# High-Performance Image Processing Application Implementation and Optimization on HammerBlade Manycore Architecture

Yijue (Edria) Li

Author Note

First paragraph: Project Purpose and Introduction

Second paragraph: Gaussian blur algorithm Introduction

Third paragraph: Gaussian blur algorithm implementation and optimization

Fourth paragraph: Summary and future work

Code for Gaussian Blur can be found here:

https://github.com/Edria-Li/CSE549_FinalDeliverable

**Abstract**

HammerBlade is a data-processing architecture that can scale from super-computer systems to mobile devices. Its software paradigm is Single-Program Multiple Data (SPMD) which allows users to write multiple kernels and run them in CUDA-lite runtime. The CUDA-Lite runtime allows users to read and write data in device DRAM, enqueue applications for launch, launch applications, and copy results back to the host. HammerBlade architecture incorporates a highspeed accelerator onchip network (NoC) which has both forward and reverse paths, supports both remote load and remote store request to globally mapped memory ranges of other tiles or external memories. Such a flexible architecture enables more possibilities in algorithm implementations. This project is intended for evaluating the capability and performance of running different algorithms in image processing application domain with different scheduling and memory allocation schemes and try to optimize the implementation to get the best performance. Specifically, this project implements a frequently-used image processing algorithm -- Gaussian Blur -- on HammerBlade manycore architecture. The implementation is optimized in various ways like applying memory level parallelism, cache locality and different data sharing schemes. We also performed an in-depth discussion of performance analysis for each algorithm and estimated and compared the overall performance with other platforms.

*Keywords:* Parallel architecture, HammerBlade, image processing, algorithm optimization

# 1. PROJECT PURPOSE AND INTRODUCTION

## 1.1. PURPOSE OF THIS PROJECT

This project is intended for evaluating the capability and performance of running different algorithms in image processing applications domain with different scheduling and memory allocation schemes and try to optimize the implementation to get the best performance.

