GPU and multicore CPU architectures. Algorithm mapping

Contributors: N. Tsapanos, I. Karakostas, I. Pitas
Aristotle University of Thessaloniki, Greece

Presenter: Prof. Ioannis Pitas
Aristotle University of Thessaloniki

pitas@aiia.csd.auth.gr
www.multidrone.eu

Presentation version 1.3
GPU and multicore CPU architectures.

Algorithm mapping

• GPU and multicore CPU processing boards
  • Graphics cards
  • NVIDIA Jetson TX2
  • NVIDIA Jetson Xavier
• GPU programming
• Algorithm mapping:
  • Convolutions
Parallel algorithm execution

• Graphics computing:
  • Highly parallelizable

• Linear algebra parallelization:
  • Vector inner products: $c = x^T y$.
  • Matrix-vector multiplications $y = Ax$.
  • Matrix multiplications: $C = AB$. 
Parallel algorithm execution

- Convolution: \( y = Ax \)
  - CNN architectures, linear systems, signal filtering.
- Correlation: \( y = Ax \)
  - Template matching, tracking.
- Signal transforms (DFT, DCT, Haar, etc):
  - Matrix vector product form: \( X = Wx \)
  - 2D transforms (matrix product form): \( X' = WX \).
Processing Units

• Multicore (CPU):
  • MIMD.
  • Focused on latency.
  • Best single thread performance.

• Manycore (GPU):
  • SIMD.
  • Focused on throughput.
  • Best for embarrassingly parallel tasks.
Pascal microarchitecture


This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 731667 (MULTIDRONE)
Pascal microarchitecture

https://devblogs.nvidia.com/inside-pascal/gp100_sm_diagram/
GeForce GTX 1080

- Microarchitecture: Pascal.
- SMs: 20.
- CUDA cores: 2560.
- Clock (base/boost): 1607/1733 MHz.
- GFLOPs: 8873.
- DRAM: 8 GB GDDR5X at 10000 MHz.
- Memory bandwidth: 320 GB/s.
- L2 Cache: 2048 KB.
- L1 Cache: 48 KB per SM.
- Shared memory: 96 KB per SM.
GPU and multicore CPU architectures. Algorithm mapping

- **GPU and multicore CPU processing boards**
  - Graphics cards
  - NVIDIA Jetson TX2
  - NVIDIA Jetson Xavier
- **GPU programming**
- **Algorithm mapping:**
  - Convolutions
ARM Cortex-A57: High-End ARMv8 CPU

• **ARMv8 architecture**
  - Architecture evolution that extends ARM’s applicability to all markets.
  - Full ARM 32-bit compatibility, streamlined 64-bit capability.

• **High-performance next-generation microarchitecture.**
  - Improved performance on all workloads – 32b/64b integer, FP/SIMD.
  - Optimized for modern high-end workloads.

• ** Significant improvements in power efficiency.**
Features

• Fixed L1-cache size
  • 48 KB I-cache
  • 32 KB D-cache

• Configurable shared L2 cache
  • 512KB/1MB/2MB
  • Fully out-of-order execution

Instruction-Fetch / BPU

• **High-bandwidth instruction-fetch**
  • I-side prefetcher.

• **Large IC/TLB**
  • 48KB, 3-way.
  • 48-entry fully-associative L1 TLB.
    • Small/Large-page caching.

• **Low-latency, high-capacity branch predictor**
  • 2K-4K Branch Target Buffer (BTB).
  • 64-entry µBTB – zero-cycle taken-branch.
  • 512-entry indirect-predictor w/ path history.
Instruction-Fetch / BPU

• Fetch-information FIFO – fully out-of-order branch resolution.

• 32-instruction fetch queue – buffering into decode/rename.

• Extensive power-saving features throughout:
  • Way prediction, BTB/tag/TLB suppression, optimized RAMs, etc.
Instruction Dispatch/Retire

• Large instruction window:
  • > 128 in-flight instructions.

• Flexible speculative register file.

• Power optimized commit/retire:
  • 40 instruction bundles (multiple instructions /bundle).
  • Maintain precise exceptions at bundle granularity.

