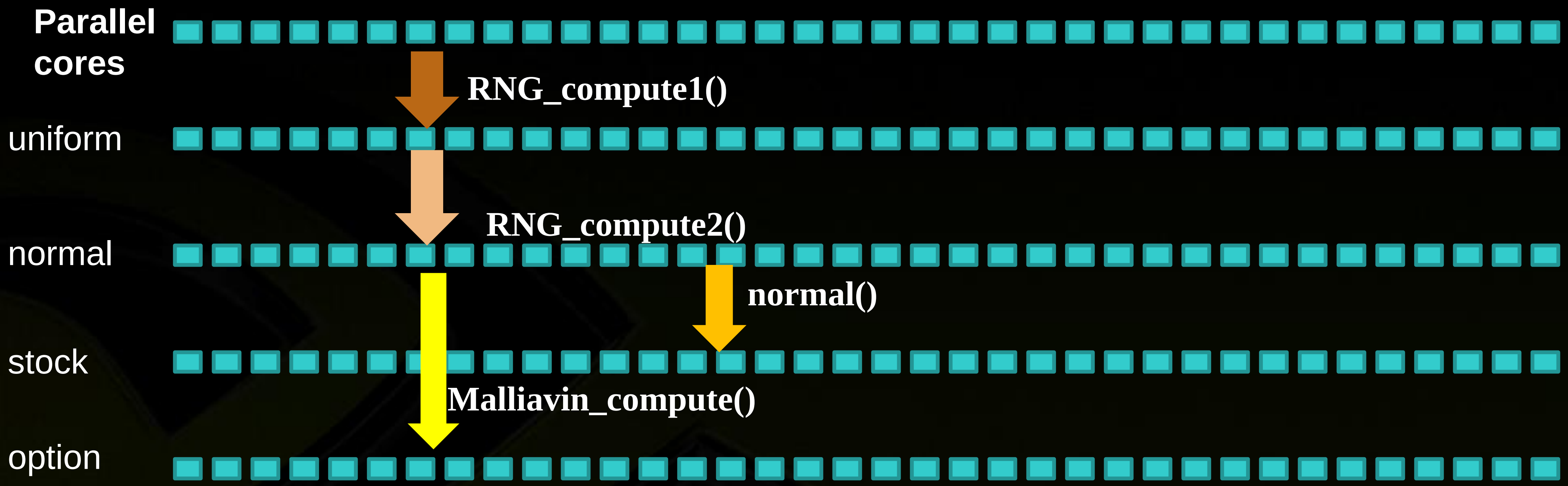
# Step2 Sketch (kernel part)

```
Void __global__  Malliavin_compute( double S , double X , double T ,  double R, double Vol, double Q, long No){
    long Simul_No = No;
    double dt = 1;   // dt = 1 / 365
        for ( int K = 0; K< Simul_No - 1   ; K++){
            Juga = S ;
             for (int i = 0; i< MaxStep - 1 ki++){
                      Juga = Juga  *  exp( (R - Q - Vol  *Vol / 2)  *  dt + Vol  *  sqrt(dt)  *  norm(k,i) );
              }

            WT = (log(Juga) - log(S) - (R - Q - 1 / 2  *  Vol  *Vol)  *  T) / Vol;
            WT_delta = (WT / (S  *  Vol  *  T));
            WT_gamma = (1 / (Vol  *  T  *  S  *S))  *  (WT*WT / (Vol  *  T) - WT - 1 / Vol);

            price = Exp(-R  *  T)  *  max(Juga - X, 0);
            delta = Exp(-R  *  T)  *  max(Juga - X, 0)  *  WT_delta;
            gamma = Exp(-R  *  T)  *  max(Juga - X, 0)  *  WT_gamma;
           sum.price = sum.price + price;
           sum.delta = sum.delta + delta;
           sum.gamma = sum.gamma + Gamma ;
        }
     r.price = sum.delta / Simul_No
     r.delta = sum.delta / Simul_No
     r.gamma = sum.delta / Simul_No
     return 0;
}
```

**Simm_No =**
**Total Sim /(N threads* M blocks)**

**Real Price =**
**Sum (r.price ) / (N*M)**

# Step2 Parallel Memory Map




data in global memory

```
(float  *)  __device__ normal(int k, int, j, int size_j, float * normal) {
    int index = k*size_j + j;
    return &normal[index];
}
```

# Step2 Sketch (host part)

```
#include <stdio.h>

__global__ RNG_compute(parameter);
__global__ Malliavin_compute(parameter);

main(){

    malloc(); //cpu malloc
    cudaMalloc(); //GPU malloc
    cudaMemcpy(); // transfer

    RNG_compute 1<<<N,M>>> (parameter); // generate RNG (uniform)

    RNG_compute2 <<<N,M>>> (parameter); // generate RNG ( BM,Moro)

    Malliavin_compute <<<N,M>>> (parameter); // simulation

    cudaMemcpy(); //get results

    return 0;

}
```

# Step2 Malliavin Monte Carlo CUDA language sketch (rng part1)

```
__global__
static void RNG_rand48_get_int(uint2 *state, int *res, int num_blocks, uint2 A, uint2 C)
{
  const int nThreads = blockDim.x*gridDim.x;


  int   nOutIdx = threadIdx.x + blockIdx.x*blockDim.x;
  uint2 lstate = state[nOutIdx];
  int i;
  for (i = 0; i < num_blocks; ++i) {

    res[nOutIdx] = ( lstate.x >> 17 ) | ( lstate.y << 7);
    nOutIdx += nThreads;

    lstate = RNG_rand48_iterate_single(lstate, A, C);
  }

  nOutIdx = threadIdx.x + blockIdx.x*blockDim.x;
  state[nOutIdx] = lstate;
}
```

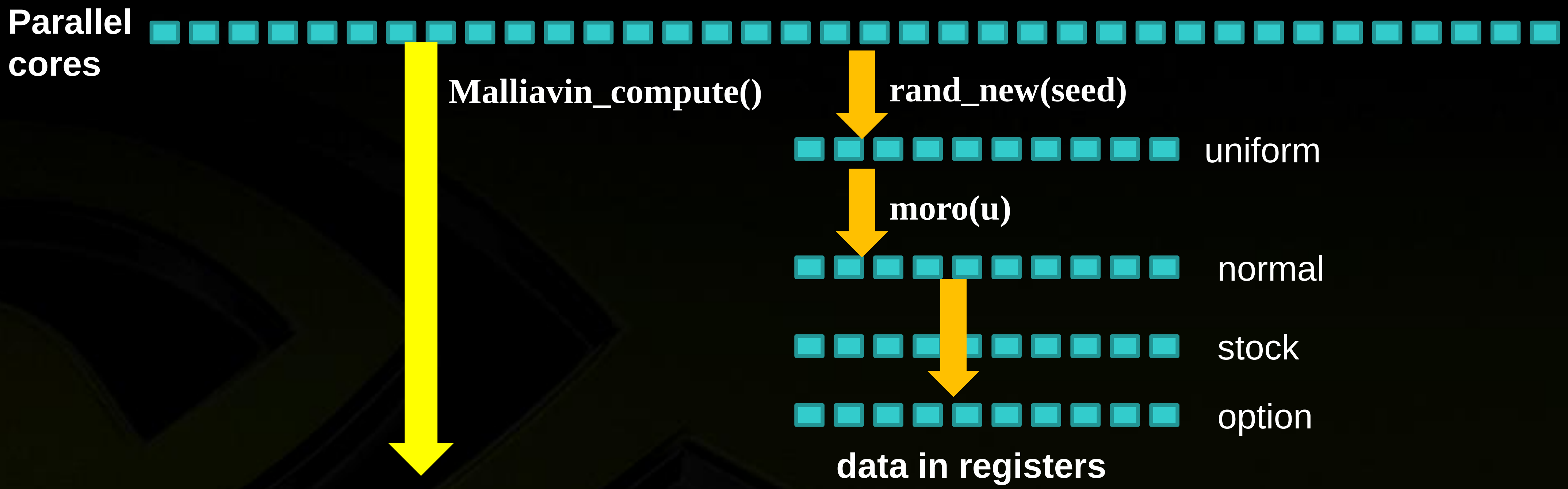
# Step2 Malliavin Monte Carlo CUDA language sketch (rng part2)

```
Void __global__  RNG_compute( int * uniform , float * normal, int length){

int index = blockDim.x*blockIdx.x+threadIdx.x;

__shared__  int s[i];
__shared__ float s_r[i];

if(  threadIdx.x ==0){
    for (int i = 0 ; i<blockDim.x ; i++){
        s[i]=uniform[blockDim.x*blockIdx.x + i];  // load uniform
    }
}
   s_r[threadIdx.x] = (float) moro(s[threadIdx.x]);  // moro inversion with parallel

if(  threadIdx.x ==0){
    for (int i = 0 ; i<blockDim.x ; i++){
        s[blockDim.x*blockIdx.x+i]= s_r[i] ];   // save normal
    }
}


}
```

## Box-Muller vs Moro Inversion

```
__device__ void BoxMuller(float& u1, float& u2){
    float   r = sqrtf(-2.0f * logf(u1));
    float phi = 2 * PI * u2;
    u1 = r * __cosf(phi);
    u2 = r * __sinf(phi);
}
__device__ Moro( float u  ){
        // skip the const value
        x = u - 0.5;
    if (abs(x) < 0.42) {
        r = x*x;
        r = x * (((a4 * r + a3) * r + a2) * r + a1) / ((((b4 * r + b3) * r + b2) * r + b1) * r + 1);
    } else{
        if (x > 0)  r = log(-log(1 - u));
        if (x <= 0)  r = log(-log(u));
        r = c1 + r * (c2 + r * (c3 + r * (c4 + r * (c5 + r * (c6 + r * (c7 + r * (c8 + r * c9)))))));
        if (x <= 0)  r = -r;
                                    }f
    return  r;
}
```

# Step3 New approach for Direct LCG and Parallel Memory Map




```
double rand_new( int next){
    next =  (a * next + b) % M;
return (double) next * (1/M);
}
```

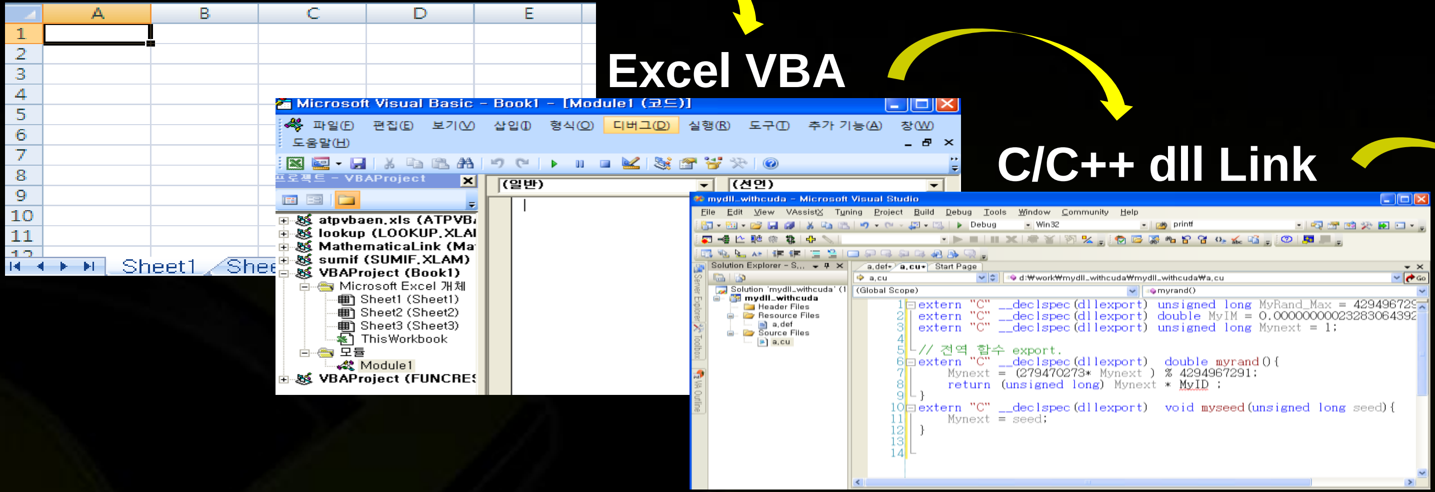
# Platform for finance

NVIDIA

**Excel Sheet**

**Excel VBA**

**C/C++ dll Link**




**Socket communication**

**Grid Manager**

**GPU cluster**

12 nodes system (1/2 for backup)

