Automatic High-Performance GPU code Generation using CUDA-CHiLL

THE UNIVERSITY OF UTAH™

USC Viterbi School of Engineering

Malik Khan (mmurtaza@usc.edu)
Jacqueline Chame (jchame@isi.edu)
Gabe Rudy, Chun Chen, Mary Hall, Mark Hall
{grudy,chunchen,mhall,markhall}@cs.utah.edu

Information Sciences Institute

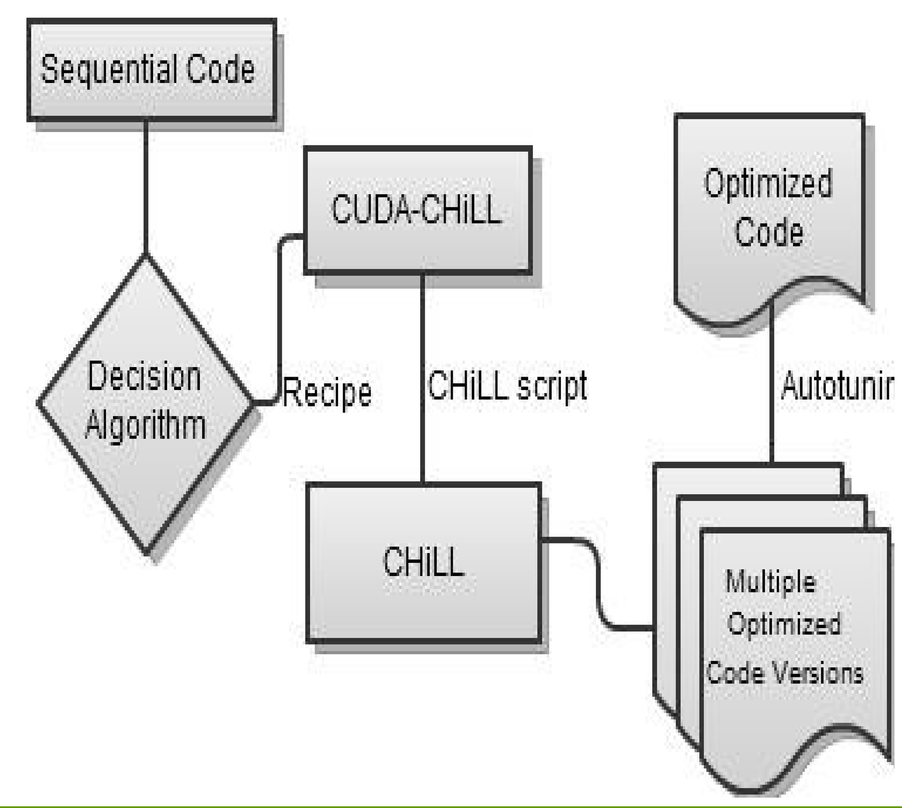
Funded by NSF award CSR-0615412, and DOE grant DE-FC02-06ER25765.

Motivation

- A Compiler-based Transformation, Code Generation and Auto-tuning System, which
  - Applies common compiler transformations.
  - Finds optimal computation mapping heuristics.
  - Searches highly-optimized code for a target GPU
- Goal: Achieve performance comparable to manually tuned code

CUDA-CHiLL

- Input: Sequential Loop nest computation
- Optimizing decisions
  - Computational decomposition
  - Data Staging
- Polyhedral framework
  - Transformation and code generation
- *Cudaize* transform
- Autotuning
- Output: CUDA code



CUDA-CHiLL Transformations

Command	Example Parameter	Description
tile_by_index	{"i","j"}	The index variables of the loops that will be tiled
	{TI,TJ}	The respective tile sizes for each index variable
	{l1_control="ii", l2_control="jj"}	A mapping that specifies control loop variable names and optionally renames tile loop index variables.
	{"ii", "jj", "i", "j"}	Final order of nested loops with updated loop index names
cudaize	"gpuMV"	The name of the kernel function
	{a=N, b=N, c=N*N}	The data sizes of the arrays if not statically determinable
	{block={"ii"}, thread={"jj"}}	Block and thread indices for mapping. The bounds for these loops are used to define the grid dimensions.
copy_to_registers	"kk"	The loop level, given as an index variable, that is the target of register structure
	"c"	The name of the array variable to be copied
copy_to_shared	"tx"	The loop level, given as an index variable, that is the target of the copied data
	"b"	The name of the array variable to be copied
	-16	Ensure the last dimension of the temporary array are coprime with 16
unroll_to_level	1	Unrolls all statements up to one level from innermost loops outwards. This construct will stop unrolling if it encounters a CUDA thread mapped index.

