Dyalog '09
Princeton, New Jersey

APL and CUDA
A Pioneering Approach to Parallel Array Processing in Quantitative and Mathematical Finance

Yigal Jhirad and Blay Tarnoff
September 16, 2009

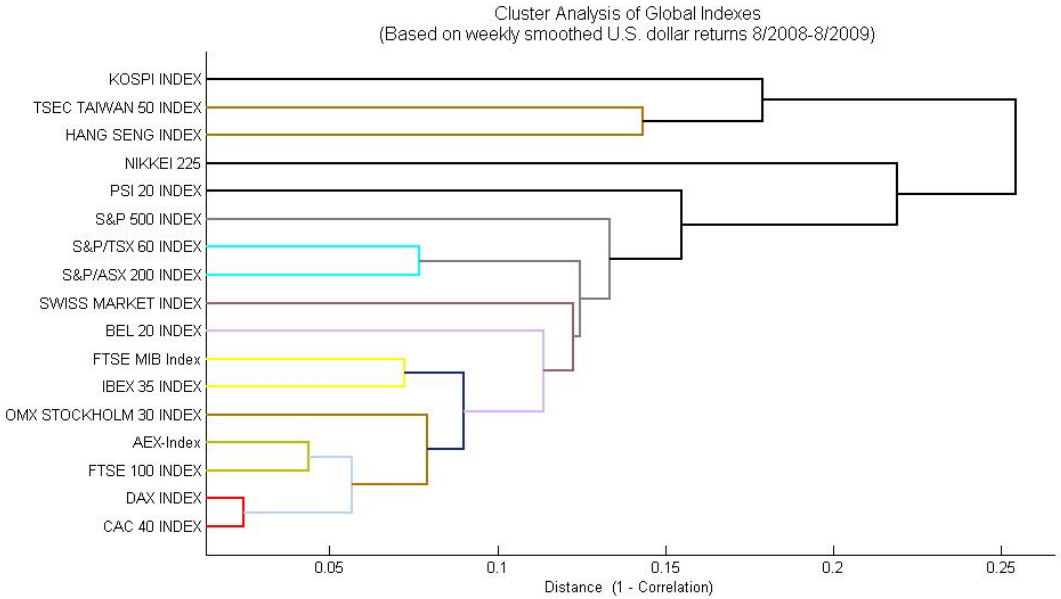
APL/CUDA: Table of Contents

APL/CUDA: The Need For Speed

- **Portfolio Management, Analytics, and Trade Execution demand increasing amounts of computing speed**
  - Must Have
  - High Frequency Trading
  - Risk Management, Market Impact
- **APL and NVIDIA GPU physics based modeling exploit hardware efficiently**
  - Array Based Processing paradigm Matrix/Vector thought process is key
  - APL leverages CPU's ability and provides for rapid development of software applications
  - CUDA leverages GPU Hardware
- **Application in Econometrics and Applied Mathematics**
  - Monte Carlo Simulations
  - Principal Components
  - Cluster Analysis
  - Fourier Analysis
  - Optimization
  - Cointegration
- **Neural Networks**
  - Rapid application development and testing of idea thesis and innovation

