# Block-based GPU Programming with Triton – Phil Tillet
# Summary notes

**Introduction:**

- Motivation behind Triton: Addressing the flexibility vs simplicity trade-off in GPU programming.
- CUDA provides flexibility but at the cost of simplicity, leading to complex and hard-to-maintain codebases.
- Graph compilers (e.g., PyTorch) offer simplicity but lack the flexibility to represent certain ideas.

**Triton: A Middle Ground:**

- Triton aims to be simpler than CUDA but more expressive than graph compilers.
- Enables developers to write algorithms out-of-scope of graph compilers (e.g., radix sort, tree traversal).
- Code can be understood, modified, and written by researchers.
- Performance-portable across different GPU vendors.
- Performance can be at least as good as graph compilers, but not better than highly optimized CUDA code.

**Machine Model and Programming Model:**

- Triton operates on a standard von Neumann architecture with multi-core CPUs and GPUs.
- Users define tensors in the GPU's SRAM and modify them using PyTorch-like operators.
- Kernels are defined in Python using the triton.jit decorator.
- Supports pointer arithmetic and shape constraints (tensors must have power-of-two dimensions).

**Examples:**

- Vector Addition: Demonstrates basic element-wise operations, parallelization, and handling arbitrary input sizes.
- Softmax: Illustrates loading entire rows into SRAM for efficient computations.

**Why Triton is Powerful:**

- People Optimizations: The compiler can recognize patterns at the block level.
- SRAM Management: The compiler can analyze live ranges and manage SRAM efficiently.

- Automatic Vectorization: The compiler can vectorize operations based on tensor layouts.

**Advantages of Triton:**

- Offloads kernel development to researchers, reducing maintenance burden.
- Helps keep technical debt in check by maintaining a more manageable codebase.
- Allows for getting good performance quickly, striking a balance between development time and achievable performance.