Decision Algorithm

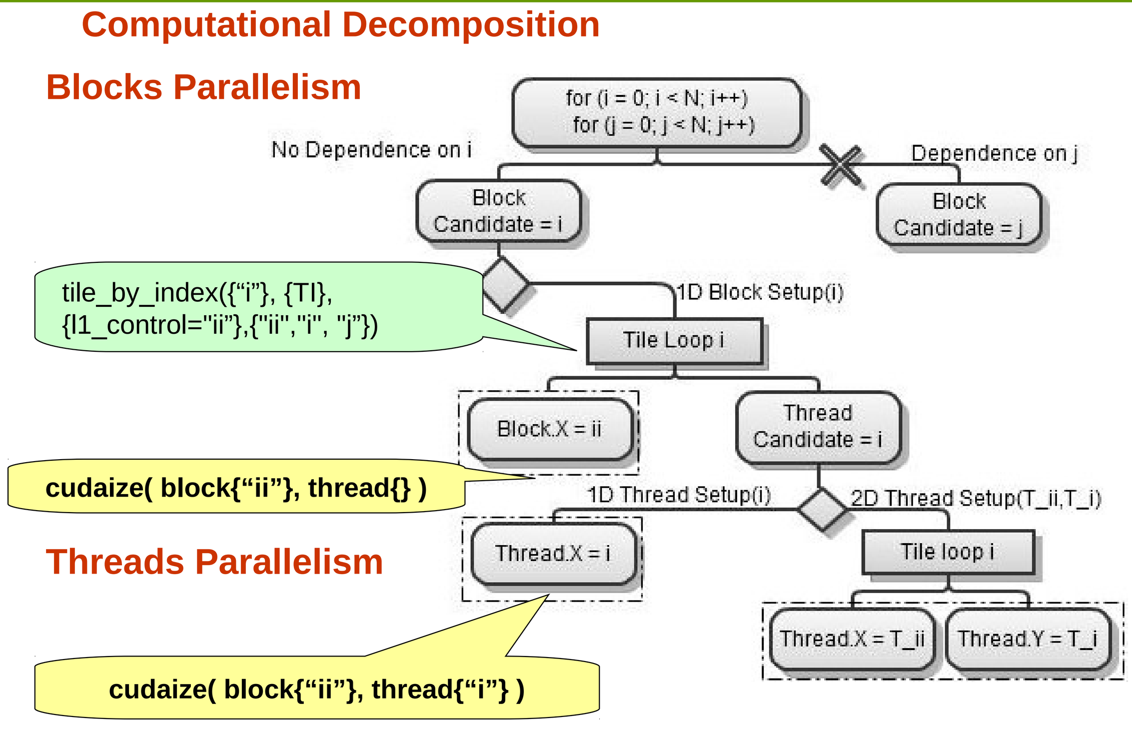
Purpose

Automatically derive transformation recipes for sequential loop nest computations (affine)

Parallel Mapping

- Select candidates for block and threads, through:
  - Dependence analysis &
  - Global memory coalescing
- Generate tiling for block and thread decomposition.