## 1.2. HAMMERBLADE OVERVIEW

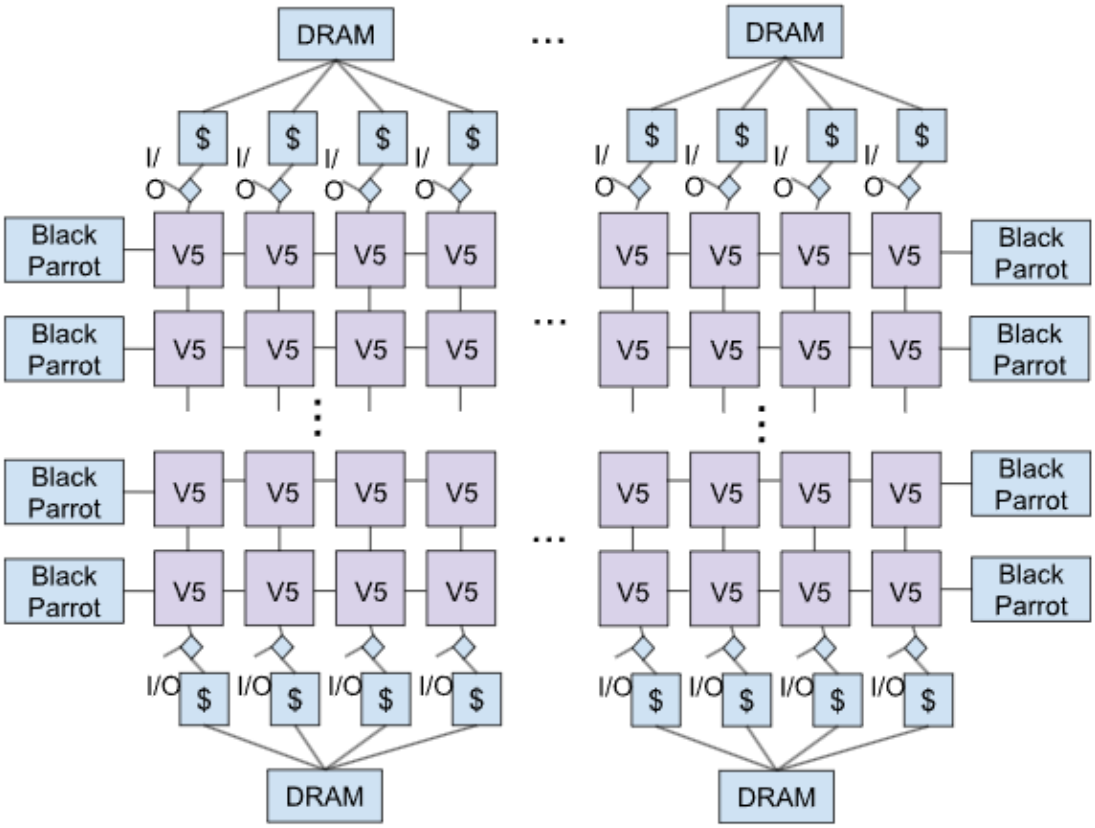



Figure 1. Hammerblade architecture

HammerBlade is a data-processing architecture that can scale from super-computer systems to mobile devices. A HammerBlade Node is architected from an array of tiles connected by a 2-D mesh network, called the manycore accelerator network, and is attached to a reconfigurable high-bandwidth memory system and the I/O system.

## 2. GAUSSIAN BLUR ALGORITHM INTRODUCTION

In image processing, Gaussian blur is used to reduce image noise and details by applying a Gaussian function to the image. Mathematically, similar to Fourier transform using Gaussian window to denoise the high-frequency components, applying Gaussian blur convolves the Gaussian distribution to the original image thus serving as a low-pass filter. In a one-dimensional data distribution, formula of Gaussian function is as follows:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

Two-dimensional Gaussian is further defined as $G(x, y) = G(x) \cdot G(y)$ and its formula as follows:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Values from the Gaussian distribution are used to build a convolution matrix with the sum of all elements equal to 1. The convolution matrix is then applied to the original image; that is, each pixel is set to a weighted average of its neighborhood, with the center original pixel having the most weight.

Given a two-dimensional grayscale image $I(x, y)$, the blurred image is defined by:

$$\widehat{I}(x, y) = \iint_{\Omega} G(u, v; x, y) \cdot I(u, v; x, y) \, \mathrm{d}u \, \mathrm{d}v$$

where $G(u, v; x, y)$ is the Gaussian function $G(u, v)$ whose peak is centered at $(0, 0)$ that is now shifted to $(x, y)$.

For discrete convolution, as to be used for grayscale image data of which the pixels can be represented in matrix with element $I_{ij}$, the above continuous formula can be further discretized in Riemann Sum as follows:

$$\widehat{I}(x, y) = \sum_{i}^{n} \sum_{j}^{n} (G(u, v; x, y) \cdot I(u, v; x, y))$$

We will use this discretized formula for our algorithm implementation.

## 3. GAUSSIAN BLUR ALGORITHM IMPLEMENTATION AND OPTIMIZATION

### 3.1. PARAMETERS, CONFIGURATION, AND BASELINE SCHEDULING

Here is our intended parameters and configuration for input and output:

| Parameters / Configurations | Value |
|---|---|
| HammerBlade Tiles | 16 × 8 (one full pod) |
| Input Image Size | 64 × 64 × 3 (RGB) |
| Output Image Size | 64 × 64 × 1 (Grayscale) |

Table 1. Basic parameters and configurations for Gaussian blur implementation.

### 3.1.1. GENERATE AND APPLY GAUSSIAN FILTER

Our goal is to implement Gaussian Blur algorithm, which in our case applies to color images of size 64 × 64 × 3 (RGB) and generates a blurred grayscale image of size 64 × 64 × 1 (black and white), on HammerBlade architecture.

