# Latte: Locality Aware Transformation for High-Level Synthesis

Jason Cong, Peng Wei, Cody Hao Yu, Peipei Zhou*
University of California Los Angeles, Computer Science
{cong, peng.wei.prc, hyu, memoryzpp}@cs.ucla.edu

***Abstract*—In this paper we classify the timing degradation problems using four common collective communication and computation patterns in HLS-based accelerator design: scatter, gather, broadcast and reduce. These widely used patterns scale poorly in one-to-all or all-to-one data movements between off-chip communication interface and on-chip storage, or inside the computation logic. Therefore, we propose the Latte microarchitecture featuring pipelined transfer controllers (PTC) along data paths in these patterns. Furthermore, we implement an automated framework to apply our Latte implementation in HLS with minimal user efforts. Our experiments show that Latte-optimized designs greatly improve the timing of baseline HLS designs by 1.50x with only 3.2% LUT overhead on average, and 2.66x with 2.7% overhead at maximum.**

## I. Introduction

Field-programmable gate arrays (FPGAs) have gained popularity in accelerating a wide range of applications with high performance and energy efficiency. High-level synthesis (HLS) tools, including Xilinx Vivado HLS [1] and Intel OpenCL [2], greatly improve FPGA design feasibility by abstracting away register-transfer level (RTL) details. With HLS tools, a developer is able to describe the accelerator in C-based programming languages without considering many hardware issues such as clock and memory controller, so the accelerator functionality can be verified rapidly. Furthermore, the developer can rely on HLS pragmas that specify loop scheduling and memory organization to improve the performance. In particular, kernel replication is one of the most effective optimization strategies to reduce the overall cycle latency and improve resource utilization. However, as reported in previous work [3, 4, 5, 6], the operating frequency of a scaled-out accelerator after place and route (P&R) usually drops, which in the end diminishes the benefit from kernel replication.

Fig. 1 illustrates the frequency degradation for a broad class of applications (details in Section IV). Each dot point shows frequency and corresponding resource utilization for an application with a certain processing element (PE) number. The (black dashed) trend line characterizes the achieved frequency under certain resource usage. On average, HLS generated accelerators sustain a 200 MHz on 30% resource usage. However, the frequency drops to 150 MHz when usage increases to 74% (shown in two triangle markers). An extreme case is when dot A runs at as low as 57 MHz when using 88% resource.

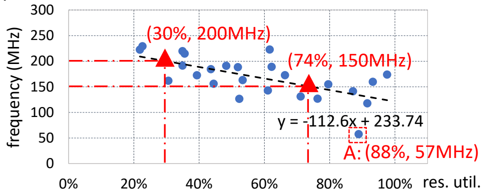


*Fig. 1:* Frequency vs. area: Freq decreases as design size scales out.

* First author. The name Latte comes from Locality Aware Transformation for high-level synThEsis.

We investigate such cases and spot the *Achilles' heel* in HLS design that has attracted less attention—in particular, one-to-all or all-to-one data movement between off-chip DRAM interface and on-chip storage logic (BRAM or FF), or inside the computation logic. Using the terminology from message passing interface (MPI) [7], we introduce four collective communication and computation patterns: scatter, gather, broadcast and reduce. They are used in most, if not all, accelerators. Different from MPI, the four patterns in HLS are in the context of **on-chip data movement**, instead of movement between servers. We observe that the on-chip data path delay in these patterns scales up when the design size increases, but HLS tools do not estimate the interconnect delay correctly or make a conscientious effort to control or cap the growth of long interconnect delays at HLS level.

A common solution to long critical path is to insert registers in the datapath in the RTL [8], logic synthesis or physical synthesis phase. However, it requires nontrivial efforts in buffering at the RTL level, which calls for a high-level solution. Also, prior work in systolic array applications [6, 9, 10, 11] and compilers [12] feature neighbor-to-neighbor interconnect in tightly coupled processing units and eliminate global interconnect. However, classic systolic array requires the application to have a regular data dependency graph, which limits the generality of the architecture. Moreover, neighbor-to-neighbor register buffering introduces logic overhead to each processing unit and incurs non-negligible overhead in the total area.