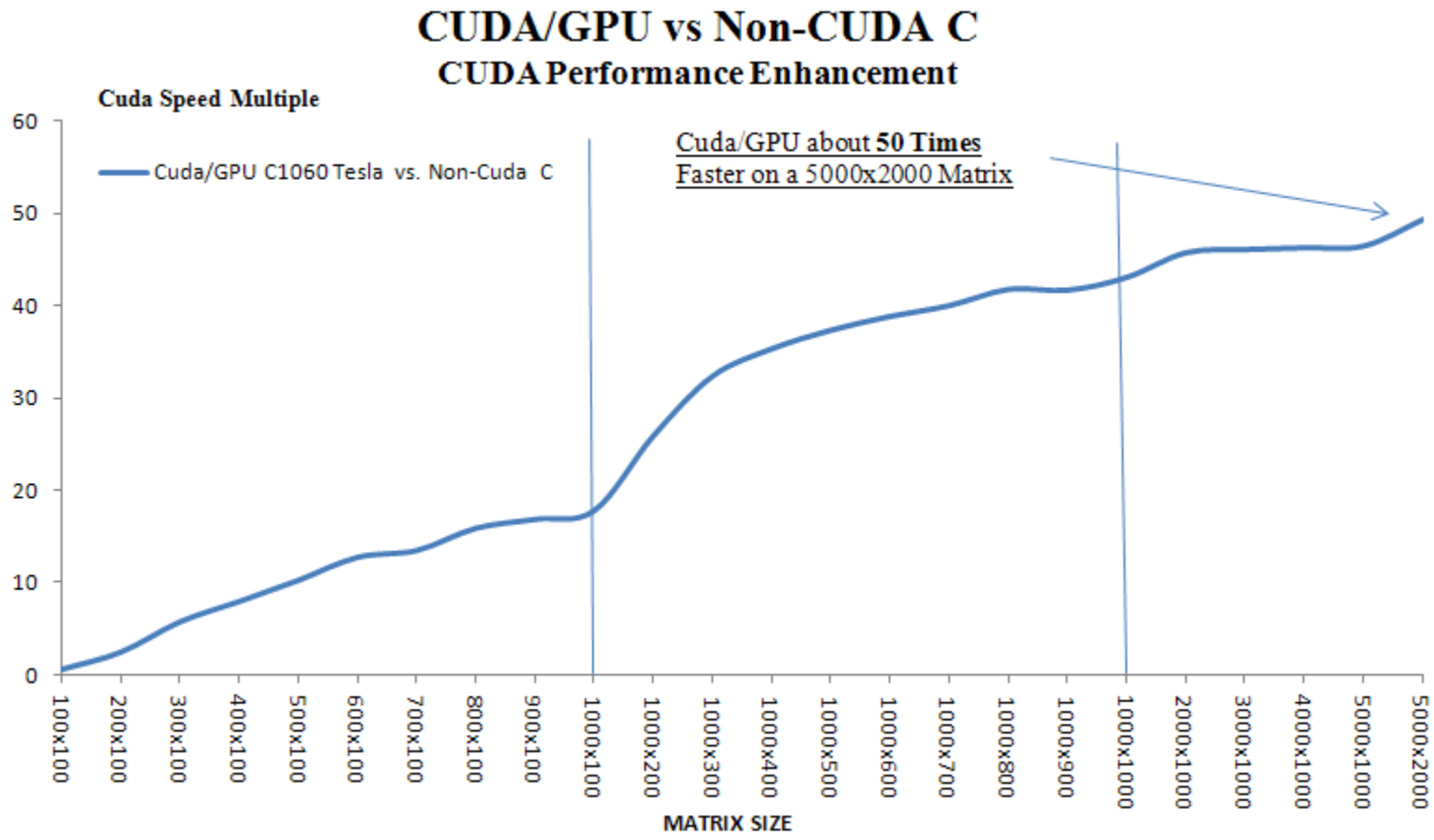
APL/CUDA: Example - Cluster Analysis

- **Cluster Analysis: A multivariate technique designed to identify relationships and cohesion**
  - Factor Analysis, Risk Model Development
- **Correlation Analysis: Pairwise analysis of data across assets. Each pairwise comparison can be run in parallel.**
  - Use Correlation as primary input to cluster analysis
  - Apply proprietary signal filter to remove selected data and reduce spurious correlations



APL/CUDA: Speed Enhancements

- **Speed Enhancement of CUDA compared to C without CUDA is significant as the matrix size gets larger**



Windows XP 64. Dual Xeon 3.2 GHz. Applications run from Dyalog Apl 64-Bit through a DLL interface.