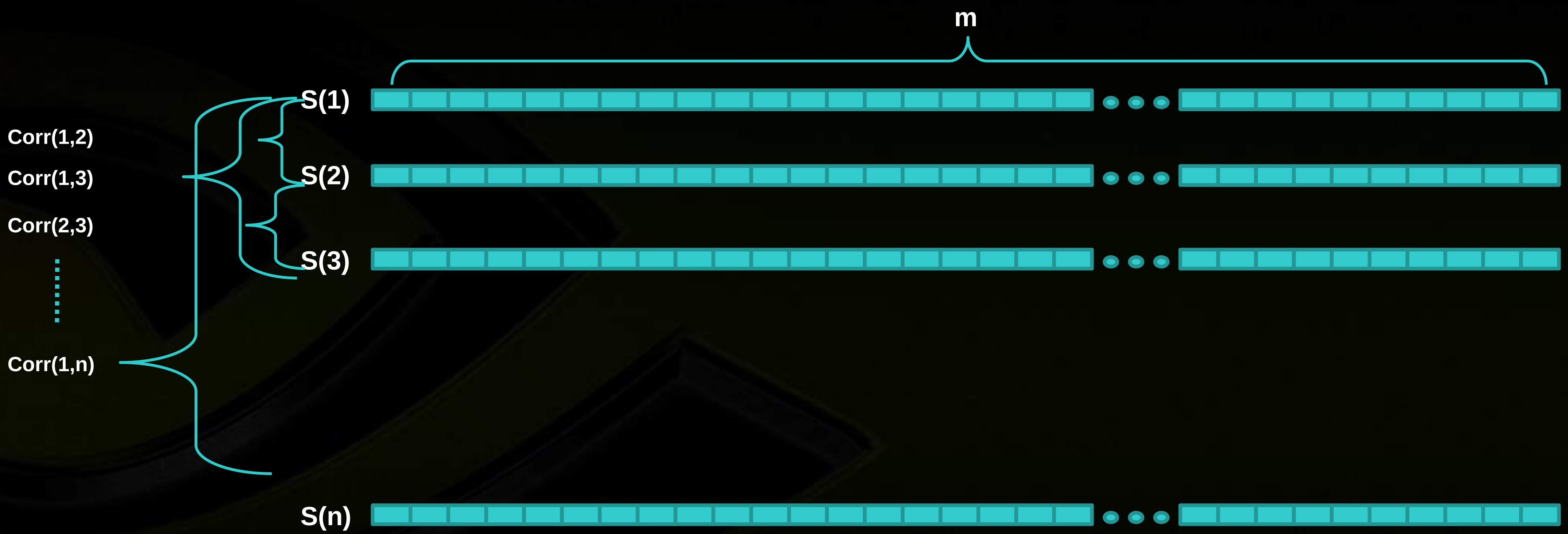
**12*8 GPU*448 core**
**= 43,000 core**

8 GPU per node

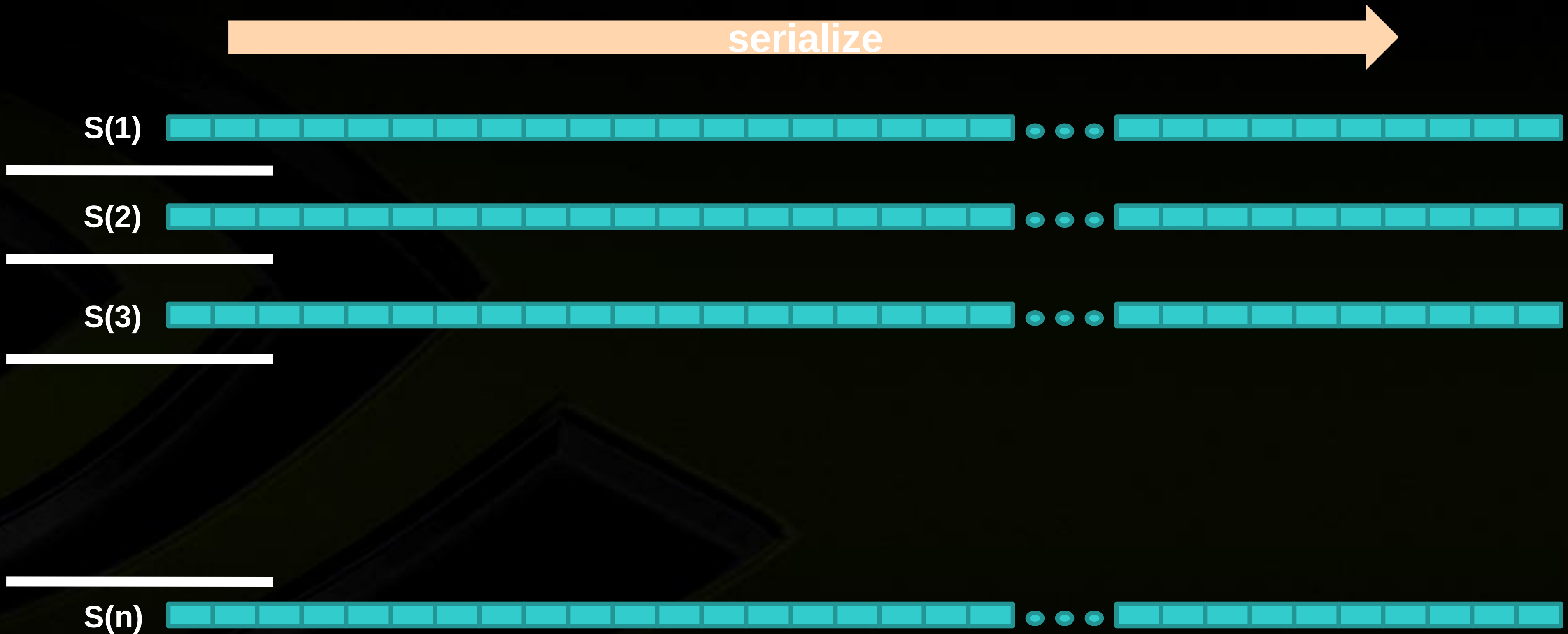
# How to Optimize?

# Time Series Analysis

# Multivariate Time-series

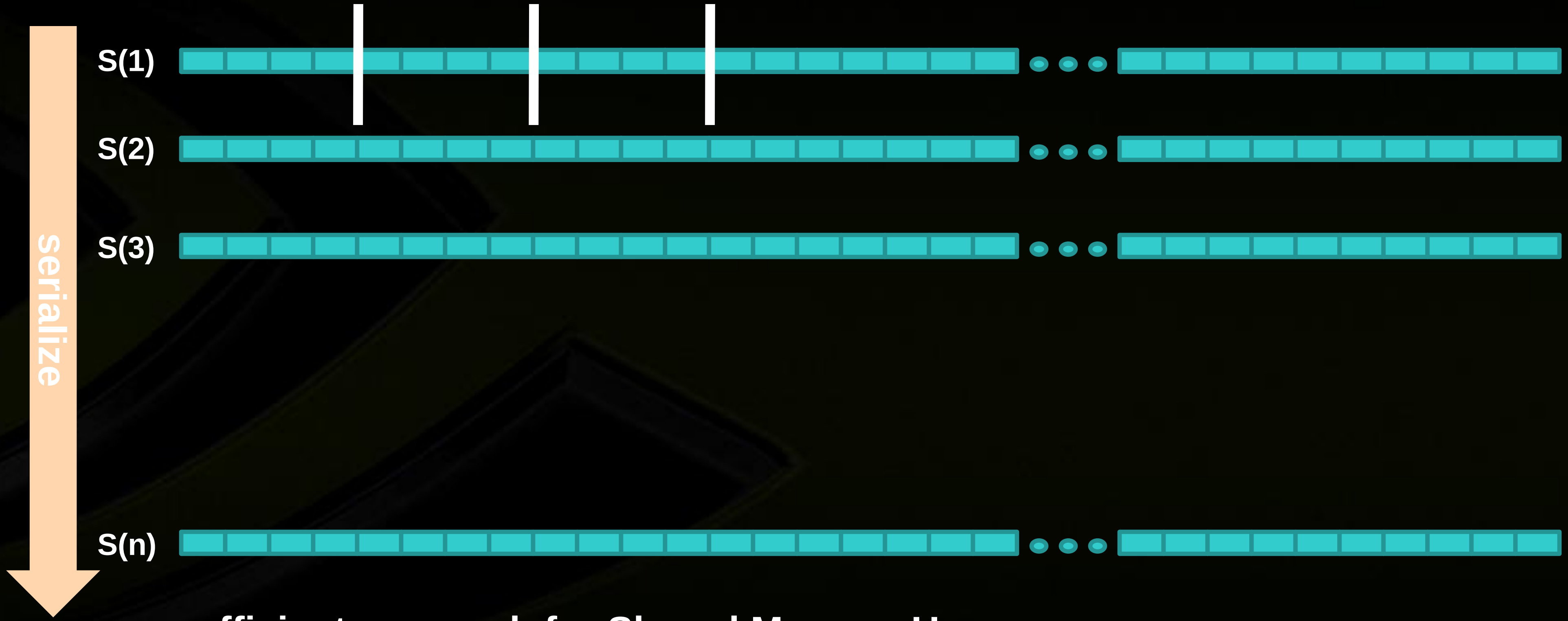


# How to parallelize with CUDA



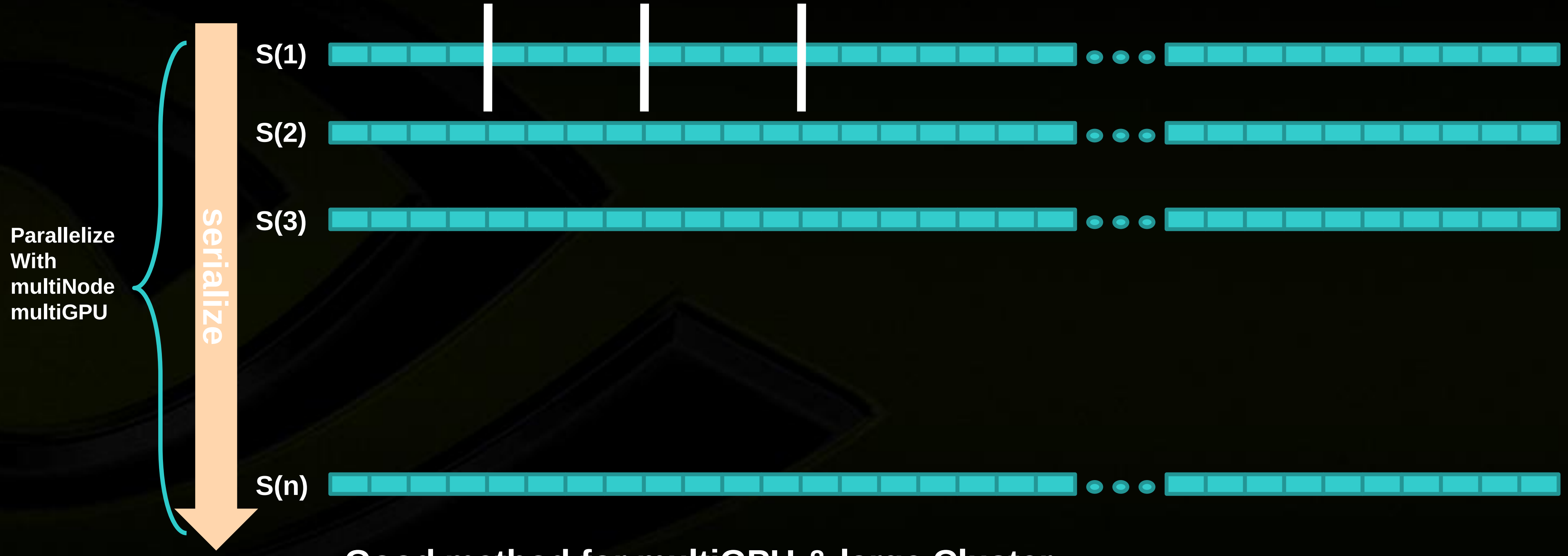

**Unefficient approach for Shared Memory Usage**

# How to parallelize with CUDA




**efficient approach for Shared Memory Usage**
**Need to reduction technique for mean etc.**

# How to parallelize with CUDA




Good method for multiGPU & large Cluster

# In pair(I,J), Pearson Correlation Coefficient

$$\rho = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 \sum (y_i - \bar{y})^2}}$$

$$\sum (x_i - \bar{x})(y_i - \bar{y}) = \sum x_i y_i - \bar{x}\sum y_i - \bar{y}\sum x_i + \sum \overline{xy}$$

$$= \sum x_i y_i - n\overline{xy}$$

$$\sum (x_i - \bar{x})^2 = \sum x_i^{\ 2} - n\bar{x}^2$$

$$\sum (y_i - \bar{y})^2 = \sum y_i^{\ 2} - n\bar{y}^2$$

$$\rho = \frac{\sum x_i y_i - n\overline{xy}}{\sqrt{(\sum x_i^{\ 2} - n\bar{x}^2)(\sum y_i^{\ 2} - n\bar{y}^2)}}$$