• 3-wide instruction dispatch.

• Sophisticated mispredict-recovery microarchitecture.

• Instruction-result handling optimized for 32b/64b operands.
## Cortex-A57 specifications

<table>
<thead>
<tr>
<th>Jetson TX2 CPU specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM ISA</td>
<td>ARMv8 (32/64-bit)</td>
</tr>
<tr>
<td>Decoder Width</td>
<td>3 micro-ops</td>
</tr>
<tr>
<td>Pipeline Length</td>
<td>18 stages</td>
</tr>
<tr>
<td>Branch Mispredict Penalty</td>
<td>15 cycles</td>
</tr>
<tr>
<td>Integer Add</td>
<td>2</td>
</tr>
<tr>
<td>Integer Mul</td>
<td>1</td>
</tr>
<tr>
<td>Load/Store Units</td>
<td>1 + 1 (dedicated L/S)</td>
</tr>
<tr>
<td>Branch Units</td>
<td>1</td>
</tr>
<tr>
<td>FP/NEON ALUs</td>
<td>2x128 bit</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>48 KB I$ + 32 KB D$</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>2 MB</td>
</tr>
</tbody>
</table>
Integer Execution pipelines

• Quad-issue into integer execution pipelines:
  • Independent OOO schedulers.
  • Low power issue-queues.
• Highly power-optimized for 32b/64b execution:
  • Activate only the datapath required for each instruction.
• Dual symmetric single-cycle ALU/shift.
• Dedicated multi-pass functional unit:
  • Shift+op.
  • MUL/DIV/MAC functionality.
• Dedicated branch resolution unit:
  • Full OOO execution.
  • High-performance mispredict-recovery microarchitecture.
FP/SIMD Execution pipelines

• Dual-issue into Neon/AdvSIMD pipelines:
  • Variable-latency functional units/pipeline.
  • 3 64-bit result buses.

• Fully compliant with IEEE754-2008 (ARMv8).

• Single and double precision SIMD capability.

• 1.5x-3x increase in IEEE-compliant FP bandwidth.
FP/SIMD Execution pipelines

- 11 FP/SIMD functional units:
  - Up to 4x32b per-unit execution capability.
  - Power optimized hybrid SP/DP functional units.
- 2x64b/4x32b FMUL/FADD.
- Dual FDIV units.
- ARMv8 cryptographic accelerator units:
  - AES, SHA, SHA2-256 — 5x performance improvement.
CPU Optimization

- Spawn threads.
- Use registers.
- Loop unrolling.
- Use SIMD capabilities.
- Take data locality into consideration.
- Trust the compiler.
Summary

• Continued scaling of high-end performance/power-efficiency:
  • Through targeted CPU/system microarchitectural advancements.
  • 1.5-2x performance increase from Cortex-A15.
• Maximum performance in smartphone power budget.
• Enhanced capabilities for enterprise:
  • Microarchitecture tuned to enterprise/high-end-mobile workloads.
• Next-generation architecture and microarchitecture:
  • Enabling a new breed of ARM computing solutions.
GPU and multicore CPU architectures. Algorithm mapping

- GPU and multicore CPU processing boards
  - Graphics cards
  - NVIDIA Jetson TX2
  - NVIDIA Jetson Xavier
- GPU programming
- Algorithm mapping:
  - Convolutions
NVIDIA Jetson Xavier

- AI Computer for autonomous machines
- Designed for robots, drones and other
- Multiple operating modes (10/15/30 W)
- Comparison to TX2:
  - Greater than 10x the energy efficiency.
  - More than 20x the performance
Jetson Xavier

• **More than 30 TOPS** (trillion operations per second) for deep learning and computer vision tasks.