APL/CUDA: Application – Correlation Kernel

Application example – Correlation function removing N/A values

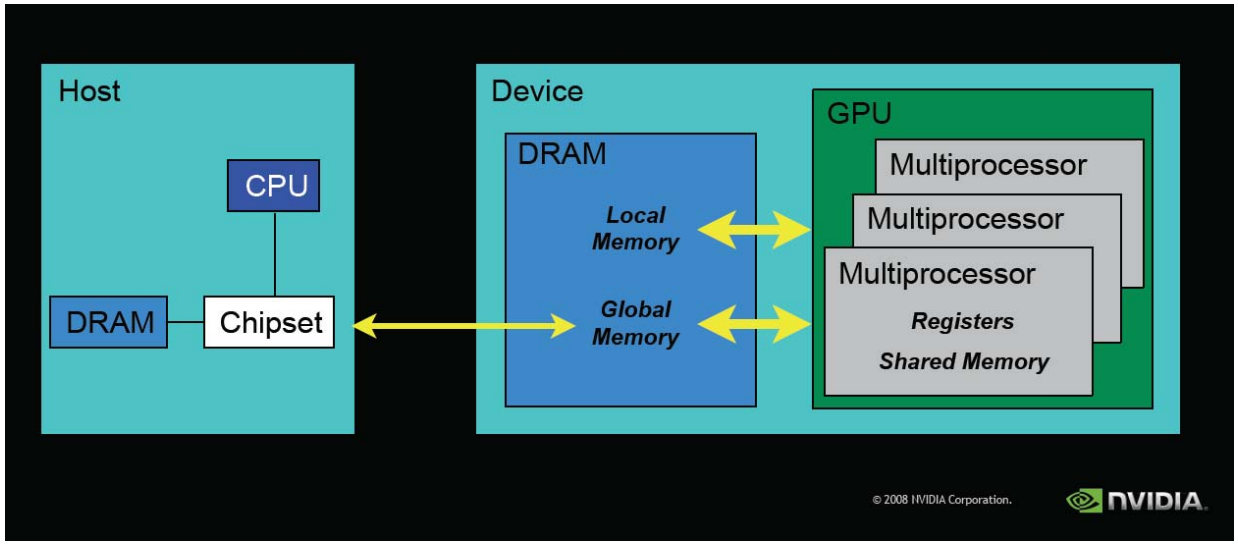
- Correlation measures the direction and strength of a linear relationship between variables. The Pearson product moment correlation between two variables X and Y is calculated as:

$$\frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n(\sum x_i^2) - (\sum x_i)^2}\ \sqrt{n(\sum y_i^2) - (\sum y_i)^2}}$$

- For N assets there are $\frac{N \times (N-1)}{2}$ unique correlation pairs
- Given an N x M matrix A in which each row is a list of returns for a particular equity, return an N x N matrix R in which each element is the scalar result of correlating each row of A to every other
  - Each element of A may be an N/A value
  - When processing a pair of rows, the calculation must include neither each N/A value nor the corresponding element in the other row. This requires evaluating each pair separately.
  - As a result the increased computational demand is more effectively implemented through a parallel processing solution. As the matrix size increases the benefits of parallel processing become more significant.

APL/CUDA: Compute Unified Device Architecture

- Host (CPU) initiates program
- Copies data from host memory to device memory
- Launches Kernels on Graphical Processing Unit (GPU)
- Copies result data from device memory back into host memory



Source: NVIDIA

APL/CUDA: Host/GPU Program Structure

Code

- Host writes N x M matrix A to GPU global memory
- Host launches N x (ceil(N/16)) grid of thread blocks
  - Each thread block is 16 x 16 threads
  - Each row of the thread block processes a single pair of rows of A
    - Each row of the thread block reads 16 contiguous elements of each of the pair of rows of A simultaneously (twice) using coalescing
    - Each row of the thread block processes 16 elements at a time in a loop
      - Each row of the thread block loops ceil(M/16) times
      - Each thread processes every 16^{th} column of A
      - Each row of the thread block reads and writes to 16 contiguous elements of shared memory with no bank conflicts
  - Each thread block yields 16 scalar results (one per row) which it writes to 16 contiguous elements of R simultaneously using coalescing
- Host reads N x N matrix R from GPU global memory