To address the above-mentioned challenges in low-level buffer insertion, generality and non-negligible area overhead, in this paper we propose the Latte microarchitecture. In data paths of the design, Latte features pipelined transfer controllers (PTC), each of which connects to only a set of PEs to reduce critical path. Intrinsically, Latte is applicable to general applications as long as the patterns occur. In addition, to improve the resource efficiency, we also explore the design choices of Latte in the number of PTC inserted, and offer performance/area-driven solutions. We implement Latte in HLS and automate the transformation for PTC insertion, which eases the programming efforts. In summary, this paper makes the followings contributions:

- Identifying four common collective communication and computation patterns: scatter, gather, broadcast and reduce in HLS that cause long critical paths in scaled-out designs.
- Latte microarchitecture featuring customizable pipelined transfer controllers to reduce critical path.
- An end-to-end automation framework that realizes our HLS-based Latte implementation.

Our experiments on a variety of applications show that the Latte-optimized design improves timing of the baseline HLS design by 1.50× with 3.2% LUT overhead on average, and 2.66× with 2.7% overhead at maximum.

## II. Motivation and Challenges

In this section we use a common practice accelerator design template shown in Fig. 2 to illustrate the low operating frequency in scaled-out designs generated by HLS tools. The `app` defines an accelerator that has input buffer `local_in` and output buffer `local_out`. In each iteration, it reads in `BUF_IN_SIZE` data (line 13) from off-chip to on-chip buffers, processes in `NumPE` kernels (line 14, 26), and then writes to off-chip from on-chip buffers (line 15). Here, double buffer optimization (A/B buffers) is applied to overlap off-chip communication and computation. Loop unroll (lines 23-26) and local buffer partitioning (lines 6-9) are applied to enable PE parallel processing.

In the remainder of the section, we summarize the design patterns from the corresponding microarchitecture in Fig. 3(a) and analyze the root cause of the critical path.

```
1  #define BUF_IN_PER_PE BUF_IN_SIZE/NumPE
2  #define BUF_OUT_PER_PE BUF_OUT_SIZE/NumPE
3  void app(int data_size,
4  int *global_in, int *global_out) {
5    // local buffer
6    int local_in_A[NumPE][BUF_IN_PER_PE];
7    int local_in_B[NumPE][BUF_IN_PER_PE];
8    int local_out_A[NumPE][BUF_OUT_PER_PE],
9    int local_out_B[NumPE][BUF_OUT_PER_PE];
10   for (int i = 0; i < data_size/BUF_IN_SIZE+1; i++) {
11     // double buffer
12     if (i % 2 == 0) {
13       buffer_load(local_in_A, global_in+i*BUF_IN_SIZE);
14       buffer_compute(local_in_B, local_out_B);
15       buffer_store(global_out+i*BUF_OUT_SIZE, local_out_A);
16     }
17     else {
18       buffer_load(local_in_B, global_in+i*BUF_IN_SIZE);
19       buffer_compute(local_in_A, local_out_A);
20       buffer_store(global_out+i*BUF_OUT_SIZE, local_out_B);
21 } } }
22 void buffer_compute(int** local_in, int** local_out) {
23   for (int i=0; i<NumPE; i++) {
24   #pragma HLS unroll
25     // kernel replication
26     PE_kernel(local_in[i], local_out[i]);}
27 }
```

*Fig. 2:* HLS accelerator design template.

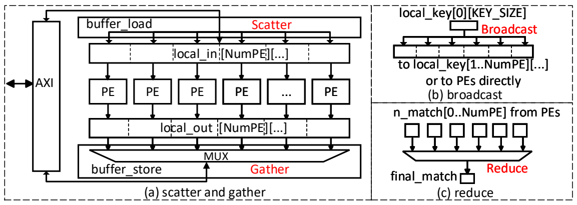


*Fig. 3:* Accelerator microarchitecture.

**One-to-all scatter.** In Fig. 3(a), `buffer_load` function is executed to read in data from DRAM using AXI protocol. As shown in Fig. 4, a common way to do this in HLS is either using `memcpy` (line 2) or in a fully pipelined loop (lines 3-6) to enable burst read. We observe that when we increase `NumPE`, the HLS report gives a constant estimated clock period for `buffer_load`, which is not the case in real layout. First, `local_in` is partitioned for parallel PE processing. Each partitioned bank (BRAM or FF) is routed to close to the corresponding PE logic, which results in scattered distribution of local buffers. We show the layout of a scatter pattern in a real application in Fig. 5(a). The yellow area highlights on-chip input buffers which span the whole chip. The white arrows show the wires connecting AXI read data port (with high fan-out) to buffers. Since HLS optimistically estimates the function delay without considering wire delay and schedule data from the AXI read port to one BRAM bank every clock cycle, the highlight wire is supposed to switch every clock cycle, and this is one cause of the critical path.

