1. the CPU is our bottleneck, it can’t make computations in the needed rate to reach FSB Target, the sequential nature of CPU execution limits performance.
2. FPGAs are ideal for accelerating image blur due to:

* Parallelism – Process multiple pixels simultaneously.
* Custom Hardware Implementation – Directly execute blur operations without CPU overhead.
* Pipelining – Increase throughput by structuring computation in stages.

1. **CPUs execute instructions sequentially, one after another. FPGAs, on the other hand, can perform many operations simultaneously.**

This massive parallelism compensates for the lower clock speed.

- Profiling and Analysis – Identify bottlenecks and analyze data dependencies.

* Algorithm Transformation – Restructure code for parallel execution.
* Hardware Design – Implement using Verilog or High-Level Synthesis (HLS) tools.
* Verification and Testing – Validate functionality via simulation and real FPGA testing.

FPGA Clock Speed is lower than CPU clock (by a massive margine).

Despite lower clock speeds, **FPGAs outperform CPUs** because they:

* Execute **many operations in parallel** instead of sequentially.
* Use **custom circuits** for specific tasks, avoiding instruction decoding overhead.
* Employ **pipelining**, improving efficiency despite lower frequency.

Basically, this is due to FPGA Parallelism.

1. The host CPU acts as the controller and is responsible for:

* Initializing the FPGA and allocating memory.
* Transferring data between CPU and FPGA memory.
* Launching kernels (hardware-accelerated functions on the FPGA).
* Handling sequential logic isn’t efficient for the FPGA.
* Retrieving results and performing final processing.



* Data Locality – Keep frequently used data on the FPGA as long as possible.
* Coalesced Memory Access – Optimize data layout to minimize unnecessary memory reads/writes.
* Overlapping Computation and Communication – Transfer data while computation is still running.
* Efficient Direct Memory Access (DMA) – Use fast data transfer methods instead of CPU-managed transfers.

void printImage(uint8\_t \*image, int M, int N) {

int paddedN = (N % 4 == 0) ? N : N + (4 - N % 4);

uint8\_t output[M + 1][paddedN];

// Copy image and pad last row and column

for (int i = 0; i < M; i++) {

for (int j = 0; j < N; j++) {

output[i][j] = image[i \* N + j];

}

for (int j = N; j < paddedN; j++) {

output[i][j] = output[i][N - 1]; // Duplicate last column

}

}

for (int j = 0; j < paddedN; j++) {

output[M][j] = output[M - 1][j]; // Duplicate last row

}

// Print words (4 bytes at a time)

for (int i = 0; i < M + 1; i++) {

for (int j = 0; j < paddedN; j += 4) {

printf("{%d, %d, %d, %d}\n",

output[i][j], output[i][j + 1],

output[i][j + 2], output[i][j + 3]);

}

}

}

void printImage(uint8\_t \*image, int M, int N) {

int paddedN = (N % 4 == 0) ? N : N + (4 - N % 4);

uint8\_t output[M + 1][paddedN];

// Copy image and pad last row and column

for (int i = 0; i < M; i++) {

for (int j = 0; j < N; j++) {

output[i][j] = image[i \* N + j];

}

for (int j = N; j < paddedN; j++) {

output[i][j] = output[i][N - 1]; // Duplicate last column

}

}

for (int j = 0; j < paddedN; j++) {

output[M][j] = output[M - 1][j]; // Duplicate last row

}

// Print interleaved first and second outputs

for (int i = 0; i < M + 1; i++) {

for (int j = 0; j < paddedN; j += 4) {

printf("{%d, %d, %d, %d}\n", output[i][j], output[i][j + 1], output[i][j + 2], output[i][j + 3]);

if (i > 0) // Second printout starts from the second row

printf("{%d, %d, %d, %d}\n", output[i - 1][j], output[i - 1][j + 1], output[i - 1][j + 2], output[i - 1][j + 3]);

}

}

}