Computational Decomposition



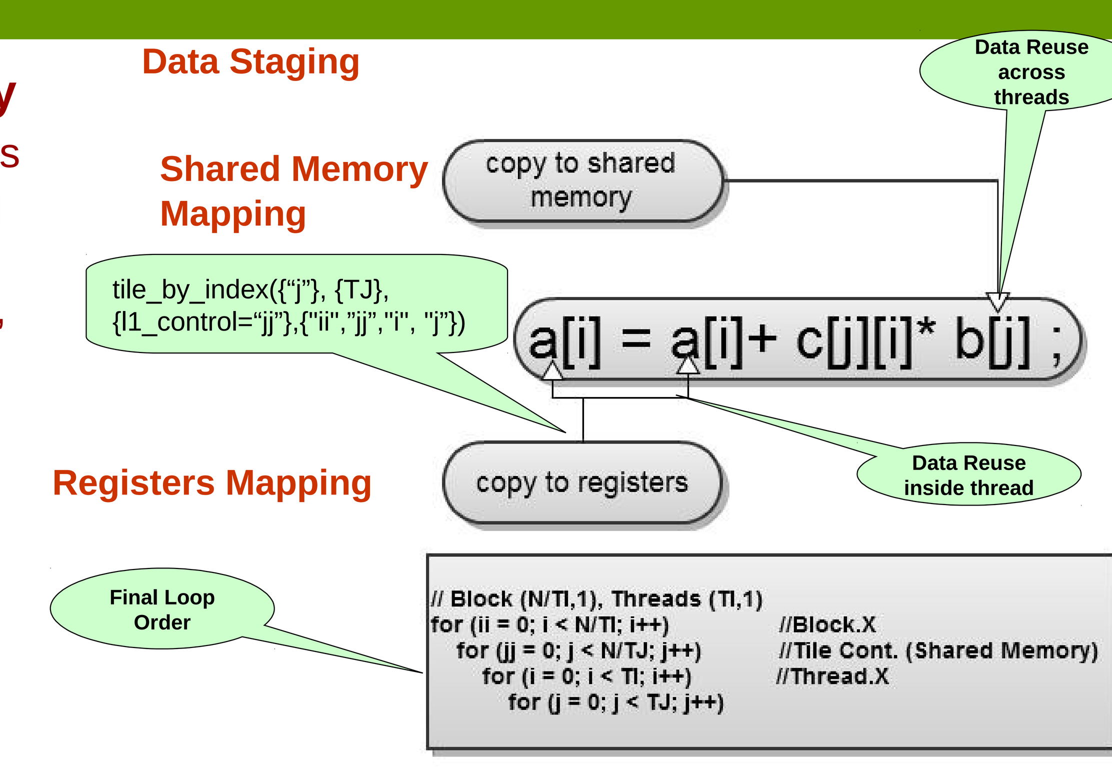
Manage Heterogeneous Memory Hierarchy

- Data with reuse inside threads, mapped to registers
- Data with reuse across threads, mapped to shared memory.
- Data with non-coalesced global memory accesses, mapped to shared memory.

Other Optimizations

- Aggressive loop unrolling to:
  - improve ILP
  - increase register reuse
  - reduce loop overhead.

Data Staging



Code Example

Matrix-Vector Multiply

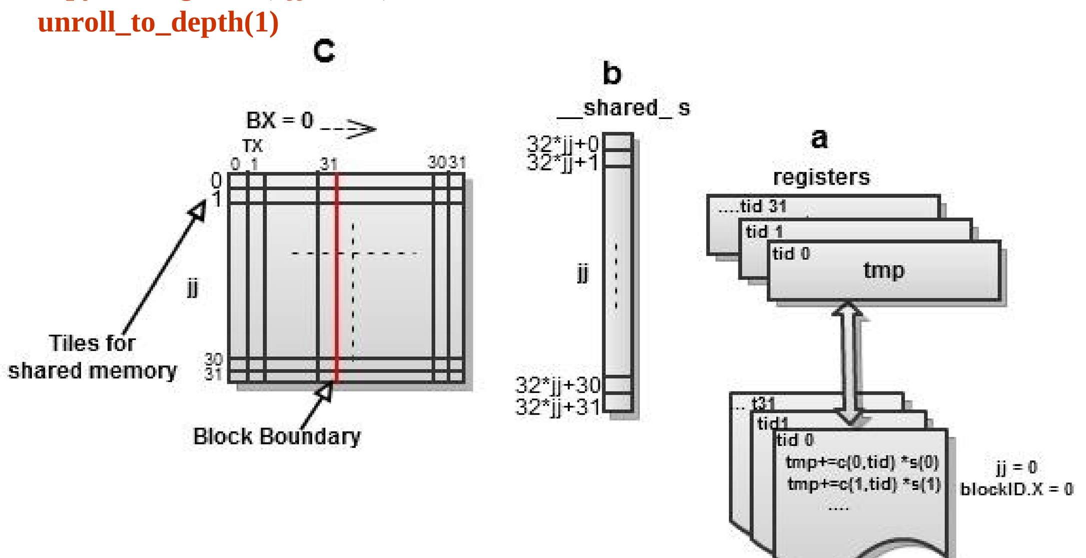
N = 1024

Sequential code*

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
      a[i] = a[i] + c[j][i] * b[j];
```

CUDA-CHiLL Recipe