First, we convert RGB to grayscale:

$$Gray = 0.299 * R + 0.587 * G + 0.114 * B$$

Instead of using bfloat16 operation, here we use int16 operation instead, the code is as follows:

$$Gray = ((77 * R + 150 * G + 29 * B) \gg 8);$$

After we converted the RGB image to grayscale, we now have a size 64 × 64 × 1 image. Our next step is to generate a Gaussian filter with kernel size 3∗3. We used $\sigma$ = 1.5 and generated a symmetric matrix (Figure 2 (a)) with the highest weight in the center while the neighbor has smaller weights as the distance to center increases, and then apply the filter onto the image with indices shown in Figure 2 (b):

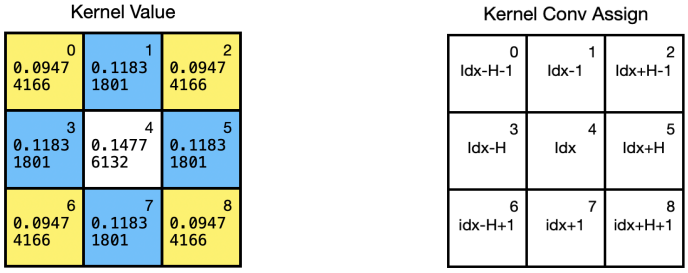


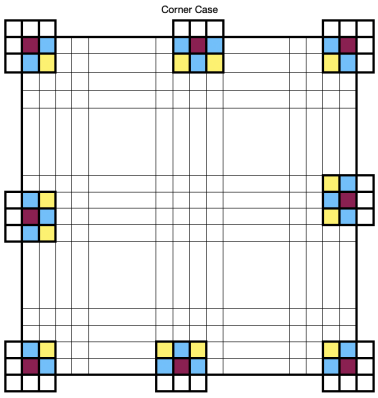

Figure 2. (a) left: Kernel Values generated with 2D Gaussian Distribution with $\sigma$ = 1.5.
(b) right: Kernel assignment when convolving onto the original image pixel indices.

The convolution is accomplished through the discretized expression with specific parameters:

$$\widehat{I}(x,y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} (G(x+i, y+j) \cdot I(x+i, y+j)), \qquad x, y \in [0, 63]$$

where `i` = -1 (in x direction) corresponds to `idx-H` (one pixel to the left) and $i$= +1 corresponds to `idx+H` (one pixel to the right) for coding indices; similarly, $j$ = -1 (in y direction) corresponds to `idx-1` (one pixel down) and $j$ = +1 corresponds to `idx+1` (one pixel up).

For our baseline algorithm, we apply the filter to every single pixel including the corners, which we used `if` branches to include all eight corner scenarios as follows:




```
if (i > 0   && j > 0  ) B[idx] += kernel[0] * A[idx-H-1];
if (i > 0             ) B[idx] += kernel[1] * A[idx-1  ];
if (i > 0   && j < H-1) B[idx] += kernel[2] * A[idx+H-1];
if (j > 0             ) B[idx] += kernel[3] * A[idx-H  ];
                        B[idx] += kernel[4] * A[idx    ];
if (j < H-1           ) B[idx] += kernel[5] * A[idx+H  ];
if (i < H-1 && j > 0  ) B[idx] += kernel[6] * A[idx-H+1];
if (i < H-1           ) B[idx] += kernel[7] * A[idx+1  ];
if (i < H-1 && j < H-1) B[idx] += kernel[8] * A[idx+H+1];
```

Figure 3. (a) Eight Corner Cases for applying Gaussian Filters on the grayscale image.
(b) code segment for if condition implementation.

For an input image to become a blurred output, the following table shows the computation intensity for each pixel:

| **Operand (Baseline Unoptimized)** | **Number of Operations for Each Pixel** |
|---|---|
| RGB2Gray | 3 Multiplications + 2 Additions |
| Indices Calculation | 8 Additions |
| Gaussian Filter (non-edge) | 9 Multiplications + 8 Additions |
| Edge Conditions | 12 Comparisons + 6 Additions + 4 And |
| Total | 52 operations/pixel |

Table 2. Number of Computations for each pixel in Gaussian Blur Algorithm.