APL/CUDA: Summary

- **APL and CUDA provide powerful capabilities in Quantitative Finance**
  - Powerful synergies in array processing provide an effective framework for amplifying the impact of APL with CUDA
  - Integrating APL with CUDA provides a powerful software and hardware platform with which to drive applications that demand computing power and speed
- **Application in Investment Management**
  - Alpha Generation and Hedging Strategies
  - Trading and Execution
  - Performance Measurement and Risk Management

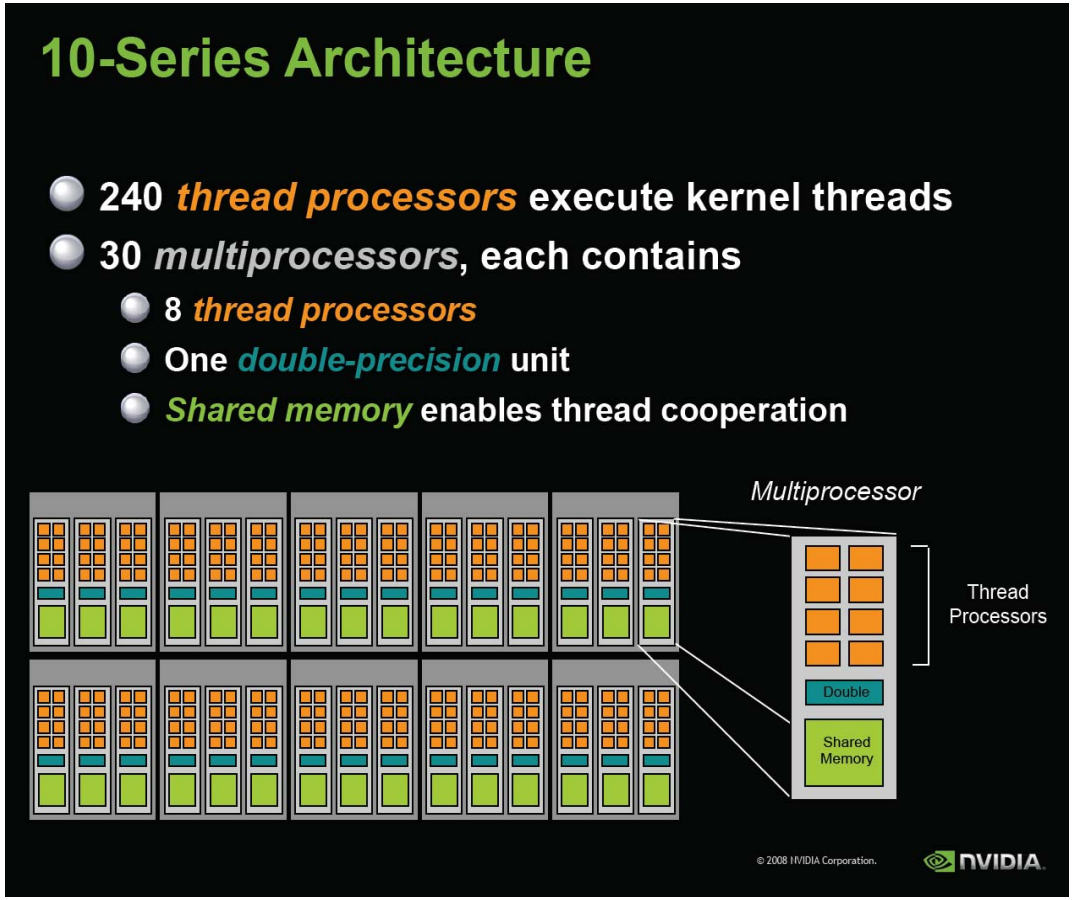
Author Biographies

- **Yigal D. Jhirad** is a senior vice president at Cohen and Steers. He is a portfolio manager and director of quantitative and derivatives strategies. Prior to joining Cohen and Steers, he was an executive director and head of portfolio and derivatives strategies at Morgan Stanley. He was responsible for developing, implementing and marketing quantitative and derivatives products to hedge funds, active and passive funds, pension funds and endowments.