• **512-core Volta GPU**: support for Tensor Cores and mixed-precision computation:
  • capable of up to 10 TFLOPS FP16 and 20 TOPS INT8.
## Jetson Xavier vs Jetson TX2

<table>
<thead>
<tr>
<th></th>
<th>Jetson Xavier</th>
<th>Jetson TX2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPU</strong></td>
<td>512-core Volta GPU with Tensor Cores</td>
<td>256-core Pascal GPU</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>8-core ARMv8.2 64-bit CPU, 8MB L2 + 4MB L3</td>
<td>HMP Dual Denver 2/2 MB L2 + Quad ARM® A57/2 MB L2</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>16GB 256-bit LPDDR4x</td>
<td>137 GB/s</td>
</tr>
<tr>
<td><strong>CSI</strong></td>
<td>Up to 16 simultaneous cameras</td>
<td>Up to 6 Cameras</td>
</tr>
<tr>
<td><strong>PCIe</strong></td>
<td>5x 16GT/s Gen 4 (1x8, 1x4, 1x2, 2x1)</td>
<td>Gen 2</td>
</tr>
<tr>
<td><strong>Footprint</strong></td>
<td>100mm x 87mm</td>
<td>50 mm x 87 mm</td>
</tr>
</tbody>
</table>
Jetson Xavier availability

• **Early access:** in August from the NVIDIA web store

• **General access:** in October from the NVIDIA web store

• **Price:** Jetson Xavier Developer Kit $1299 (USD)
GPU and multicore CPU architectures.

Algorithm mapping

- NVIDIA embedded processing boards
  - NVIDIA Jetson TX2
  - NVIDIA Jetson Xavier
- GPU programming
- Algorithm mapping:
  • Convolutions
CUDA

• Compute Unified Device Architecture (CUDA) is a parallel programming framework.

• Developed by Nvidia.

• Started as an attempt to give C/C++ programs access to GPU resources.

• Microarchitectures are name after famous physicists (Kepler, Maxwell, Pascal, Turing, Volta).
CUDA

• Data in CPU RAM are moved to device RAM, then device L2 cache then SM L1 cache.

• The CUDA kernel is the function that will run in parallel.

• When a kernel is launched, threads are grouped into blocks and all blocks form the CUDA grid for the kernel.

• Blocks are assigned to SMs in thread warps.

• Each CUDA kernel can handle 4 threads.

• GPU usage can be monitored through command line tools (nvidia-smi) or NVIDIA’s API (NVAPI).
CUDA Example

```
int main(void)
{
    int N = 1<<20;
    float *x, *y, *d_x, *d_y;
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));
    cudaMalloc(&d_x, N*sizeof(float));
    cudaMalloc(&d_y, N*sizeof(float));
    cudaMalloc(&d_x, N*sizeof(float));
    cudaMalloc(&d_y, N*sizeof(float));
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }
    cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
    saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(d_x);
    cudaFree(d_y);
    free(x);
    free(y);
}
```
CUDA Example

__global__
void saxpy(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

https://devblogs.nvidia.com/even-easier-introduction-cuda/
CUDA Optimization

- Use `cudaMallocHost()` or `cudaHostAlloc()`.
- Use `cudaMemcpy2D(dest, dest_pitch, src, src_pitch, w, h, cudaMemcpyHostToDevice)`.

Pageable Data Transfer

- Device
  - DRAM
  - Host
  - Pageable Memory
  - Pinned Memory

Pinned Data Transfer

- Device
  - DRAM
  - Host
  - Pinned Memory

CUDA Optimization

• Shared memory.

• Copy 2D piece of matrix.

CUDA Optimization

- Structure program to divide work into appropriate subtasks.

GPU and multicore CPU architectures. Algorithm mapping

- GPU and multicore CPU processing boards
  - Graphics cards
  - NVIDIA Jetson TX2
  - NVIDIA Jetson Xavier
- GPU programming
- Algorithm mapping:
  - Convolutions
GPU and multicore CPU architectures.