**We can parallelize the Sumation !!**
**After sumation, find the mean.**

# How to parallelize with CUDA : Flow Chart

NVIDIA

## Method 1

Input A, B

Find the mean of A, B
start to sum (Ai), (Bi)

Find Cov(A,B), Cov(A,A), Cov(B,B)
start to sum (Ai,Bi), (Ai^2), (Bi^2) with mean

Benefit : Easy to implementation with two function

**R+CUDA project**

http://brainarray.mbni.med.umich.edu/Brainarray/rgpgpu/

## Method 2

Input A, B

Start to sum (Ai), (Bi), (Ai,Bi), (Ai^2), (Bi^2)

Find the mean of A, B

Find Cov(A,B), Cov(A,A), Cov(B,B)

Benefit : oneshot sum (speed-up)

# In pair(i,j), Pearson Correlation Coefficient

FOR  (i, j) – pair : serial

FOR k (time-series) : parallel

compute

reduction for results

compute mean(i), mean(j),

compute cov(i,j), cov(i,i) ,cov(j,j)

compute corr(i,j)

# In pair(i,j), Pearson Correlation Coefficient

FOR (i, j) – pair : serial

FOR k (time-series) : parallel

FOR **shared memory**

$\sum x_i \quad \sum y_i \quad \sum x_i y_i \quad \sum x_i^2 \quad \sum y_i^2$

compute

reduction for results

compute mean(i), mean(j),

compute cov(i,j), cov(i,i) ,cov(j,j)

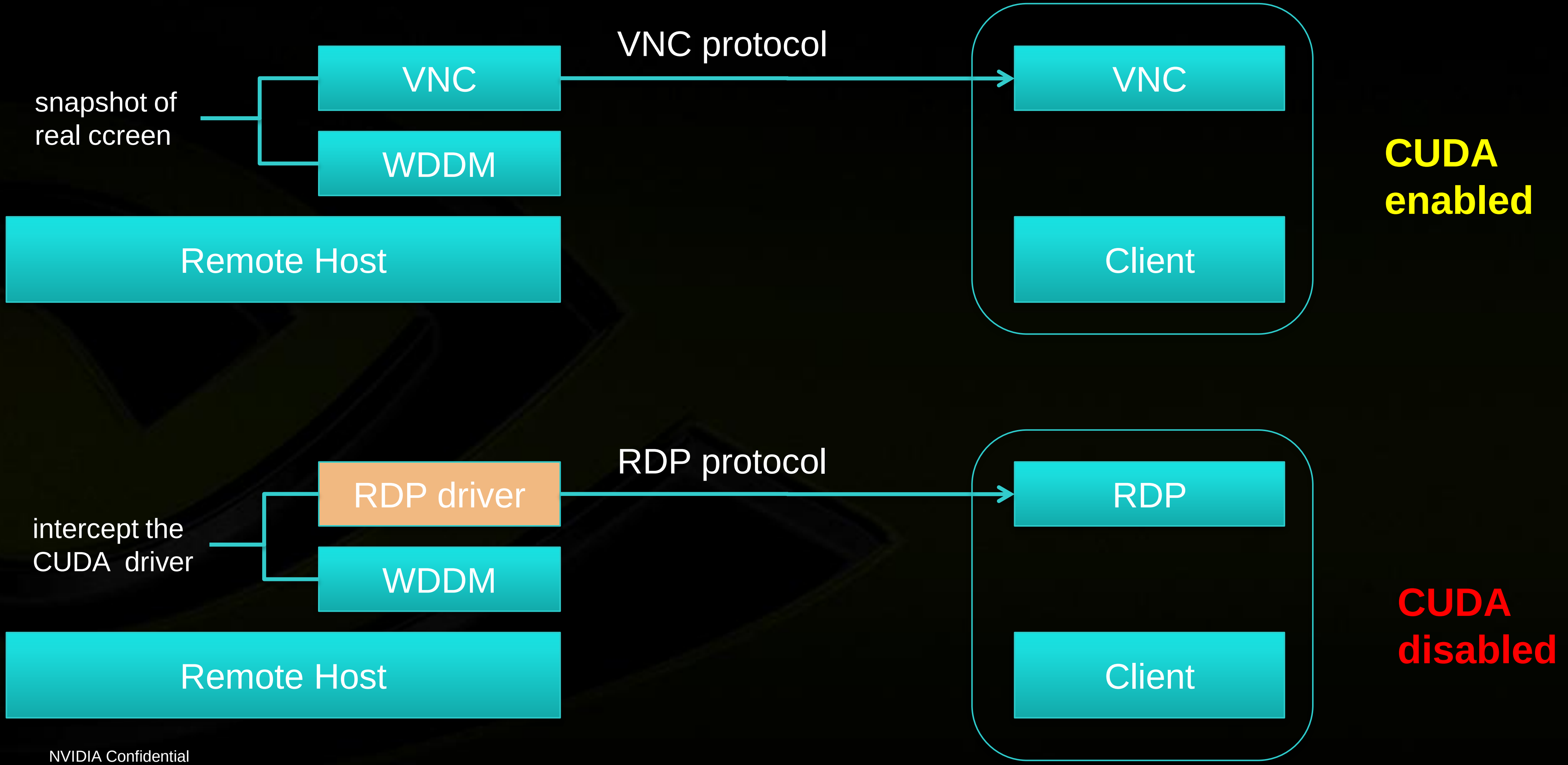
compute corr(i,j)

# How to Optimize?

# Focus on Optimization

# System Optimization Tips before start CUDA programming

# Remote Use

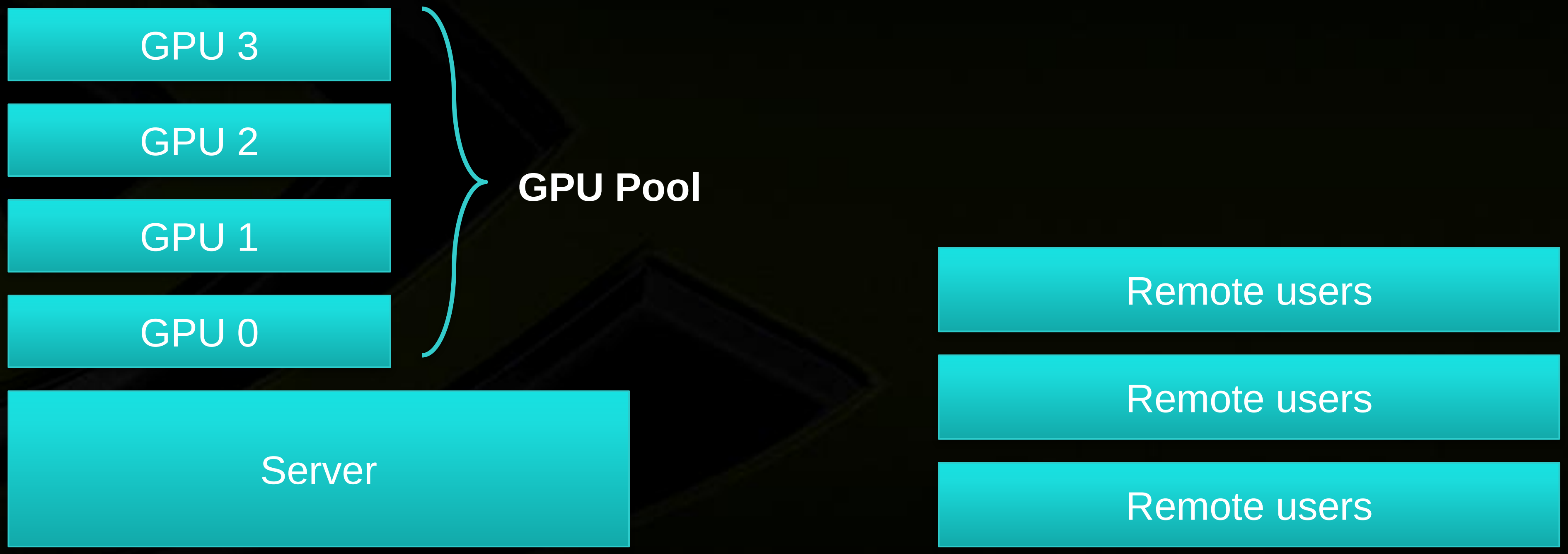


VNC protocol
snapshot of
real ccreen
VNC
WDDM
Remote Host
VNC
Client
CUDA
enabled
RDP protocol
intercept the
CUDA driver
RDP driver
WDDM
Remote Host
RDP
Client
CUDA
disabled

# TCC driver


- Remote Desktop
- Kernel Launch Over Head
- GPU exclusive Mode
- Windows Service for Session0/1
- large single Malloc