```
N = 1024
TI= TJ = 32
tile_by_index({"i","j"}, {TI,TJ},{l1_control="ii", l2_control="k"},{"ii", "jj","i", "j"})
normalize_index("i")
cudaize("mv_GPU", {a=N, b=N, c=N*N},{block={"ii"}, thread={"i"}})
copy_to_shared("tx", "b", 1)
copy_to_registers("jj", "a")
unroll_to_depth(1)
```



* Following CUBLAS notation for matrix vector multiplication which takes transposed matrix as input.

GPU Code

Generated Code: with Computational decomposition only.

```
__global__ GPU_MV(float* a, float* b, float** c) {
int bx = blockIdx.x; int tx = threadIdx.x;
int i = 32*bx+tx;
for (j = 0; j < N; j++)
        a[i] = a[i] + c[j][i] * b[j]; }
```

Final MV Generated Code: with Data staged in shared memory & registers.

```
__global__ GPU_MV(float* a, float* b,
    float** c) {
int bx = blockIdx.x; int tx = threadIdx.x;
__shared__ float bcpy[32];
float acpy = a[tx + 32 * bx];
for (jj = 0; jj < 32; jj++) {
    bcpy[tx] = b[32 * jj + tx];
    __syncthreads();
//this loop is actually fully unrolled
   for (j = 32 * jj; j <= 32 * jj + 32; j++)
      acpy = acpy + c[j][32 * bx + tx] *
    bcpy [j];
     __syncthreads();
}
a[tx + 32 * bx] = acpy; }
```

2D Convolution: CUDA-CHiLL recipe and optimized code

Sequential Code

```
for(i=0;i<N;++i)
  for(j=0;j<N;++j)
    for(k=0;k<M;++k)
     for(l=0;l<M;++l)
        c[i][j] = c[i][j] + a[k+i][l+j] * b[k][l];
```

CUDA-CHiLL Recipe

```
N=4096, M=32
TI =32, TJ = 16, Tl=4
permute(0,{"j","i","k","l"})

tile_by_index({"j","i"}, {TI,TJ},
   {l1_control="jj", l2_control="ii"}, {"jj",
   "ii", "j", "i","k","l"})