Algorithm mapping

• GPU and multicore CPU architectures
  • Multicore CPUs
  • GPUs

• Algorithm mapping:
  • Convolutions
Linear, cyclic 1D convolution

- The one-dimensional (linear) convolution of:
  - an input signal $x$ and
  - a convolution kernel (filter finite impulse response) $h$ of length $N$,

is given by:

$$y(k) = h(k) * x(k) = \sum_{i=0}^{N-1} h(i)x(k-i)$$

- For a convolution kernel centered around 0 and $N = 2\nu + 1$, it takes the form:

$$y(k) = h(k) * x(k) = \sum_{i=-\nu}^{\nu} h(i)x(k-i)$$
Linear, cyclic 1D convolution

• Convolution should not be confused with correlation:

\[ y(k) = \sum_{i=0}^{N-1} h(i) x(k + i) \]

• as they only coincide when:
  • convolution terms are transposed: \( x(k) \ast h(k) = h(k) \ast x(k) \) and
  • the centered convolution kernel is symmetric around 0: \( h(k) = h(-k) \), \( k = 1, \ldots, v \).

• Convolution: use in CNNs, linear systems, signal filtering.

• Correlation: use in template matching, object tracking.
Linear, cyclic 1D convolution

• One-dimensional cyclic convolution of length \( N \), \((k)_N = k \mod N\):

\[
y(k) = x(k) \otimes h(k) = \sum_{i=0}^{N-1} h(i)x((k - i)_N)
\]

• Embedding linear convolution in a cyclic convolution \( y(n) = x(x) \otimes h(n) \) of length \( N \geq L + M - 1 \) and then performing a cyclic convolution of length \( N \):

\[
y(k) = x(k) \otimes h(k) = \sum_{i=0}^{N-1} x_p(i)h_p((k - i)_N)
\]

requires:

• Zero-padding signal vector \( x \) to length \( N \): \( x_p = [x(0), \ldots, x(L - 1), 0, \ldots 0]^T \).
• Zero-padding convolution kernel vector \( h \) to length: \( h_p = [h(0), \ldots, h(M - 1), 0, \ldots 0]^T \).
Linear, cyclic 1D convolution

• Cyclic convolution can also be defined as:

\[ y = Hx \]

where:

• \( x = [x(0), \ldots, x(N - 1)]^T \): the input vector

• \( y = [y(0), \ldots, y(N - 1)]^T \): the output vector

• \( H \): a \( n \times N \) Toeplitz matrix of the form:

\[
H = \begin{bmatrix}
    h(0) & h(1) & h(2) & \ldots & h(N - 1) \\
    h(N - 1) & h(0) & h(1) & \ldots & h(N - 2) \\
    h(N - 2) & h(N - 1) & h(0) & \ldots & h(N - 3) \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    h(1) & h(2) & h(3) & \ldots & h(0)
\end{bmatrix}
\]
Linear and cyclic 2D and multidimensional convolutions

• Two-dimensional linear convolution of convolutional kernel $h$ of size $N_1 \times N_2$ is given by:

$$y(k_1, k_2) = h(k_1, k_2) \ast \ast x(k_1, k_2) = \sum_{i_1}^{N_1} \sum_{i_2}^{N_2} h(i_1, i_2) x(k_1 - i_1, k_2 - i_2)$$

• Its two-dimensional cyclic convolution counterpart of support $N_1 \times N_2$ is defined as:

$$y(k_1, k_2) = h(k_1, k_2) \oslash \oslash x(k_1, k_2) = \sum_{i_1}^{N_1} \sum_{i_2}^{N_2} h(i_1, i_2) x((k_1 - i_1)_{N_1}, (k_2 - i_2)_{N_2})$$
Linear and cyclic 2D and multidimensional convolutions

• A 2D linear convolution of convolutional kernel $h$ of size $N_1 \times N_2$ operating on an image $x$ of size $M_1 \times M_2$ of size produces an output image $y$:
  • of size $(N_1 + M_1 - 1)(N_2 + M_2 - 1)$, if the input image borders are padded with initial conditions (typically zero-valued pixels)
    • Complexity $O(N^4)$: $N_1 N_2 (N_1 + M_1 - 1)(N_2 + M_2 - 1)$ multiplications $O(N^4)$.
  • of size $(N_1 - M_1 + 1) (N_2 - M_2 + 1)$, without input image border padding.
    • Complexity $O(N^4)$: $N_1 N_2 (N_1 - M_1 + 1) (N_2 - M_2 + 1)$ multiplications.
Basic methods for speeding up convolution algorithms