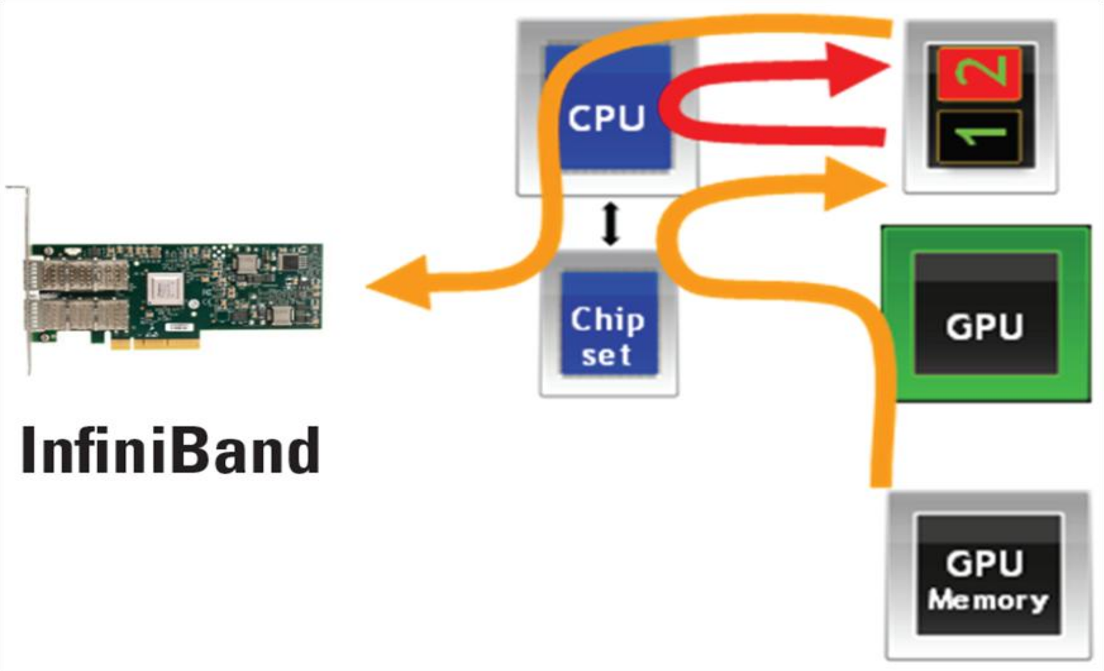
# GPU exclusive Mode for multiuser



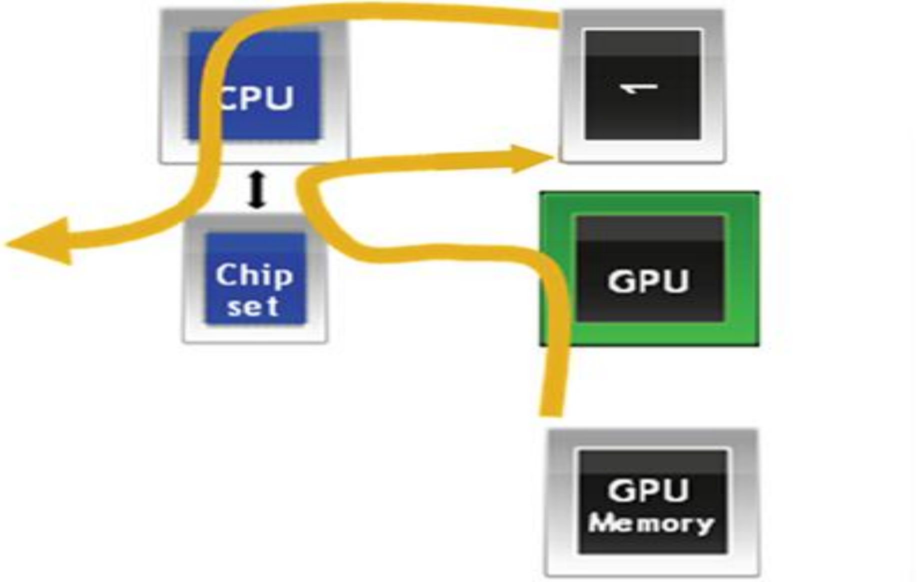
# GPUDirect for GPU cluster

## MPI Communication

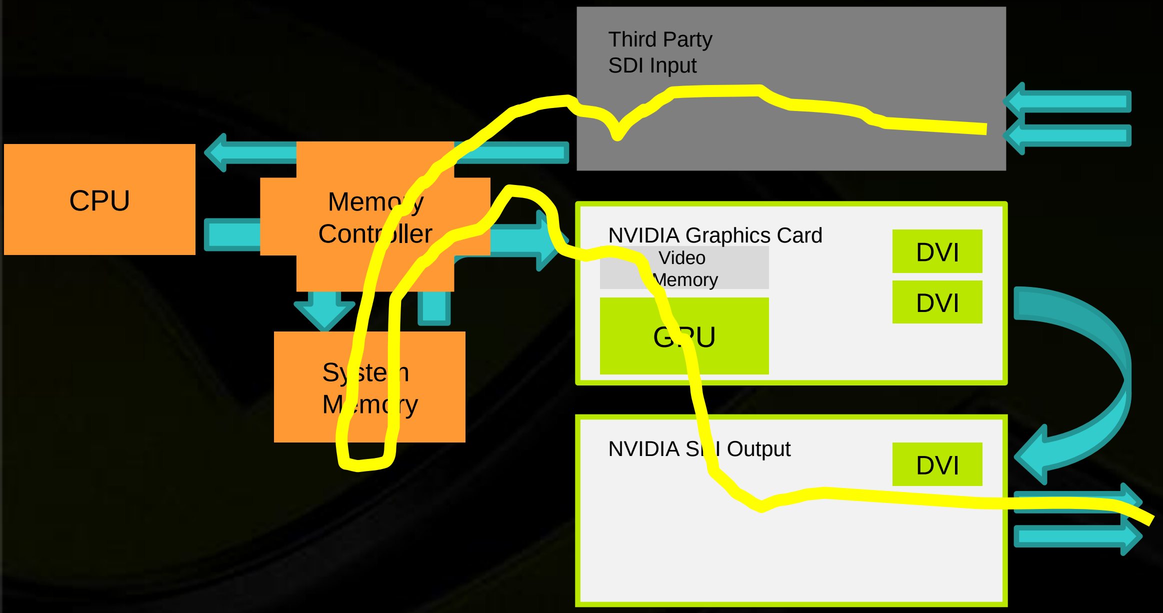
System Pinned Memory
System Pageable Memory