normalize_index("j")
normalize_index("i")
cudaize("Kernel_GPU", {a=(N+M)*(N+M),
   b=M*M, c=(N+M)*(N+M)},
   {block={"jj","ii"}, thread={"j","i"}})
copy_to_shared("tx","a",-16)
copy_to_shared("tx","b",-16)
copy_to_registers("tx", "c")
Unroll_to_depth(1)
```

Complex bounds for shared memory copy loops

Copy Code for Shared Memory

Copy Code for Registers

Data structures selected for shared memory

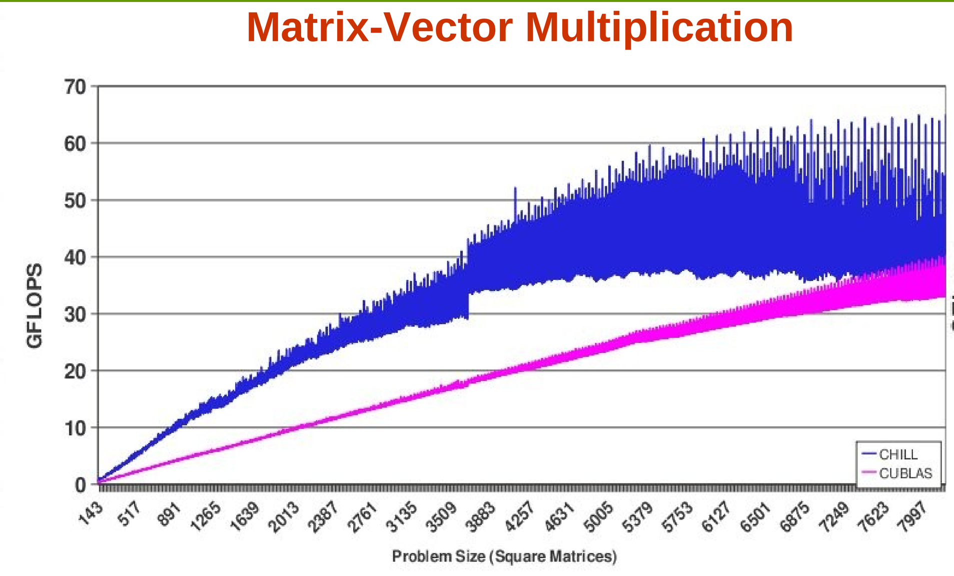
Optimized Code

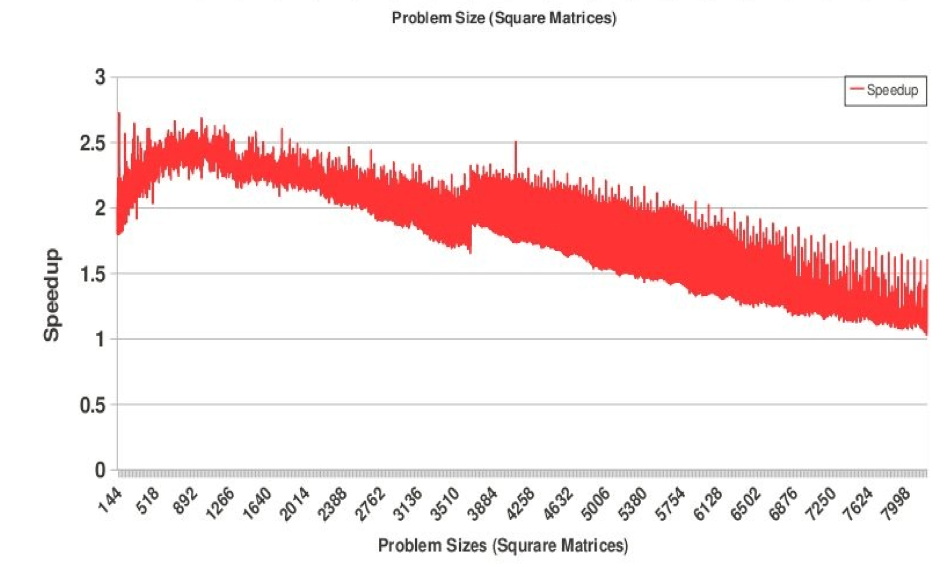
```
__shared__ float (_P1[47])[31];
 __shared__ float (_P2[16])[17];float tmp3;