Now we consider how to schedule onto the HammerBlade architecture.

### 3.1.2. BASELINE SCHEDULING

For vanilla core tile assignment, we first calculated the total number pixels divided by the total number of cores: $(64 \times 64) \div (16 \times 8) = 32$, i.e. each core will need to execute 32 pixels. Therefore, we assign 32 pixels in one single column to one core, and each column of total 64 pixels will need 2 cores (see Figure 4 below).

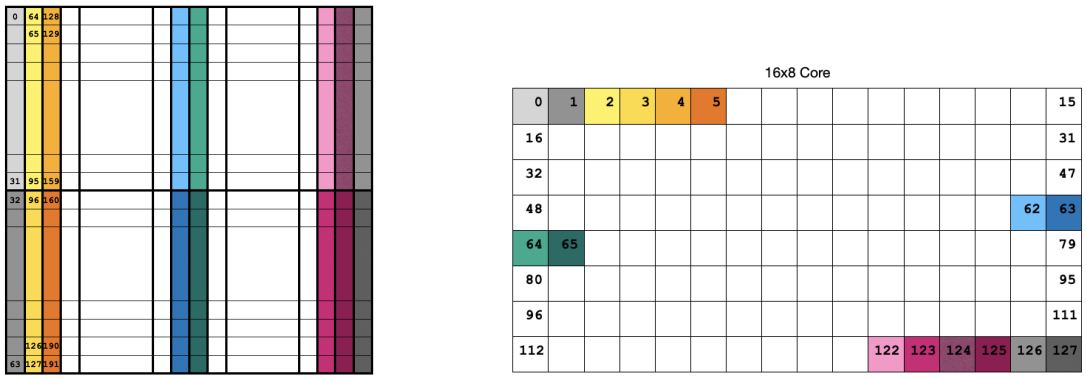



Figure 4. Pixel Vector to Core Tile Assignment with same-shaded-lot examples showing mapping of vectors to tile. Left (a) vectorized column of pixels with each column has 2 vectors each with 32 pixels. Right (b) 128 Vanilla Cores to be assigned with pixel vectors with the same shade indicated, each executing 32 pixels.

We expect our baseline performance to be really awful to begin with, given our 8 `if` conditions (see Figure 3(b) for code segment) that need to be executed sequentially and memory load for each execution without any memory level parallelism. Our observed baseline performance meets our expectation (Table 3):

| Attempt | Warm Cache | DRAM Utilization | Vcache Utilization | Core Utilization | Runtime | Total Instruct. | Total Core Stall | DRAM Load Stall |
|---|---|---|---|---|---|---|---|---|
| Baseline | No | 4.09% | 19.98% | 7.56% | 54,311 | 401,166 | 4,611,610 | 2,208,334 |

Table 3. Unoptimized baseline performance for Gaussian Blur algorithm.

Even if we unrolled the loop, merely marginal benefit can be observed:

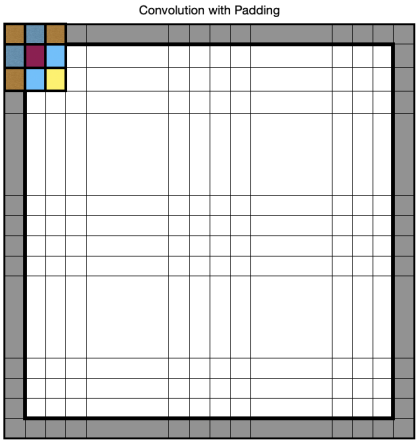
| Attempt | Warm Cache | DRAM Utilization | Vcache Utilization | Core Utilization | Runtime | Total Instruct. | Total Core Stall | DRAM Load Stall |
|---|---|---|---|---|---|---|---|---|
| Baseline (`if` w/ unroll) | No | 5.86% | 15.79% | 8.77% (+16.01%) | 39,188 (-27.9%) | 338,088 (-15.7%) | 3,318,265 (-28.05%) | 1,507,206 (-31.75%) |

Table 4. Unoptimized baseline with if condition and with loop unrolling.
Marginal benefit comparison to Table 3 performance.