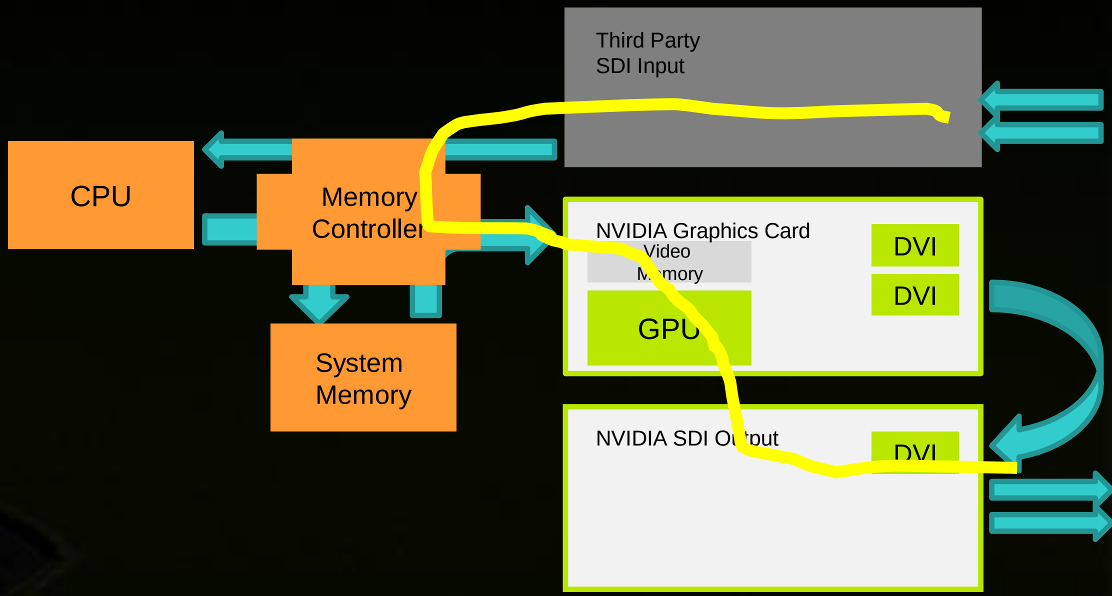

System Pinned Memory share



# FBO on SDI with Quadro




Write to Host memory and to write GPU memory



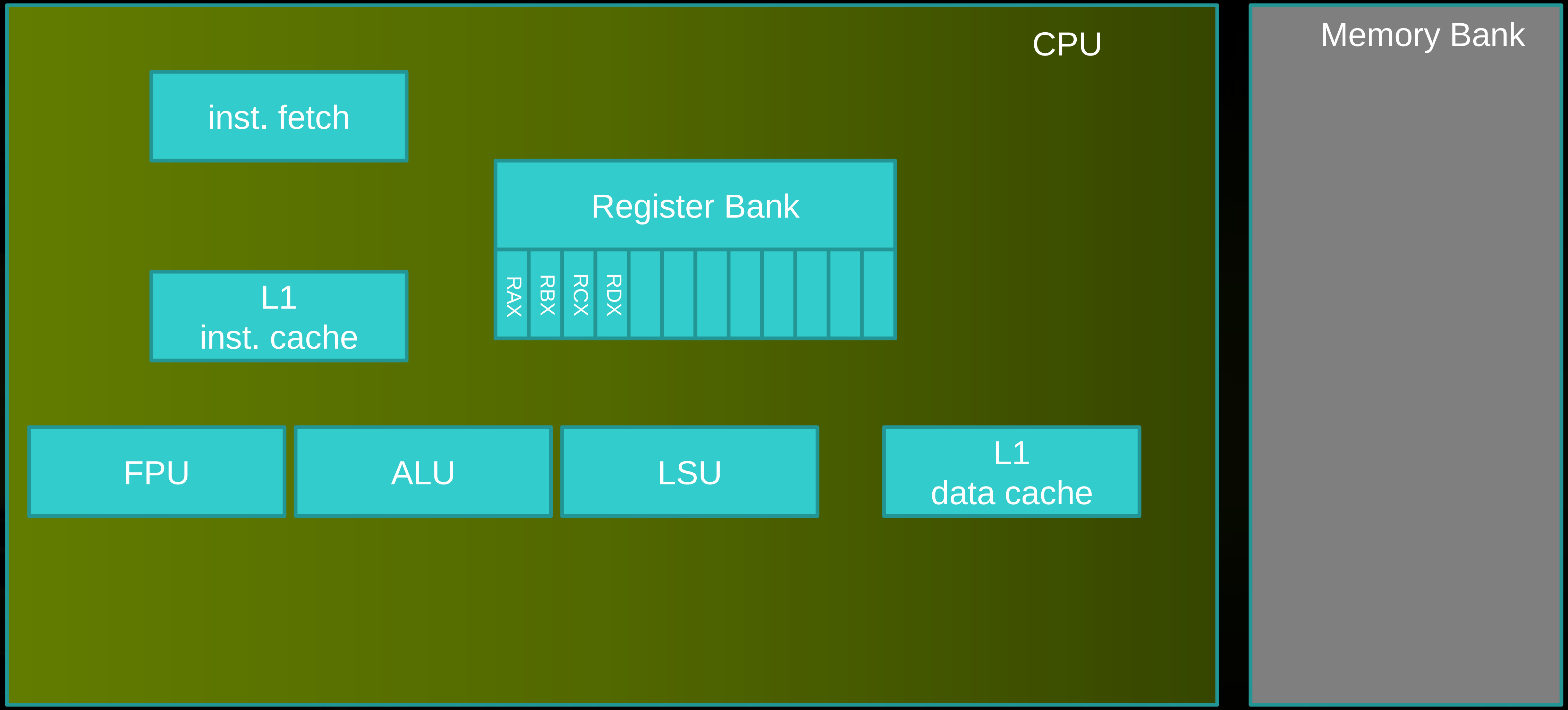

Direct write to OpenGL Frame Buffer Object

# Conceptual Tips for CUDA optimization

# SIMT architecture

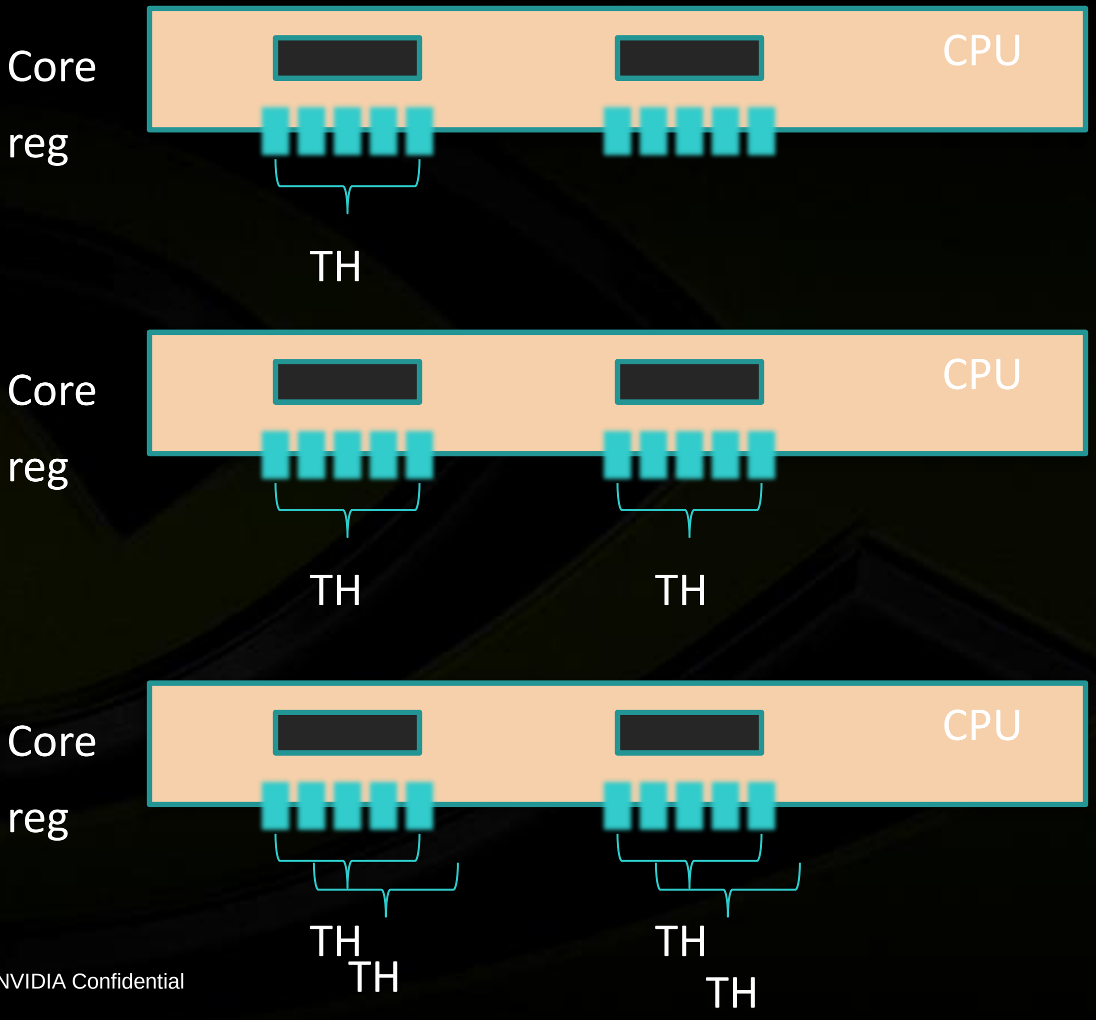
Single Instruction Multiple Threads

# Abstract overview of CPU Core



# Threads on Multicore CPU

**CPU**



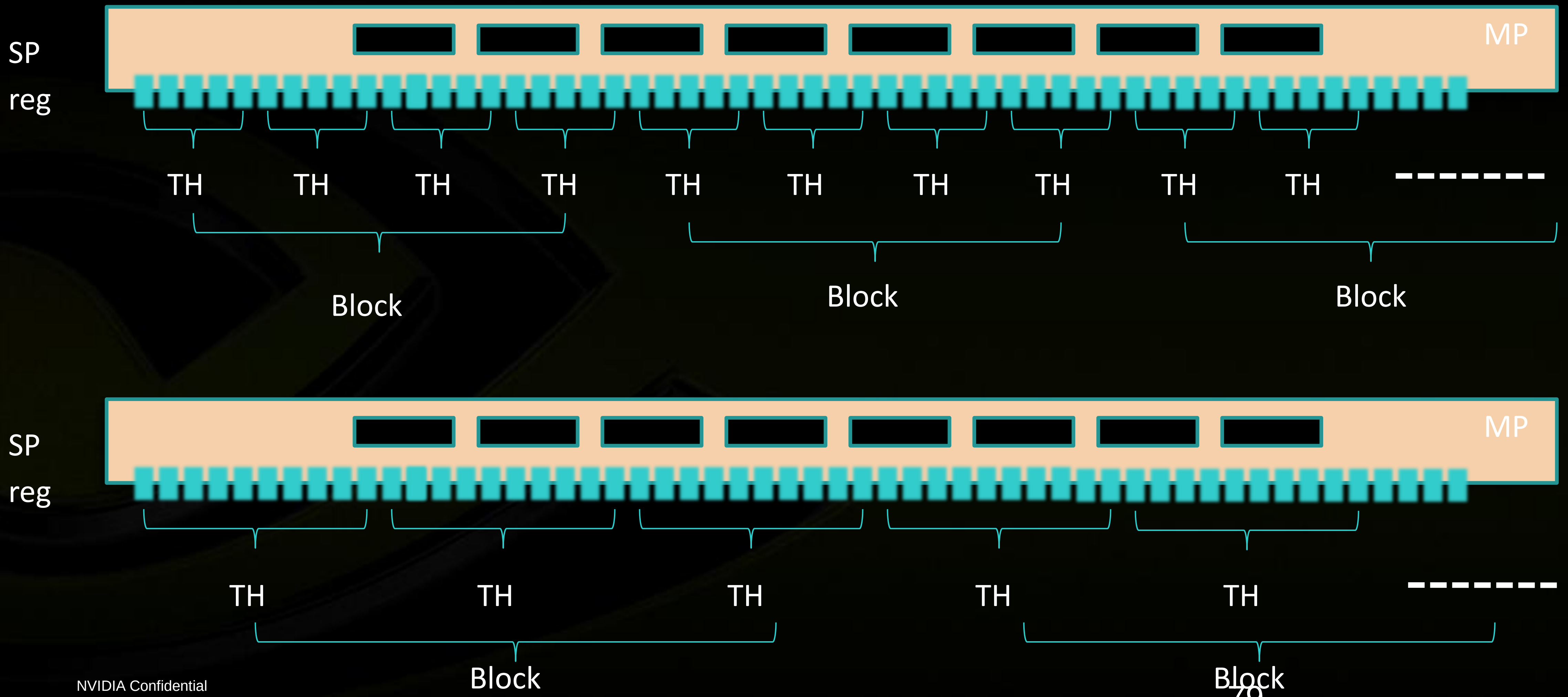

General Programming

Winthread, pthread

Hyper-threading

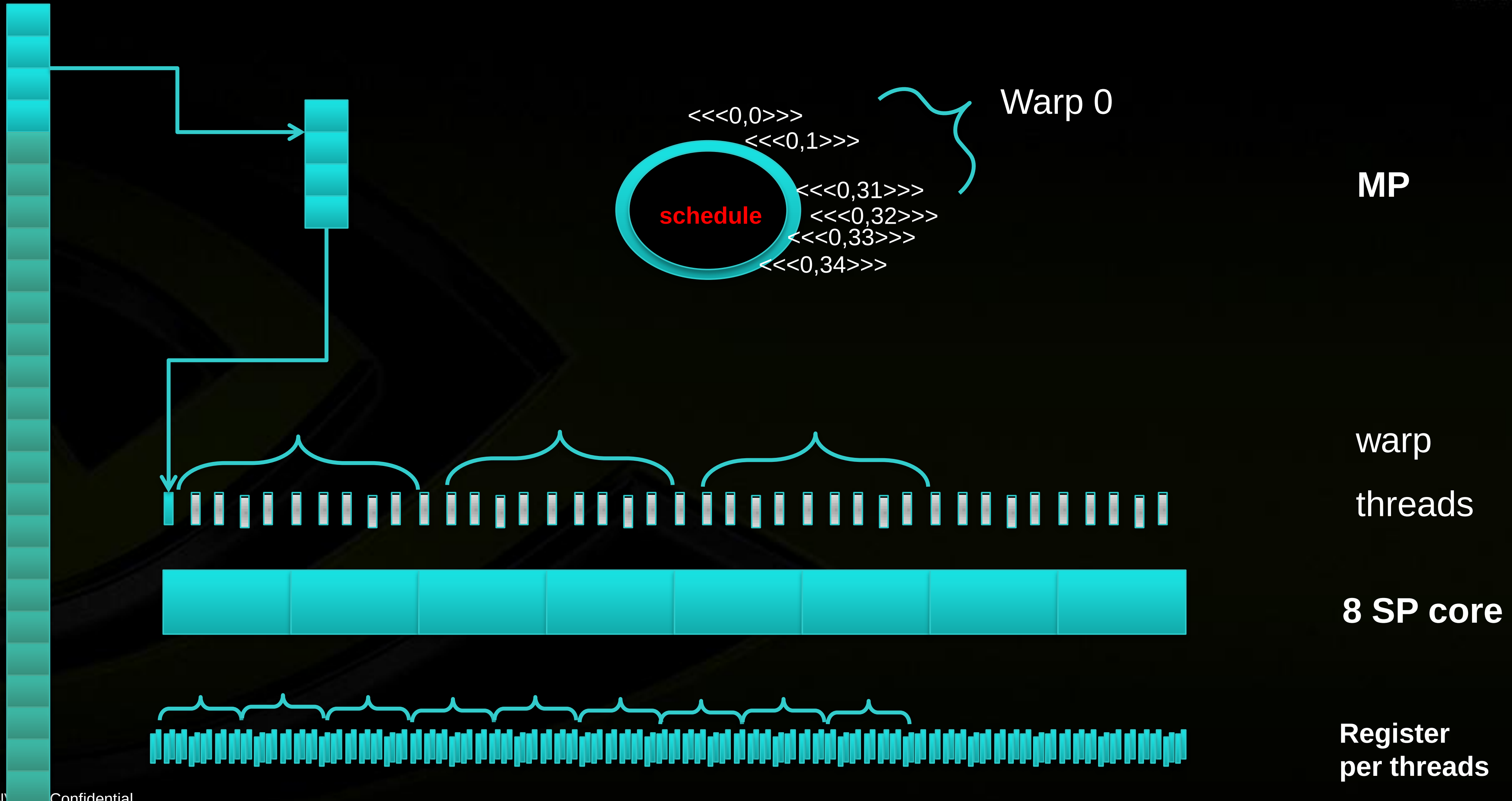
# Threads on Manicore GPU

H/W Multi Processor vs S/W Active Block

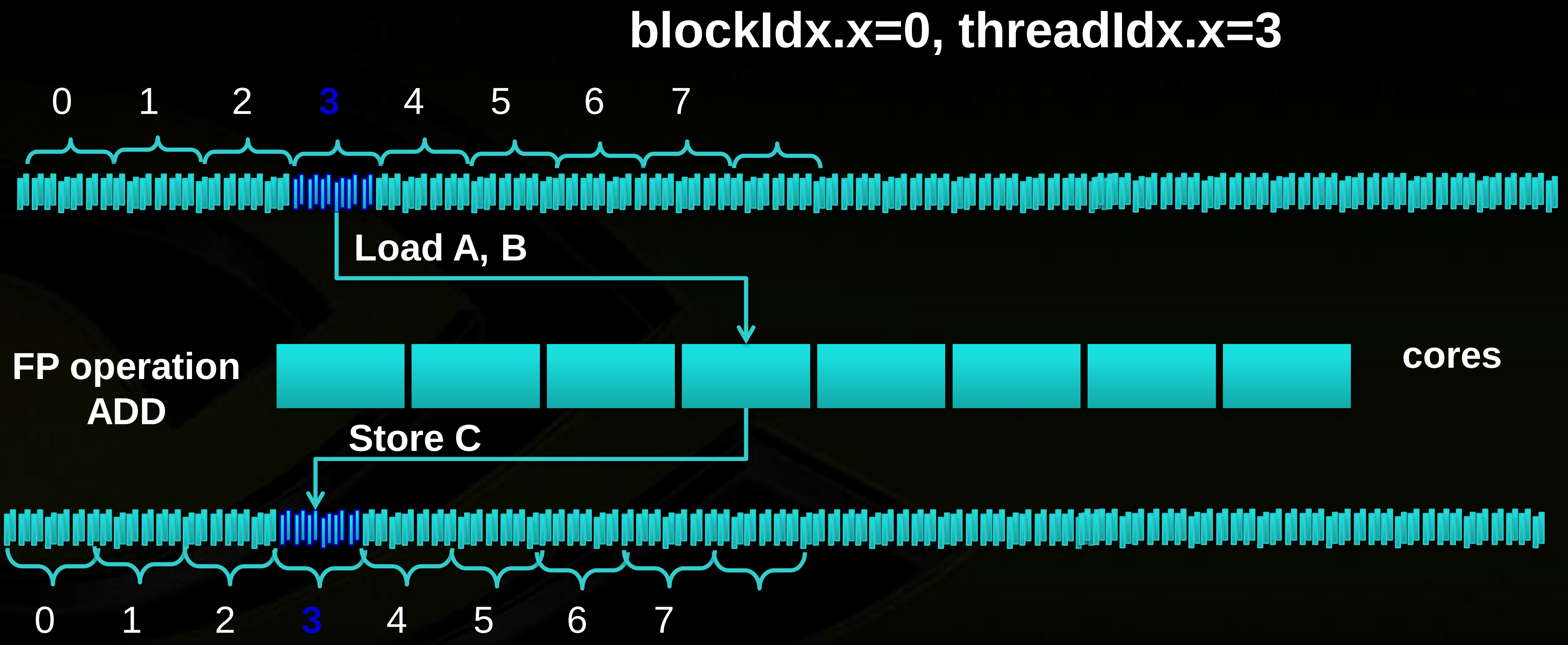



NVIDIA Confidential

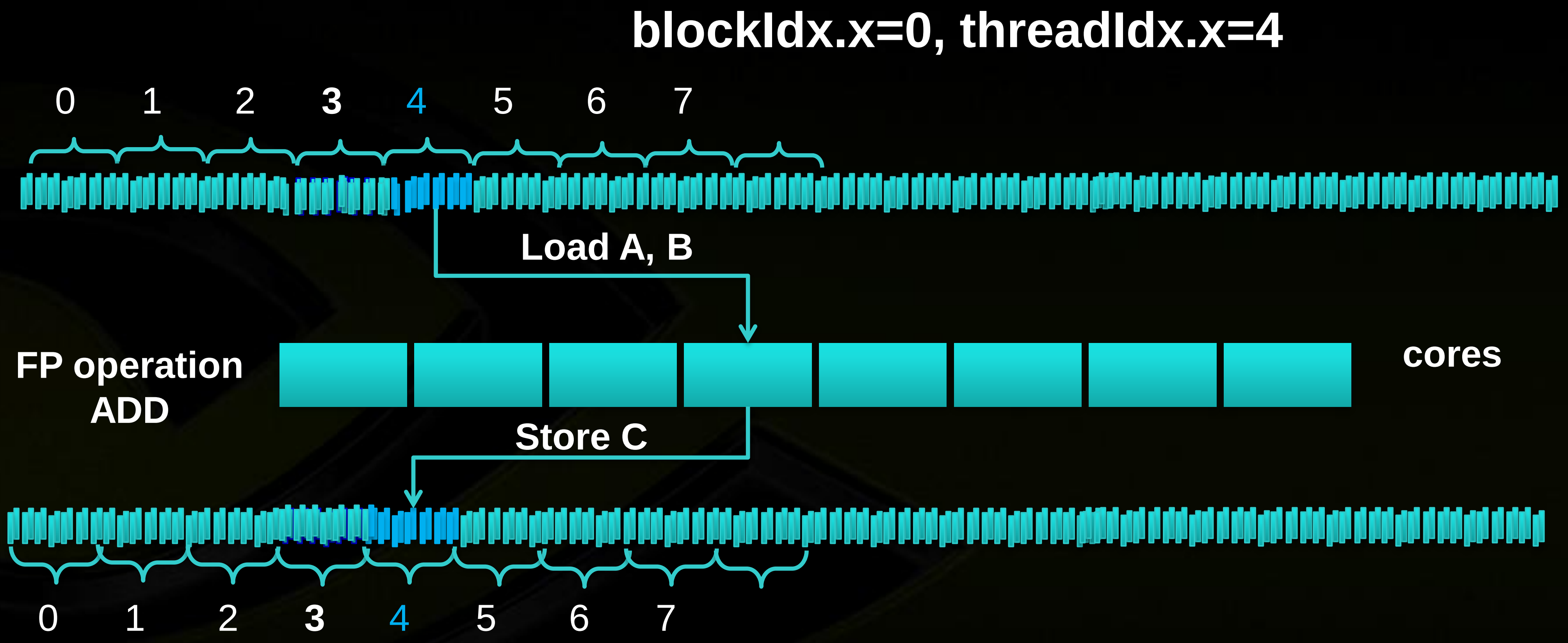
# Overview of WARP schedule



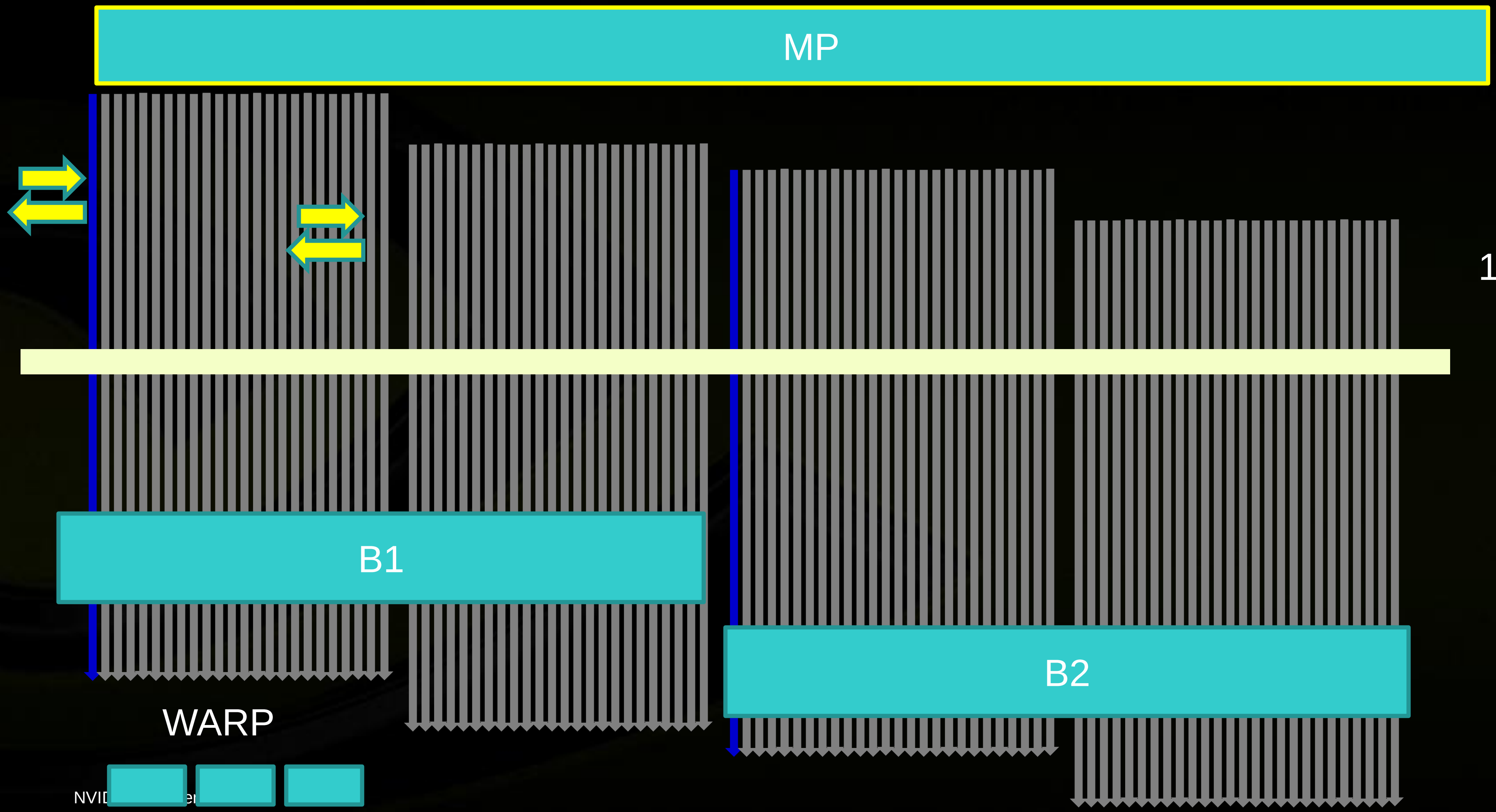
# Overview of Instruction Fetch



# Overview of Instruction Fetch



# Thread schedule within MP : WARP




1024 * 30 : 30K

**Look like Cocurrent Threads**

# Occupancy Calculator on CUDA SDK

## CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

| | | |
|---|---|---|