- **Blay A. Tarnoff** is a senior project manager and applications developer. He specializes in array based languages and has developed equity and equity derivatives applications for program trading, proprietary trading, quantitative strategy and risk management. He was most recently a senior consultant at Morgan Stanley where he developed quantitative and real time trading and risk technology.

Appendix: Device Architecture

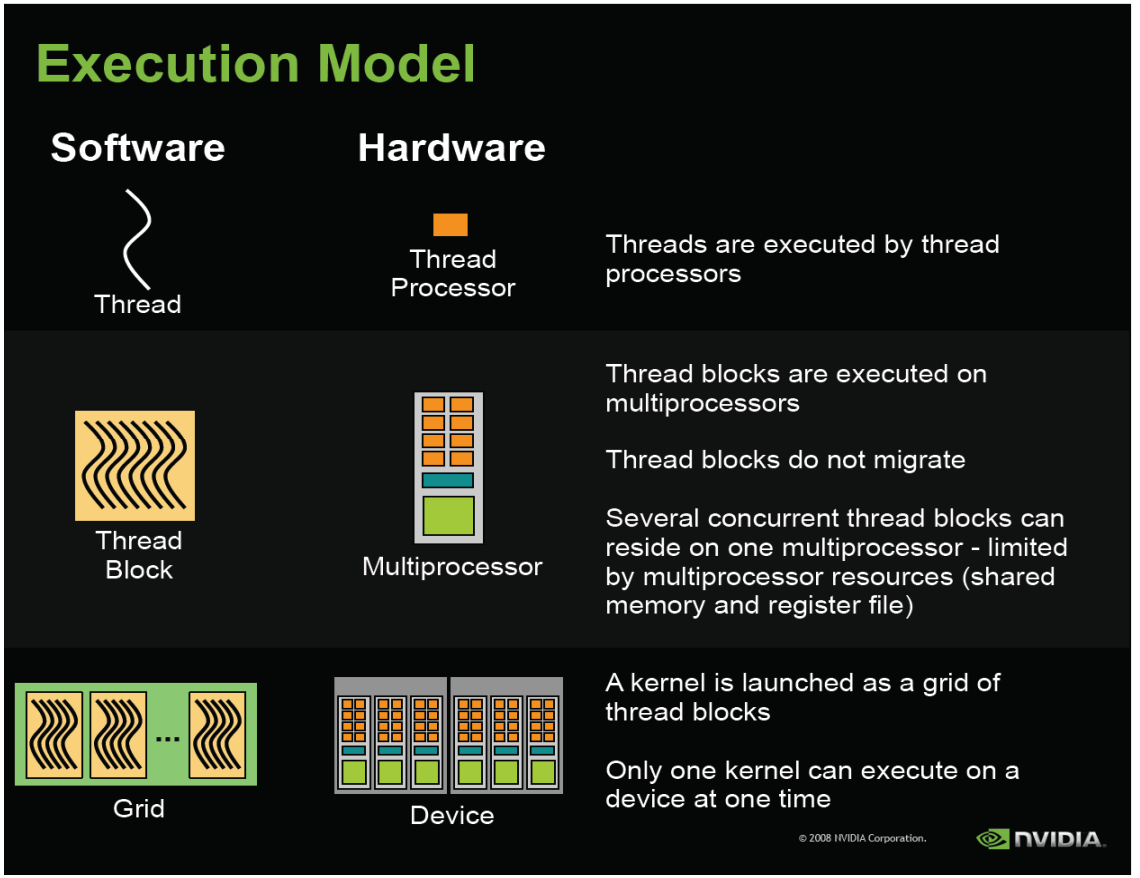
- Global, Constant, Texture Memory
- Graphical Processing Unit (GPU)
- Streaming Multiprocessors (SM)
  - Shared memory
  - Register memory
  - 8 Scalar Processors (SP)
  - Runs blocks of threads
  - Schedules Blocks/Warps