Therefore, our next optimize attempt will focus on removing branch conditions.

### 3.2. Optimize Attempt 1: with Padding

We chose to use padding = 1, i.e. to left out the outside frame and only focus on convolve the 62 × 62 “inner image” for simplification (see Figure 5 below).




```
for (int idx = idx_start; idx < idx_end; idx++) {
  B[idx] += kernel[0] * A[idx-H-1];
  B[idx] += kernel[1] * A[idx-1  ];
  B[idx] += kernel[2] * A[idx+H-1];
  B[idx] += kernel[3] * A[idx-H  ];
  B[idx] += kernel[4] * A[idx    ];
  B[idx] += kernel[5] * A[idx+H  ];
  B[idx] += kernel[6] * A[idx-H+1];
  B[idx] += kernel[7] * A[idx+1  ];
  B[idx] += kernel[8] * A[idx+H+1];
 }
}
```

Figure 5. (a) Visualization for padding = 1 scenario with 62×62 pixels to convolve the Gaussian filter. (b) Code segment for convolution without `if` condition.

We recalculate core tiles for parallel execution: (62×62) ÷ (16×8) ≈ 30.03, therefore we need to dispatch 31 pixels for each tile. And we will need 124 cores total (62×62÷31=124) to compute all pixels in parallel. Therefore, given the padding frame, we chose to use cores of `_bsg_id` = 2 to 125, of which can later be used in vcache column alignment. For example, pixels number 65 to 95 are assigned to core 2, pixel 129 to 159 are assigned to core 4, etc. Since now we have less branch conditions to calculate, we hypothesized that our computation intensity will decrease by 44.11% – originally 52 (with if) × 32 (# of pixels) operations for each core tile, now 30 (no if operands) ×31 (# of pixels) operations – which can be embodied through runtime decrease and total instruction decrease.

After we removed the branch conditions, our performance meets our expectation: core utilization increased now from 7.56% to 24.24% by +2.2x, with runtime decreased from 54,311 now to 12,951 by 76.15%. DRAM load stall has now lowered from 2,208,334 to 792,620 (-64.11%):

| **Attempt** | **Warm Cache** | **DRAM Utilization** | **Vcache Utilization** | **Core Utilization** | **Runtime** | **Total Instruct.** | **Total Core Stall** | **DRAM Load Stall** |
|---|---|---|---|---|---|---|---|---|
| Optim#1 Padding (no `if`) | No | 12.21% | 17.71% | 24.24% (+ 2.2x) | 12,951 (-76.2%) | 329,877 (-17.8%) | 1,030,908 (-77.65%) | 792,620 (-64.11%) |

Table 5. Optimization #1 with padding, removed if conditions. Performance compared with baseline benchmark.

As expected, DRAM load stall encompasses 58.24% of the total core stall; therefore, our next step is to unroll the `for` loop to hide latency for DRAM loading.

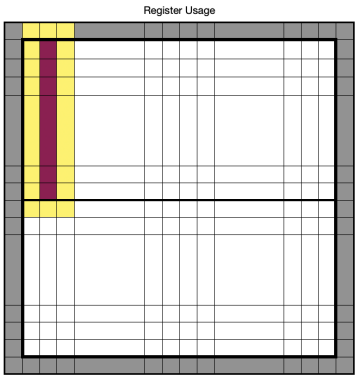



Figure 6. Register usage calculation for Optimization Attempt 2. The 31 burgundy pixels are to be put into registers for output, the yellow 33∗3 columns (including the burgundy pixels inside) of pixels needed in registers for the Gaussian filter.