for (tmp = 16 * by + 3 * ty; tmp <= min(16 * by + 30,
   16 * by + 3 * ty + 2); tmp++)
   for (tx1 = 2 * tx; tx1 <= min(2 * tx + 1, 46); tx1+
   +)
         _P1[ tx1][tmp - 16 * by] = a[tmp][32 * bx +
   tx1];
__syncthreads();
   for (tmp = 0; tmp <= 15; tmp++)
   for (tx1 = 2 * tx; tx1 <= 2 * tx + 1; tx1++)
         _P2[tx1][tmp] = b[tmp][tx1];
__syncthreads();
tmp3 = c[k + 16 * by][tx + 32 * bx];
for (k = 0; k <= 15; k++)
   for (l = 0; l <= 15; l++)
         tmp3 = tmp3 + _P1[l + tx ][k + ty] * _P2[l]
   [k];
c[k + 16 * by][tx + 32 * bx] = tmp3;
```

Performance Comparison

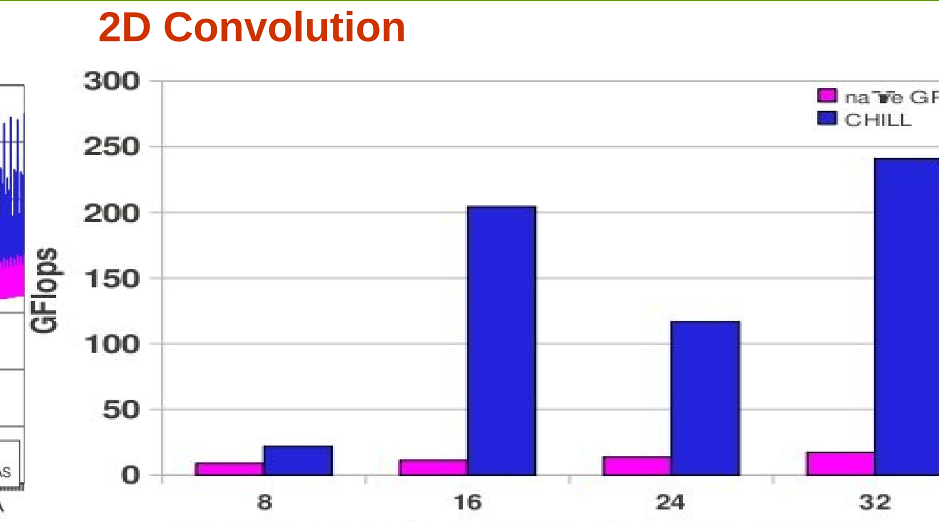
Name	Performance Summary				Generated Code		
	Max Perf	Max Spdup	Min Spdup	Avg Spdup	Min Len	Max Len	Num Ver
VV	14.2GF	16x	0.31x	1.20x	89	179	32
MV	65.0GF	2.73x	1.03x	1.78x	57	110	32
TMV	18.0GF	1.70x	0.82x	1.05x	45	60	2
MM	435.4GF	2.54x	0.93x	1.54x	326	1148	32

Matrix-Vector Multiplication

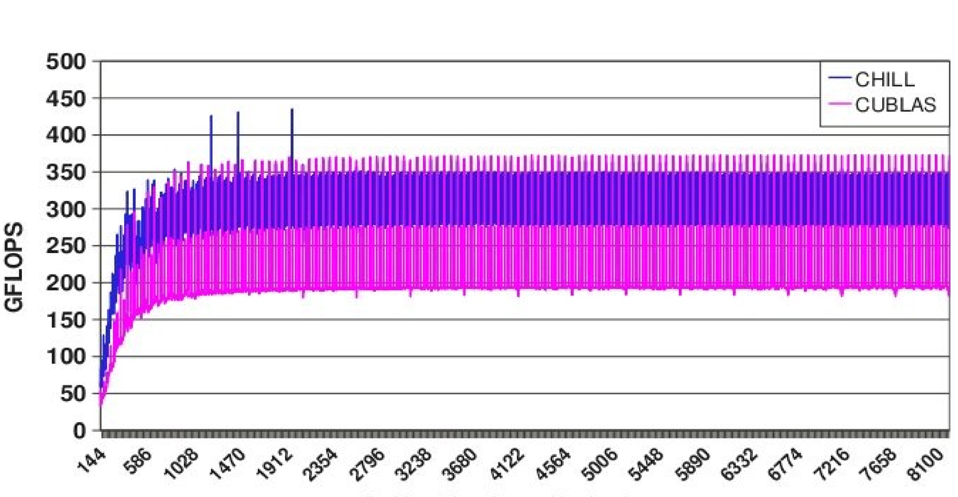


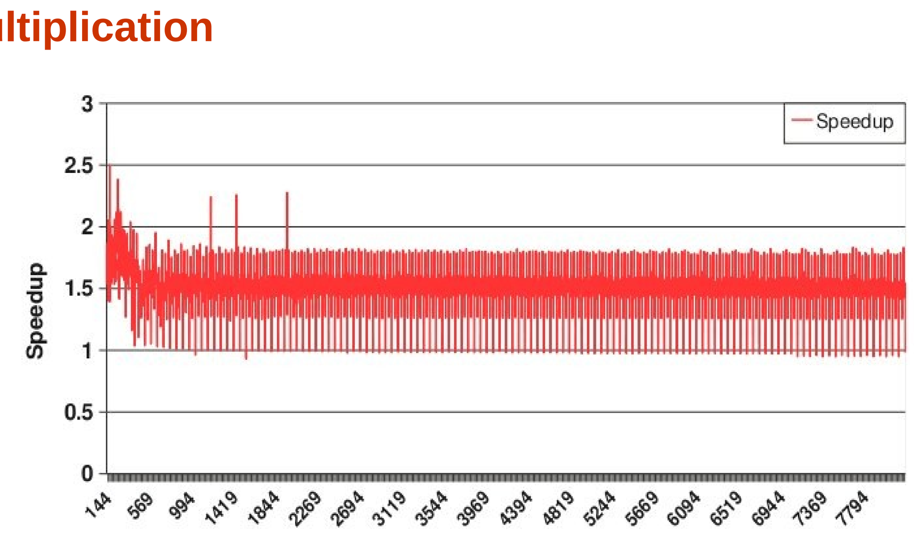


2D Convolution



Matrix-Matrix Multiplication





Matrix Transpose