| 1.) Select Compute Capability (click): | 2.0 | (Help) |

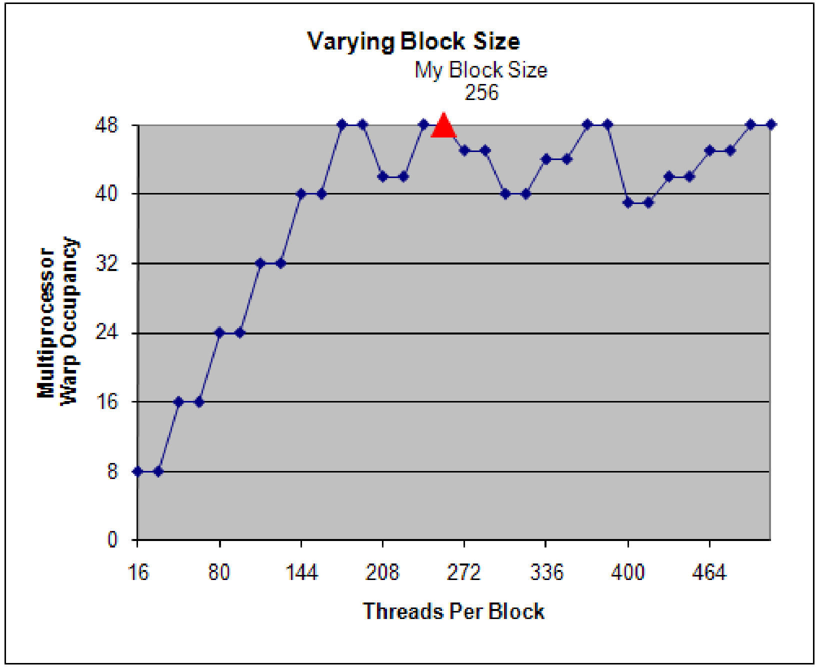
| 2.) Enter your resource usage: | | |
|---|---|---|
| Threads Per Block | 256 | (Help) |
| Registers Per Thread | 8 | |
| Shared Memory Per Block (bytes) | 1024 | |

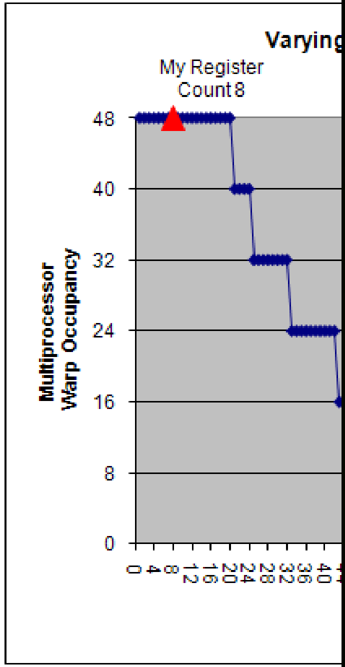
(Don't edit anything below this line)

| 3.) GPU Occupancy Data is displayed here and in the graphs: | | |
|---|---|---|
| Active Threads per Multiprocessor | 1536 | (Help) |
| Active Warps per Multiprocessor | 48 | |
| Active Thread Blocks per Multiprocessor | 6 | |
| Occupancy of each Multiprocessor | 100% | |

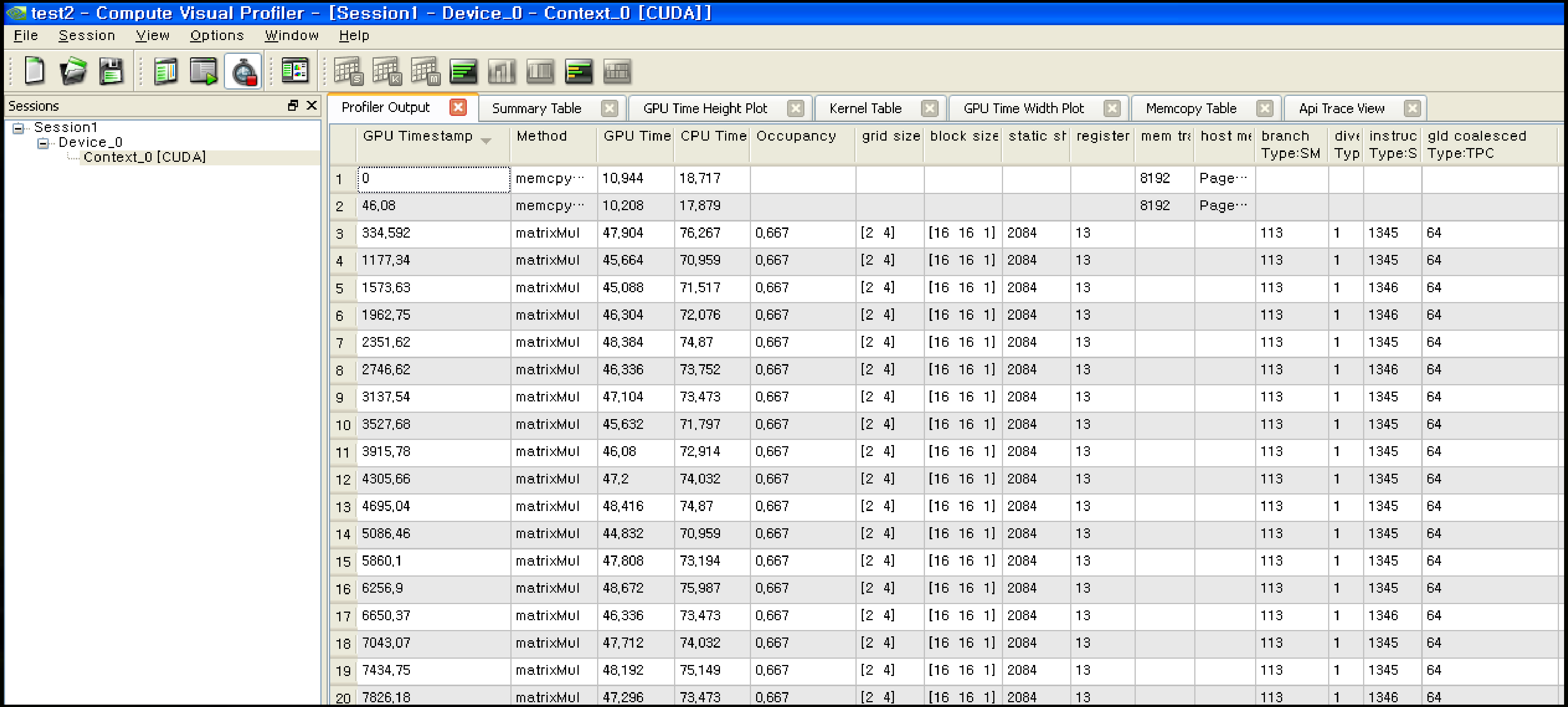
| Physical Limits for GPU Compute Capability: | 2.0 |
|---|---|
| Threads per Warp | 32 |
| Warps per Multiprocessor | 48 |
| Threads per Multiprocessor | 1536 |
| Thread Blocks per Multiprocessor | 8 |
| Total # of 32-bit registers per Multiprocessor | 32768 |
| Register allocation unit size | 64 |
| Register allocation granularity | warp |
| Shared Memory per Multiprocessor (bytes) | 49152 |

Your chosen resource usage is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared mem