• Definition-based parallelization of linear convolution.
• Spedup up based on linear algebra parallelization methods.
• Transform methods:
  • Cyclic convolution using the Discrete Fourier transform:
    \[ y(n) = IDFT[DFT(x(n)) \otimes DFT(h(n))] \]
    DFT, IDFT: forward and inverse DFT linear operators
  • Its computational complexity is less than that of the direct definition computation, if:
    \[ 6N_1N_2 \log_2(N_1N_2) < N_1^2N_2^2 \]
  • Block-based parallelization methods.
Overlap-add/overlap-save block-based methods

• Block-based methods:
  • The 2D overlap-add is based on the distributive property of convolution:
    • An image $x(i_1, i_2)$ can be divided into $K_1 \times K_2$ non-overlapping subsequences, having dimensions $N_{B1} \times N_{B2}$, each:
      $$x_{k_1k_2}(i_1, i_2) = \begin{cases} 
      x(i_1, i_2) & k_1N_{B1} \leq i_1 < (k_1 + 1)N_{B1}, \ k_2N_{B2} \leq i_2 < (k_2 + 1)N_{B2} \\
      0 & \text{otherwise}
      \end{cases}$$
    • The sequence $x(i_1, i_2)$ can be easily reconstructed from its blocks:
      $$x(i_1, i_2) = \sum_{k_1=1}^{K_1} \sum_{k_2=1}^{K_2} x_{k_1k_2}(i_1, i_2)$$
    • Overlp-save works along similar lines.
Basic methods for speeding up convolution algorithms

- The linear convolution output $y(n_1, n_2)$ is the sum of the (easily parallelizable) convolution outputs produced by the input sequence blocks:

$$y(i_1, i_2) = x(i_1, i_2) \ast h(i_1, i_2)$$

$$= \sum_{k_1=1}^{K_1} \sum_{k_2=1}^{K_2} (x_{k_1k_2}(i_1, i_2)) \ast h(1_1, 1_2) = \sum_{k_1=1}^{K_1} \sum_{k_2=1}^{K_2} y_{k_1k_2}(i_1, i_2)$$
Fast 1D cyclic convolution algorithms with minimal complexity

- Winograd convolution algorithms:
  \[ Y = C(Ax \otimes Bh) \]
  - They require only \( 2N - \nu \) multiplications in their middle vector product, thus having minimal complexity.
  - \( \nu \): number of cyclotomic polynomial factors of polynomial \( z^N - 1 \) over the rational numbers \( Q \).
- GEneral Matrix Multiplication (GEMM) BLAS or CUBLAS routines can be used.
Fast 1D convolution algorithms with minimal computational complexity

• Can be equivalently expressed as:
  \[ Y = RB^T (Ax \otimes C^TRh) \]

• Matrices \( A, B \) typically have elements \( 0, +1, -1 \).

• \( R \) is an \( N \times N \) permutation matrix.

• If \( N = N_1 N_2 \ldots N_n \), the Winograd convolution takes the ‘nested’ form:
  where \( \times \) denotes Kronecker product
  \[ Y = C_n \times \ldots \times C_1 (A_n \times \ldots \times A_1 x \otimes B_n \times \ldots \times B_1 h) \]
Fast 2D linear convolution

- A $3 \times 3$ filter $h$ operating on a $4 \times 4$ pixel image $x$ produces an output image $y$ of size $2 \times 2$ pixels (without input border zero-padding and requires $9 \times 2 \times 2 = 36$ multiplications).
- A fast Winograd-like linear convolution algorithms has the form:

$$Y = A^T [(CGC^T) \odot (B^TDB)] A$$

where $\odot$ denotes the Hadamard product, $G$ is a $3 \times 3$ filter, $D$ is a $4 \times 4$ image tile,

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

- This allows the computation of the $2 \times 2$ output of the application of a $3 \times 3$ filter on a $4 \times 4$ image tile with 16 multiplications, instead of 36, for an acceleration of 2.25.
Fast 2D $3 \times 3$ cyclic convolution algorithm having minimal computational complexity