Source: NVIDIA

Appendix: Execution Model

- **Launch Kernel on GPU**
  - Threads run kernel Code in parallel
- **Thread Blocks are arrays of threads grouped into warps**
  - A warp is a group of 32 threads which execute simultaneously
  - Warps/blocks run in unpredictable order or interleaved
  - Thread Blocks Automatically scheduled by the GPU
  - Global vs. Shared memory



Source: NVIDIA

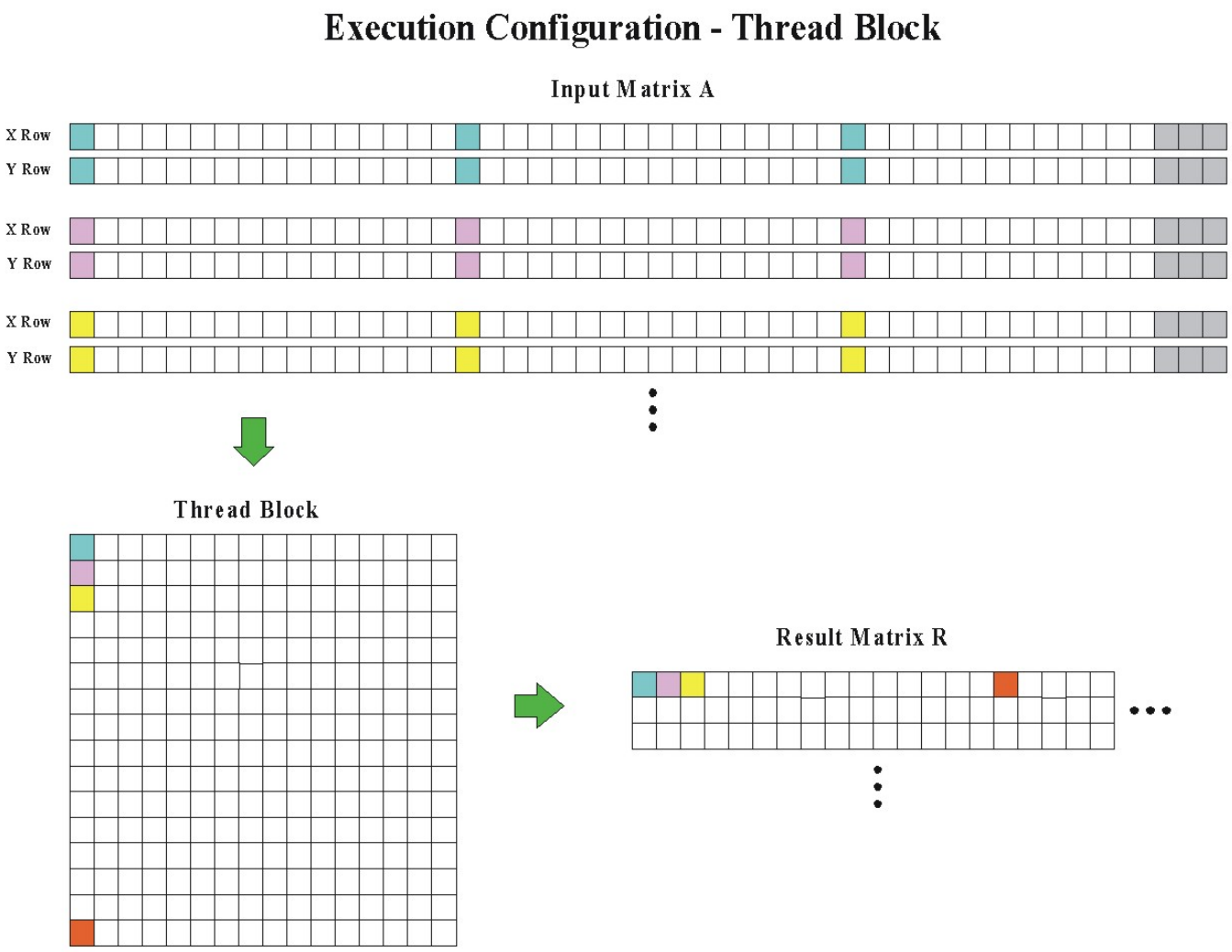
Appendix: Execution Configuration

- The problem must be broken down into logical blocks (thread blocks) that will all be handled as similarly as possible yet independently
- Host launches a very large one or two dimensional grid of **thread blocks**, which are automatically scheduled by the GPU
  - Blocks are small but many can be launched
  - Each block should be designed to be as independent as possible from all the others
  - Thread block is a small one, two, or three dimensional array of threads, grouped into **warps**