### 3.3. Optimize Attempt 2: with Memory Level Parallelism

For better use of memory level parallelism, we first need to calculate how many short ints can be stored in the registers. For each core, we have 32 registers of size 32 bit; therefore, one tile can register 64 short ints, i.e. 64 pixels. In order to hide the most latency, we design to also include the filters in the registers for each loop. Therefore, for each core tile, we are in need of 31 registers for the blurred pixels for output, and 99 registers for the 3∗3 Gaussian filter (as visualized in Figure 6 above).

To traverse through all 31 pixels using registers, we therefore will need to loop 3 times (130 mod 64 + 1 = 2 +1 = 3). For a more balanced loop structure, we design to operate on 11, 11, and 9 pixels respectively in each loop. Our hypothesis for performance: since we hide the latency in 3 loops, we expect 10 times (31/3) less of stall on DRAM load compared to no unrolling (optimization #1), i.e. will roughly decrease by 90%.

Our observed result of performance meets our expectation: compared with baseline benchmark, the core utilization increased 5.56x, and -86.67% DRAM load stall (close to our expectation) compared to padding without unrolling the loop; runtime decreases significantly from baseline (-91.5%) as well as padding only (-64.31%).

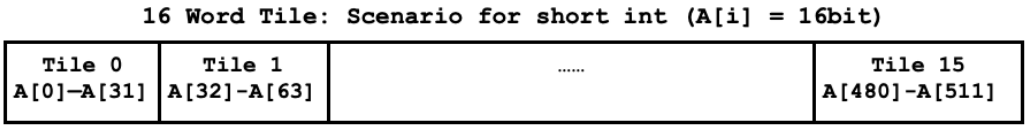
| Attempt | Warm Cache | DRAM Utilization | Vcache Utilization | Core Utilization | Runtime | Total Instruct. | Total Core Stall | DRAM Load Stall |
|---|---|---|---|---|---|---|---|---|
| Optim#2 Padding w/ MLP | No | 35.29% | 12.16% | 49.59% | 4,622 | 267,759 | 271,435 | 105,635 |
| Baseline | | | | + 5.56x | -91.5% | -33.25% | -94.11% | -95.22% |
| Optim#1 | | | | + 1.05x | -64.31% | -18.83% | -73.67% | -86.67% |

Table 6. Optimization #2 with padding and with an unrolled loop (MLP). Expect 90% drop in DRAM load stall from optimization #1. Performance compared with baseline benchmark as well as Optimization #1.

But still, we observed a high cache miss of 28.85% and a high DRAM utilization of 35.29% without cache. Therefore, our next step is to consider warm cache.

### 3.4. Optimize Attempt 3: with Warm Cache

Each of the Hammerblade vcache tile has 32KB storage with 16-word cache line size; and for a full pod, we have 16 cache tiles in both south and north side of the core tiles. Therefore, we can align the 16 cache tiles (on each side) with the 16 columns of cores. Since we are using short int for each pixel, vcache tile 0 will thus store 0-31 pixels (32 half word), and vcache tile 1 will store 32-63, and etc. (see Figure 7).




```
#ifdef WARM_CACHE
__attribute__((noinline))
static void warmup(short int * A, short int * B, int H)
{
  for (int i = __bsg_id*CACHE_LINE_WORDS; i < H*H;
            i += bsg_tiles_X*bsg_tiles_Y*CACHE_LINE_WORDS) {
      asm volatile ("lw x0, %[p]" :: [p] "m" (A[i*2]));
      asm volatile ("sw x0, %[p]" :: [p] "m" (B[i*2]));
  }
  bsg_fence();
}
#endif
```

Figure 7. (a) (up) Vcache tile allocation of pixels to align with the same column cores. Each 16-word line size can store 32 short bits. (b) (down) code segment for warm_cache and vcache assignment.