- Fast 2D $3 \times 3$ cyclic convolution algorithm:

$$y = C(Ax \otimes Bh)$$

where:

$$A = B = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & -1 & 1 & 0 & -1 & 1 & 0 & -1 \\
0 & 1 & -1 & 0 & 1 & -1 & 0 & 1 & -1 \\
1 & -1 & 0 & 1 & -1 & 0 & 1 & -1 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & -1 & -1 \\
0 & 0 & 0 & 1 & 1 & 1 & -1 & -1 & -1 \\
1 & 1 & 1 & -1 & -1 & -1 & 0 & 0 & 0 \\
1 & 0 & -1 & 0 & -1 & 1 & -1 & 1 & 0 \\
0 & 1 & -1 & 1 & -1 & 0 & -1 & 0 & 1 \\
1 & -1 & 0 & -1 & 0 & 1 & 0 & 1 & -1 \\
1 & 0 & -1 & -1 & 1 & 0 & 0 & -1 & 1 \\
0 & 1 & -1 & -1 & 0 & 1 & 1 & -1 & 0 \\
1 & -1 & 0 & 0 & 1 & -1 & 1 & 0 & 1
\end{bmatrix}$$

$$C = \frac{1}{27} \begin{bmatrix}
3 & 3 & -6 & 3 & 3 & -6 & 3 & -2 & 4 & -2 & 3 & -6 & 3 \\
3 & 3 & 3 & -6 & 3 & -6 & 3 & -2 & -2 & 4 & 3 & 3 & -6 \\
3 & -6 & 3 & 3 & 3 & -6 & 3 & 4 & -2 & -2 & -6 & 3 & 3 \\
3 & 3 & -6 & 3 & 3 & 3 & -6 & 7 & -5 & -2 & -6 & 3 & 3 \\
3 & 3 & 3 & -6 & 3 & 3 & -6 & -2 & 7 & -5 & 3 & -6 & 3 \\
3 & -6 & 3 & 3 & 3 & -6 & -5 & -2 & 7 & 3 & 3 & -6 & 3 \\
3 & 3 & -6 & 3 & -6 & 3 & 3 & -5 & 1 & 4 & 3 & 3 & -6 \\
3 & 3 & 3 & -6 & 3 & 3 & 4 & -5 & 1 & -6 & 3 & 3 & -6 \\
3 & -6 & 3 & 3 & -6 & 3 & 3 & 1 & 4 & -5 & 3 & -6 & 3
\end{bmatrix}$$
Convolution algorithms in CNN

• Typical 2D convolutional layer \( l \) of a CNN:

\[
x(i, j, c_{l+1}, l+1, k) = f(b(l, k) + \sum_{c=1}^{C_{l}} \sum_{i'=0}^{H_{l,k}} \sum_{j'=0}^{W_{l,k}} h(i', j', l, k)x(i-i', j-j', c, l, k))
\]

• input feature map \( \mathbf{X}_l : N_{l} \times M_{l} \times C_{l} \)-dimensional 3D tensor

• \( \mathbf{w}_{l,k} : N_{l,k} \times W_{l,k} \times C_{l} \)-dimensional 3D tensor

• \( b(l, k) \): bias term

• \( f \): nonlinear activation function

• Very expensive \( O(N^4) \), when calculated by definition.

• \( H, W \) are typically small (3x3, 5x5).

• It can be parallelized in various ways.
2D convolution algorithms in computer vision/image processing

• The typical 2D linear convolution can be used in:
  • Image filtering
  • Filter mask $N \times M$ is typically small ($3 \times 3, 5 \times 5$).
  • Direct implementation is preferable.

• The typical 2D linear correlation can be used in:
  • Template matching
  • Object detection
  • Template mask $N \times M$ is typically large (e.g. $100 \times 100$ pixels).
  • Transform implementation is preferable.
Thank you very much for your attention!

Contact: Prof. I. Pitas
pitas@aiia.csd.auth.gr
www.multidrone.eu