  - Each block should be designed to be small enough in terms of the number of threads and memory it uses to fit several within the resources provided by each SM yet employ enough threads to take advantage of simultaneous processing (normally 64 – 256 threads)

Appendix: Warps

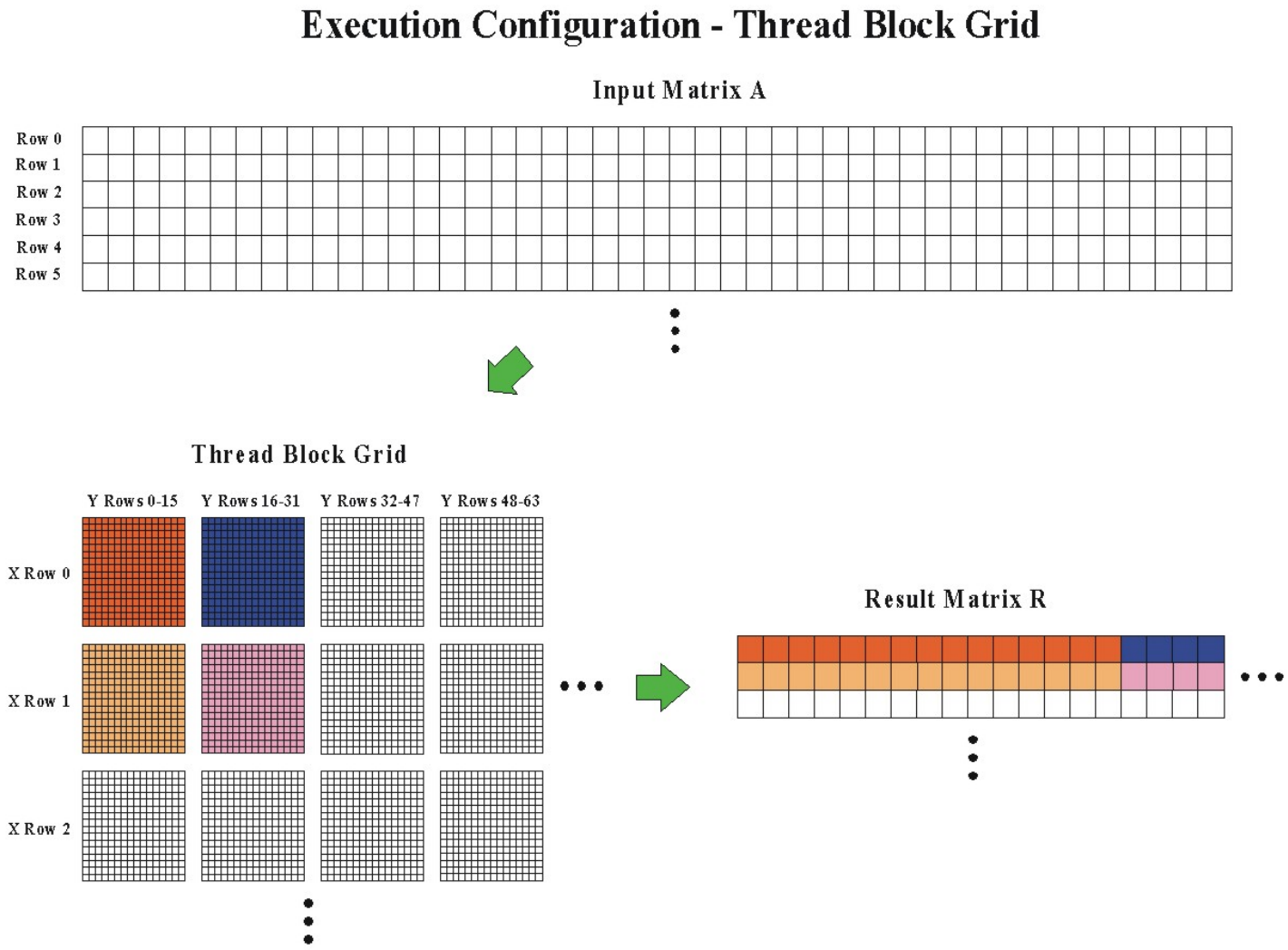
- **Warp is a group of 32 threads which all execute simultaneously by SIMT (Single Instruction, Multiple Thread) process**
  - Optimal to program warps to read and write to contiguous global memory locations to enable **coalescing**
    - Coalescing is the process of reading or writing 16 global memory elements simultaneously in a single instruction
  - Optimal to program warps to read and write to contiguous shared memory locations to eliminate **bank** conflicts
    - Banks are 16 divisions in shared memory to which data can be read or written simultaneously in a single instruction
  - Optimal to minimize **code divergence** within a warp
    - Code diverges when two threads in the same warp take different paths via program control statements (if, switch, for, etc.)