In this cache assignment, the core can access the blurred pixel in the same column vcache whereas the filter will need to be accessed from the 2 left or right columns for the middle cores and 14 columns apart for the side cores. However, we expect just 1 additional horizontal hop to the second next column for the middle cores and 4 additional horizontal hops for the side cores. Also, due to

the high traffic in X direction, the horizontal hop will have minimal detrimental effect on runtime. We hypothesize that DRAM utilization should be near zero, and expect an increase in vcache utilization; moreover, we expect further drop in DRAM load stall.

We observed a 0.77% usage of DRAM, significant decrease of 97.82% compared with optimization#2 benchmark. Core utilization increased 8.05x compared with baseline benchmark. Total instruction roughly stayed the same as in optimization #2 (see Table 7 below).

| Attempt | Warm Cache | DRAM Utilization | Vcache Utilization | Core Utilization | Runtime | Total Instruct. | Total Core Stall | DRAM Load Stall |
|---|---|---|---|---|---|---|---|---|
| Optim#3 w/o filter preload | Yes | 0.77% | 15.69% | 68.44% | 3,580 | 267,593 | 122,661 | 43,432 |
| Baseline | | | | + 8.05x | -93.4% | -33.3% | -97.34% | -98.03% |
| Optim#2 | | -97.82% | | +38.01% | -22.54% | -0.06% | -54.81% | -58.88% |

Table 7. Optimization #3 build on optimization #2 (with padding and MLP), now with warm cache, but without filter preload. Expect near zero DRAM utilization and further decrease in DRAM load stall. Performance compared with baseline benchmark as well as Optimization #2.

We also preload our filter pixels to the same column vcache as well to test our theory. As expected, the performance roughly stays the same; but due to more load to cache, the total instructions increased (see Table 8 below).

| Attempt | Warm Cache | DRAM Utilization | Vcache Utilization | Core Utilization | Runtime | Total Instruct. | Total Core Stall | DRAM Load Stall |
|---|---|---|---|---|---|---|---|---|
| Optim#3 w/ filter preload | Yes | 3.14% | 15.99% | 69.40% (+ 1.4%) | 3,515 (-1.82%) | 267,837 (+244) | 117,323 (-4.35%) | 39,341 (-9.42%) |

Table 8. Optimization #3 with filter preload. Expect marginal improvement on core utilization as well as DRAM load stall. Performance compared with baseline benchmark as well as Optimization #3 without filter preload.

### 3.5. OUTCOME COMPARISON AND VISUALIZATION

Figure 8 below shows the comparison of four implementations (baseline unoptimized, Optimize #1 with padding, optimization #2 with MLP, and optimization #3 without filter preload) for core utilization, total core stall, total runtime, and DRAM load stall. For the parallelism algorithm implemented, we observed linear improvement of core utilization (Figure 8(a)); runtime decrease

and plateaued around 3,500 (Figure 8(b)); exponential decrease in DRAM load stall (Figure 8(c)); and similar trend for tremendous total stall (Figure 8(d)).