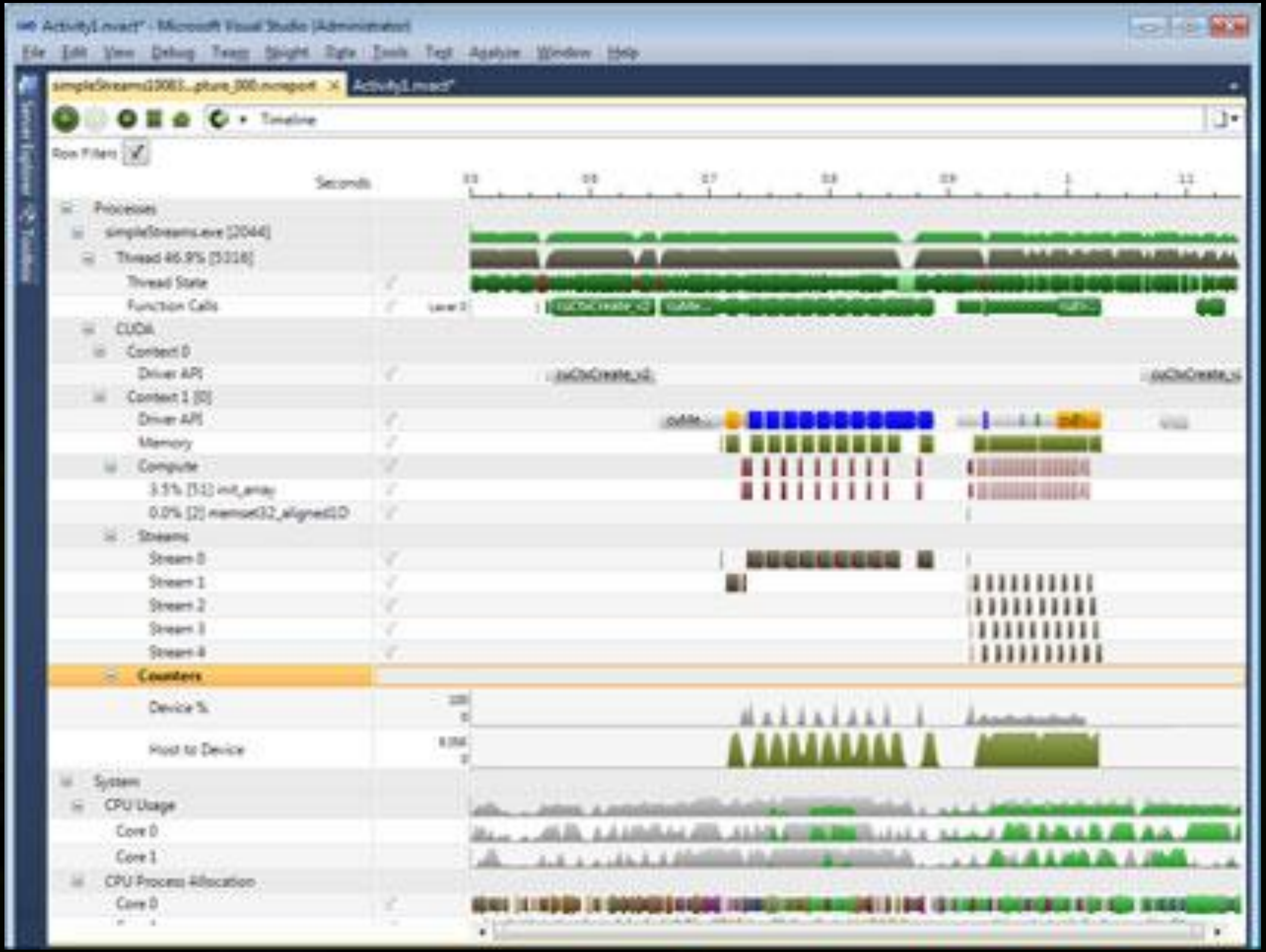



# CUDA profiler on CUDA toolkit



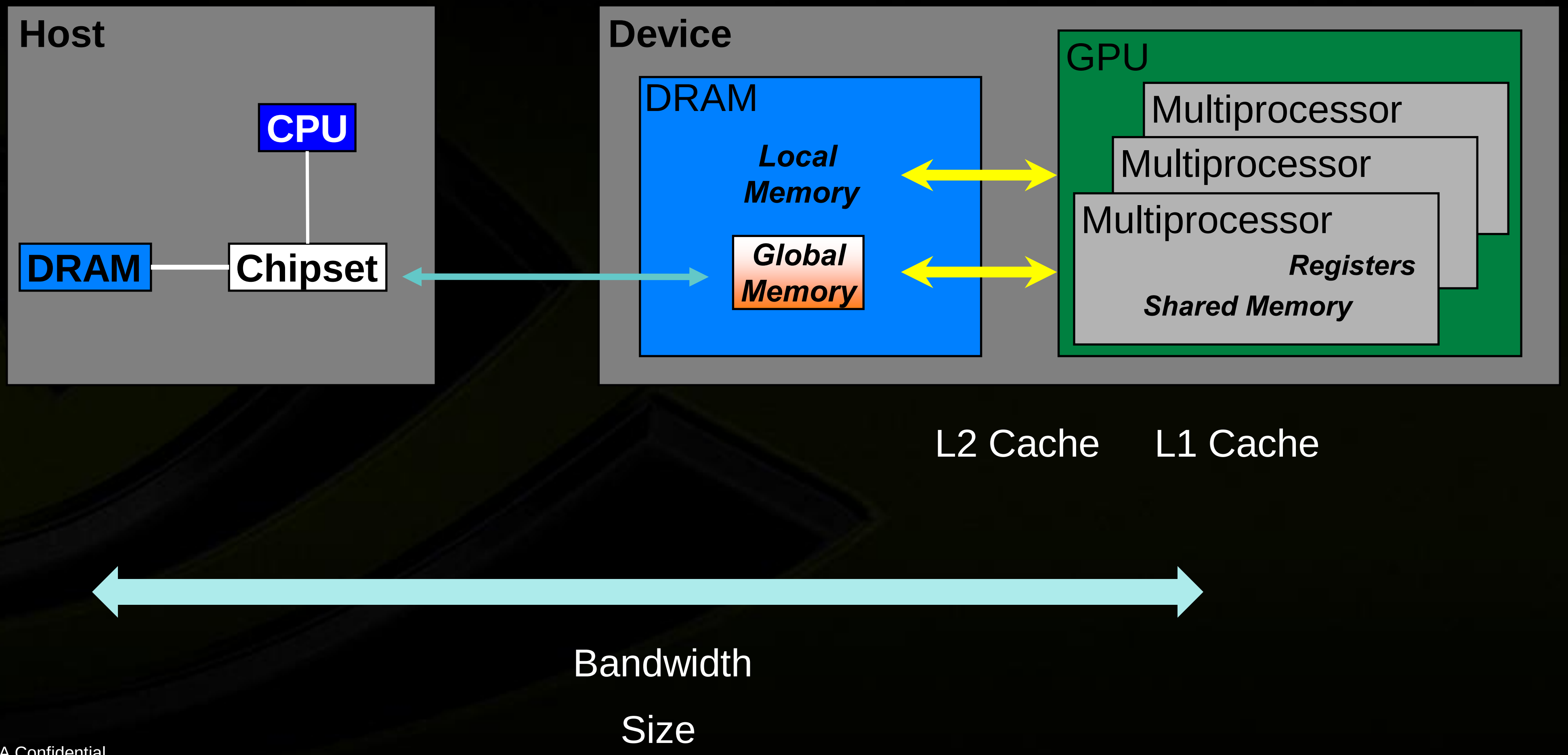

| | GPU Timestamp | Method | GPU Time | CPU Time | Occupancy | grid size | block size | static sh | register | mem tr | host me | branch Type:SM | dive Typ | instruc Type:S | gld coalesced Type:TPC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | memcpy… | 10,944 | 18,717 | | | | | | 8192 | Page… | | | | |
| 2 | 46,08 | memcpy… | 10,208 | 17,879 | | | | | | 8192 | Page… | | | | |
| 3 | 334,592 | matrixMul | 47,904 | 76,267 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1345 | 64 |
| 4 | 1177,34 | matrixMul | 45,664 | 70,959 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1345 | 64 |
| 5 | 1573,63 | matrixMul | 45,088 | 71,517 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1346 | 64 |
| 6 | 1962,75 | matrixMul | 46,304 | 72,076 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1346 | 64 |
| 7 | 2351,62 | matrixMul | 48,384 | 74,87 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1345 | 64 |
| 8 | 2746,62 | matrixMul | 46,336 | 73,752 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1346 | 64 |
| 9 | 3137,54 | matrixMul | 47,104 | 73,473 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1345 | 64 |
| 10 | 3527,68 | matrixMul | 45,632 | 71,797 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1345 | 64 |
| 11 | 3915,78 | matrixMul | 46,08 | 72,914 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1345 | 64 |
| 12 | 4305,66 | matrixMul | 47,2 | 74,032 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1345 | 64 |
| 13 | 4695,04 | matrixMul | 48,416 | 74,87 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1345 | 64 |
| 14 | 5086,46 | matrixMul | 44,832 | 70,959 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1345 | 64 |
| 15 | 5860,1 | matrixMul | 47,808 | 73,194 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1345 | 64 |
| 16 | 6256,9 | matrixMul | 48,672 | 75,987 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1345 | 64 |
| 17 | 6650,37 | matrixMul | 46,336 | 73,473 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1346 | 64 |
| 18 | 7043,07 | matrixMul | 47,712 | 74,032 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1345 | 64 |
| 19 | 7434,75 | matrixMul | 48,192 | 75,149 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1345 | 64 |
| 20 | 7826,18 | matrixMul | 47,296 | 73,473 | 0,667 | [2 4] | [16 16 1] | 2084 | 13 | | | 113 | 1 | 1346 | 64 |