Appendix: Host and GPU Program Structure



Source: Blay Tarnoff

Appendix: Host and GPU Program Structure



Source: Blay Tarnoff

Appendix: Example of Parallel Thinking

Procedural language algorithm

Sum = 0
For (i = 0 to 15) Sum += Array[i]

Sum now contains sum of Array

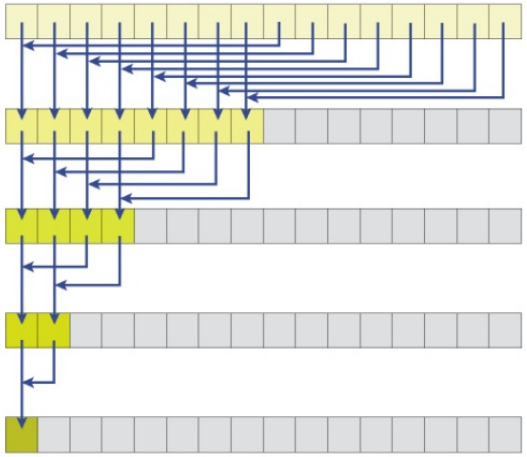
- This algorithm is inefficient in parallel computing because each iteration (read: thread) writes to the same memory location, causing a bottleneck necessitating 16 instructions to complete

Parallel algorithm

Array[current thread] += Array[current thread + 8]
Array[current thread] += Array[current thread + 4]
Array[current thread] += Array[current thread + 2]
Array[current thread] += Array[current thread + 1]

Array[0] now contains sum of Array

- This algorithm is more efficient because it writes to contiguous banks simultaneously and thus takes only 4 instructions to complete



Source: Blay Tarnoff