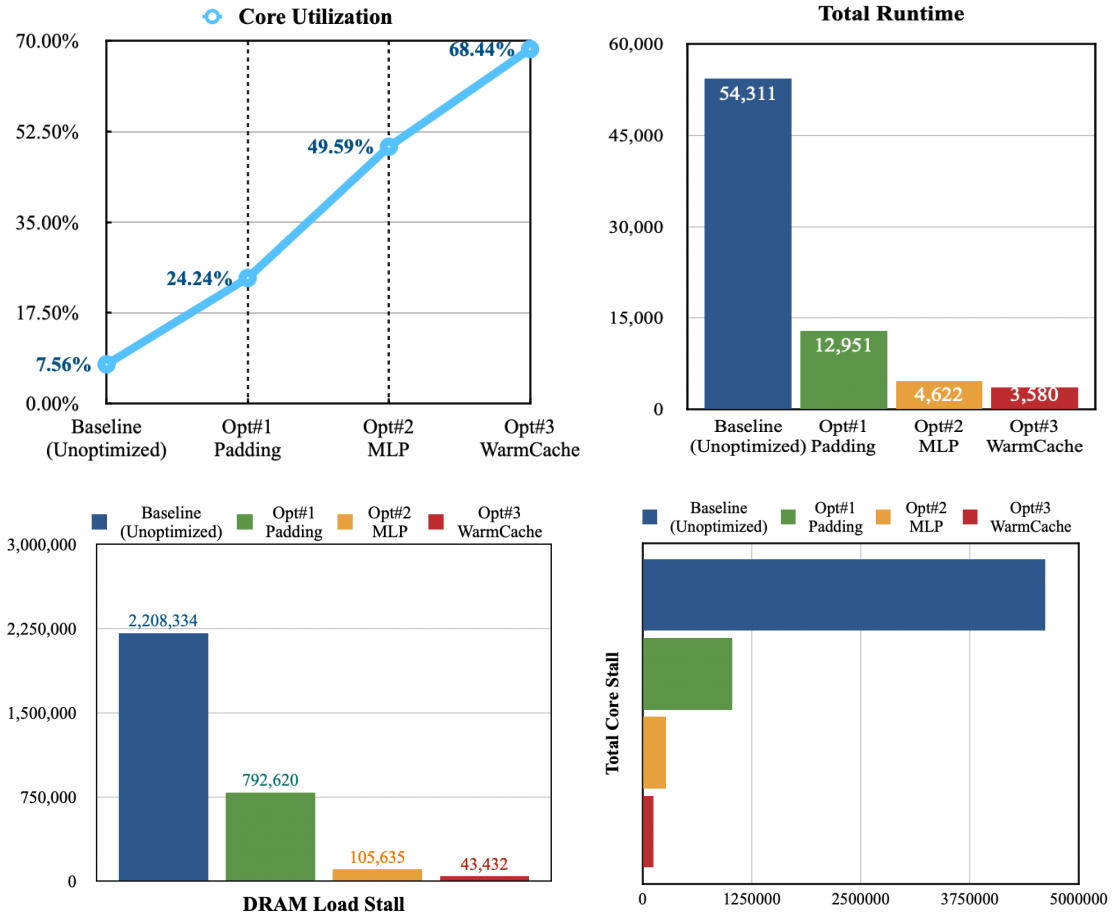



Figure 8 Comparison of four implementations (baseline unoptimized, Optimize #1 with padding, optimization #2 with MLP, and optimization #3 without filter preload) for (a) (upper left) core utilization, (b) (upper right) total runtime, (c) (lower left) DRAM load stall, and (d) (lower right) total core stall.

## 4. SUMMARY AND FUTURE WORK

In this project, we implemented the classic image processing algorithm Gaussian Blur on HammerBlade manycore architecture with different implementation schemes. We fully used the HammerBlade flexible vcache locality and tried several techniques to optimize the algorithm such as memory-level parallelism.

This project can be further extended in several areas:

- Extend the HammerBlade to 32x16 (two pod) tiles and compare the performance. This work didn't include these two configurations due to the limited project time and much

larger simulation time on these two configurations. With 32x16 tiles, the runtime and energy can be much more accurate than current estimation.

- Further optimize the performance of these two algorithms in remote load latency hiding.
- Implement other algorithms and find the bottleneck of HammerBlade architecture and make rtl modifications.

## References


[1] "HammerBlade Manycore Overview," Google Docs. Accessed: Mar. 05, 2024. [Online]. Available: https://docs.google.com/document/d/1wpdx0FykCyIAL3VdJEBz0tK-aQyChW0TKdHfbIXQJQI/edit?usp=embed_facebook

[2] "Gaussian Blur," *Wikipedia*. Nov. 27, 2023. Accessed: Mar. 08, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Gaussian_blur