# Parallel NSight 1.5 Professional

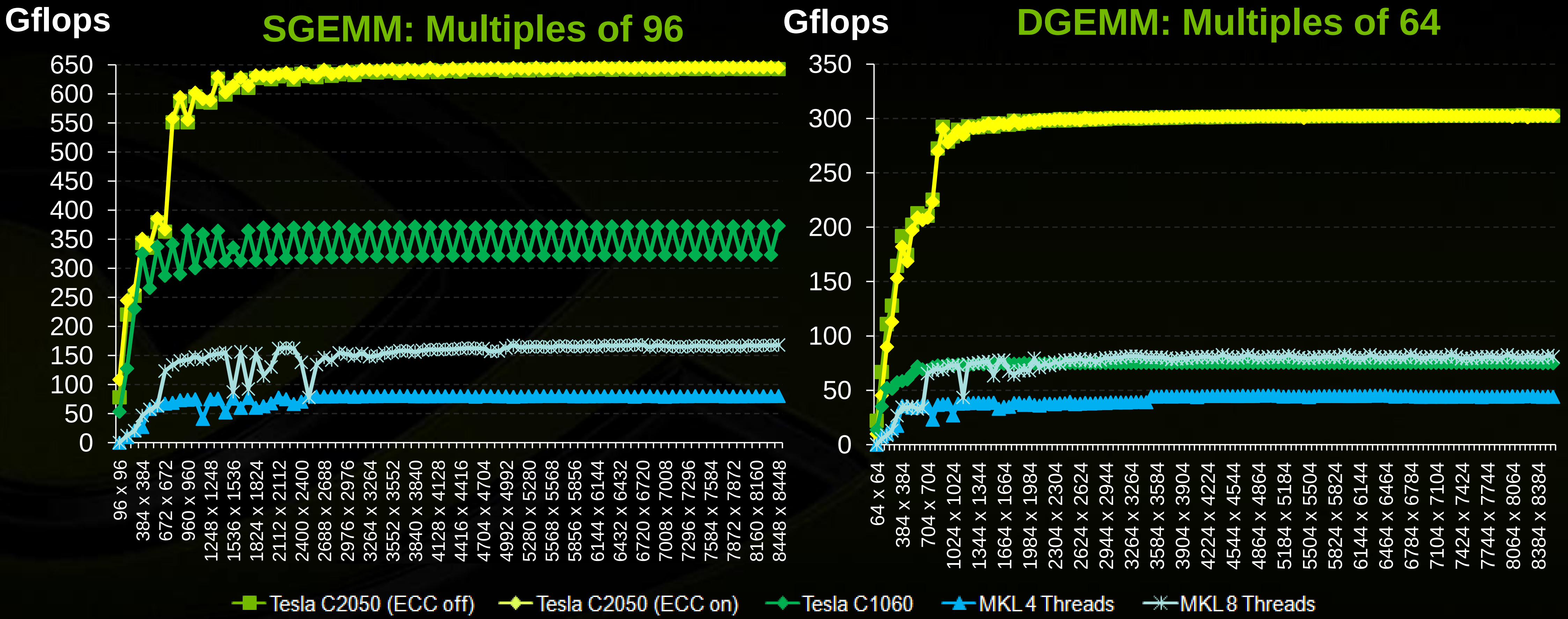


# Memory Hierarchy

# Managing Memory



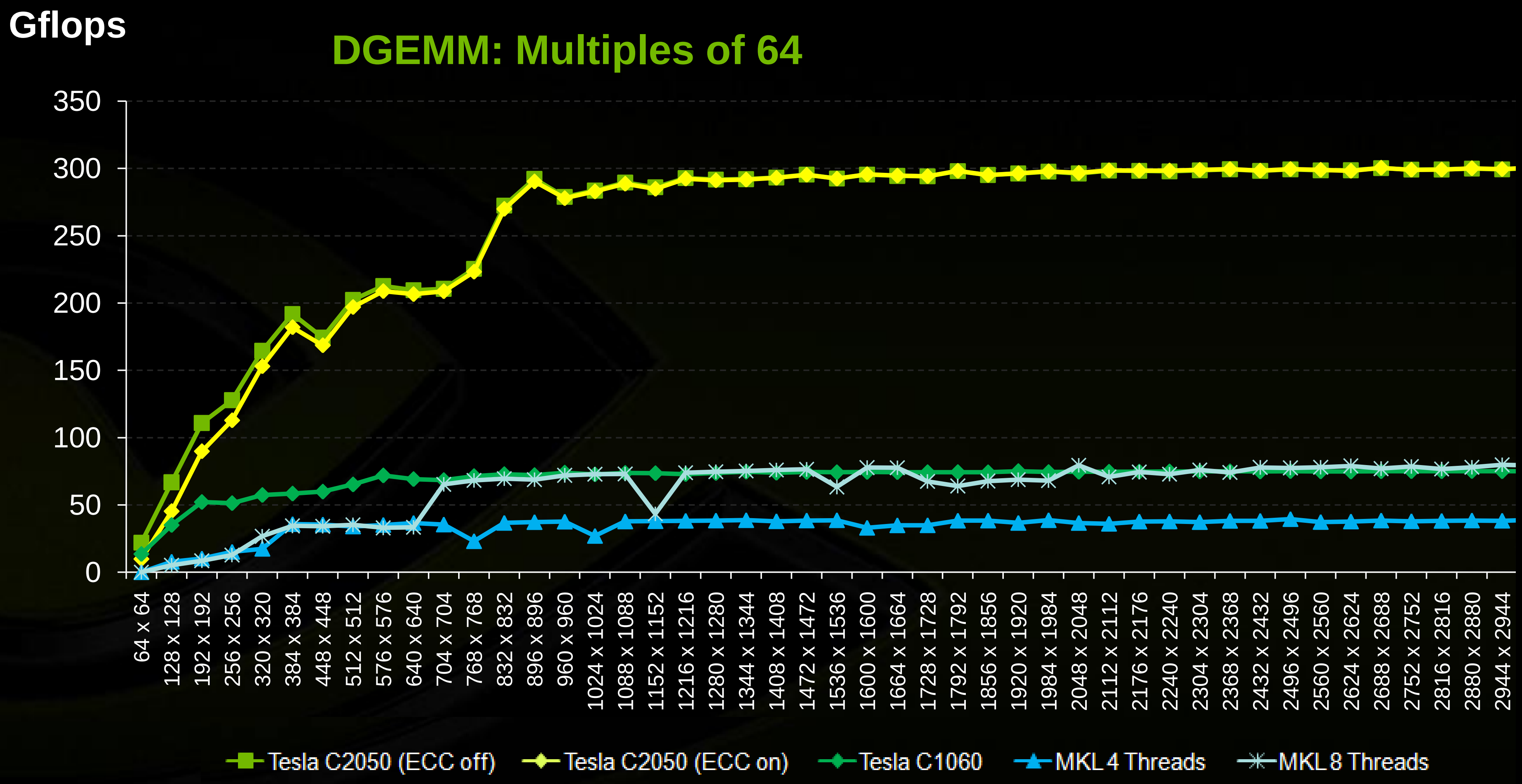
# Matrix Size for Best CUBLAS3.2 Performance




**cuBLAS 3.2: NVIDIA Tesla C1060, Tesla C2050 (Fermi)**

**MKL 10.2.4.32: Quad-Core Intel Xeon 5550, 2.67 GHz**

# cuBLAS level III

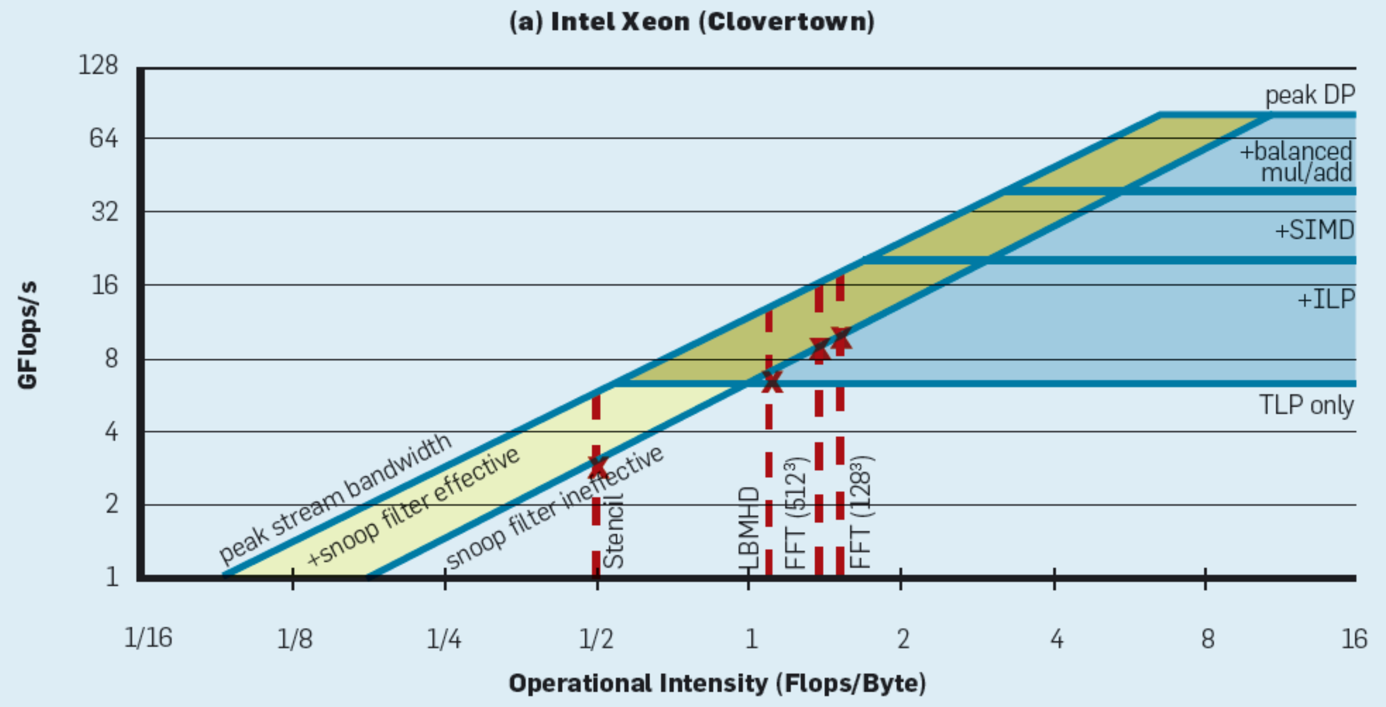



cuBLAS 3.2: NVIDIA Tesla C1060, Tesla C2050 (Fermi)

MKL 10.2.4.32: Quad-Core Intel Xeon 5550, 2.67 GHz

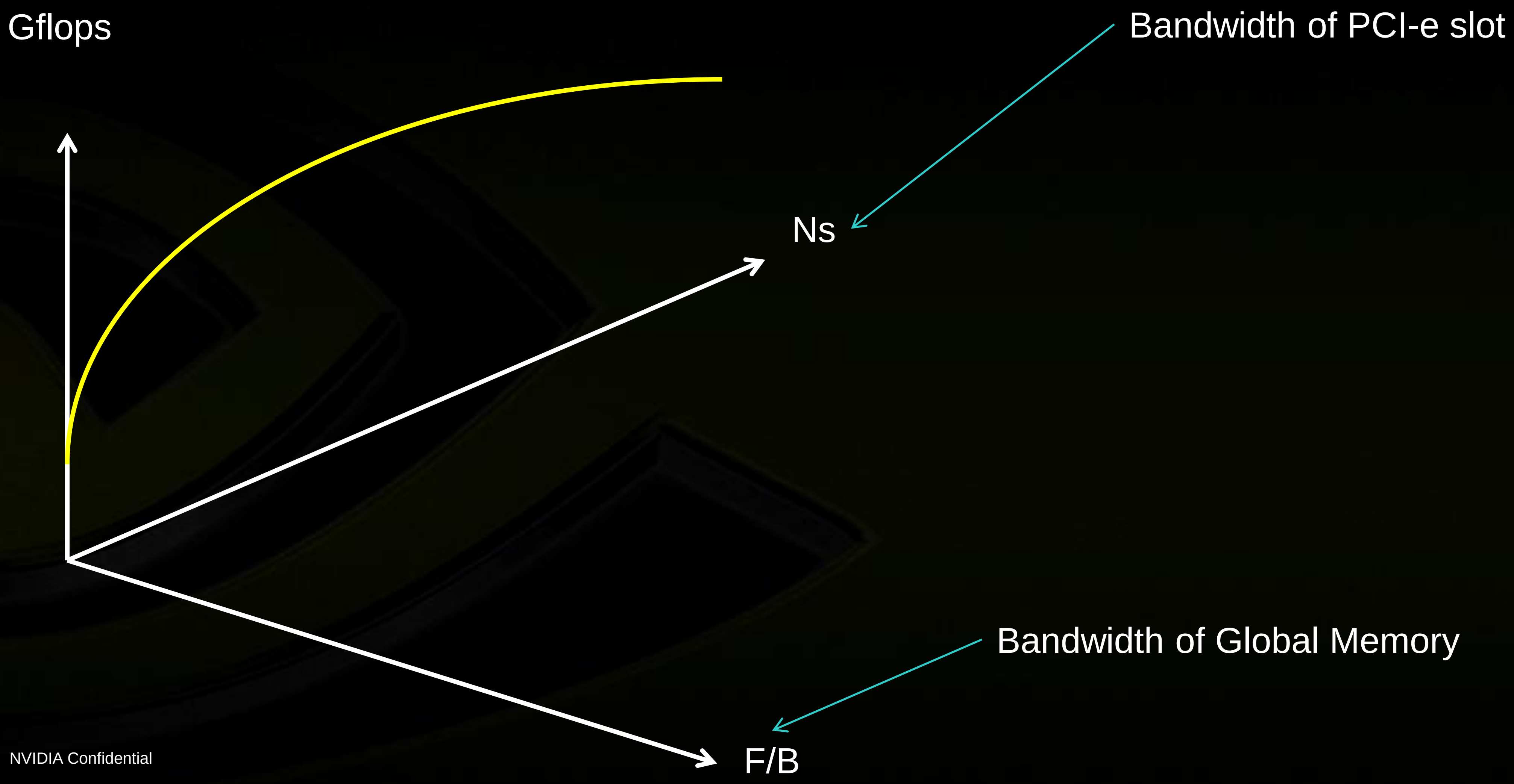
# Roofline Analysis (Arithmetic Intensity)

**Figure 3a–3c: Roofline model for Intel Xeon, AMD Opteron X4, and IBM Cell.**




Samuel Williams, Andrew Waterman, and Dav id Patterson,
Roofline: An Insightful Visual Performance Model for Multicore Architectures

# Perf. on CUDA Application



# Tips for Optimization

- Consider Algorithm for parallel (naïve algorithms will be good)
- Consider Occupancy ( SIMT)
- Consider Memory Bottleneck

# More Information for CUDA Optimization

- CUDA Zone
  http://www.nvidia.com/CUDA

- Developer Zone
  http://developer.nvidia.com

- GTC 2010 contents
  http://www.nvidia.com/gtc2010-content

- 쿠다 카페 (CUDA café in Korea)
  http://cafe.daum.net/KCUG

# Thanks

Hyungon Ryu
hryu@nvidia.com