**All-to-one gather.** Fig. 3(a) shows the `buffer_store` module connecting partitioned buffer banks and the AXI write port. In order to select the data from a particular bank in one cycle, A NumPE-to-1 multiplexer (MUX) is generated. We highlight `buffer_store` in violet and MUX logic in yellow in an accelerator layout shown in Fig. 5(b). Similarly, long wires from partitioned storage banks to the AXI port through distributed MUX are the cause of long interconnect delay.

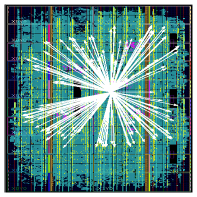
**One-to-all broadcast.** As distinct PEs span a large area, they incur long wires to broadcast data to computation logic directly (`bc_in_compute`), e.g., matrix A is broadcast to multiplier-accumulators in matrix-multiplier [13]) or to local copies of shared data within each PE (`bc_by_copy`), e.g., Advanced Encryption Standard (AES) [14] broadcasts a shared key to all processing elements that perform encryption tasks independently.

**All-to-one reduce.** Reduce is a common operation that returns a single value by combining an array of input. One example is the string matching application KMP to count the number of a certain word found in a string, where different string matching engines need to accumulate their results to get the final count.

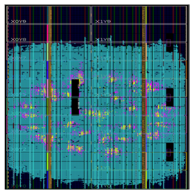
We show architecture of broadcast and reduce in Fig. 3(b)(c) and baseline code in Fig. 6. Layout of broadcast wires are similar to those in scatter and reduce as in gather patterns.

```
1  void buffer_load(int local_in[NumPE][], int* global_in) {
2    // memcpy(local_in, global_in, BUF_IN_SIZE);// burst read
3    for(int i = 0; i < NumPE; i++)
4      for(int j = 0; j < BUF_IN_PER_PE; j++) { // for each PE
5      #pragma HLS pipeline II = 1
6      local_in[i][j] = global_in[i*BUF_IN_PER_PE + j];}
7  }
8  void buffer_store(int* global_out, int local_out[NumPE][]) {
9    memcpy(global_out, local_out, BUF_OUT_SIZE);
10   // for loop (similar to buffer_load, not shown)
11 }
```

*Fig. 4:* HLS baseline buffer load and store.



(a) Scatter pattern



(b) Gather pattern

*Fig. 5:* Layout of accelerator architecture.

The four patterns are common and appear in most accelerator designs. As shown in Table I, we have implemented several accelerators from a variety of domains of applications and reported location of the critical path in the baseline designs. Except NW and VITERBI, where critical paths lie in the computation PEs, all the other designs have critical paths that result from the four patterns.

*TABLE I:* Benchmarks and *Achilles's heel* patterns in baseline designs.

| Benchmark | Domain | Scatter | Gather | Broadcast | Reduce |
|---|---|---|---|---|---|
| AES | Encryption | ✓⋆ | ✓⋆ | ✓ | |
| FFT | Signal | ✓ | ✓⋆ | | |
| GEMM | Algebra | ✓ | ✓⋆ | ✓⋆ | |
| KMP | String | ✓ | | ✓ | ✓⋆ |
| NW | Bioinfo. | ✓ | ✓ | | |
| SPMV | Algebra | ✓ | ✓⋆ | ✓ | |
| STENCIL | Image | ✓ | ✓⋆ | ✓ | |
| VITERBI | DP | ✓ | ✓ | | |

Checkmark ✓ represents the design has the pattern.
A star ⋆ represents that a critical path lies in the pattern.
For broadcast, GEMM uses bc_in_compute while others use bc_by_copy.

```
// broadcast_in_compute omit here due to space limit
// broadcast_by_copy defined
void bc_by_copy(int local_key[NumPE][], int* global_key) {
  memcpy(local_key[0], global_key, KEY_SIZE);// to 1st copy
  for(int j = 0; j < KEY_SIZE; j++){
  #pragma HLS pipeline II = 1
   for(int i = 1; i < NumPE; i++){
   #pragma HLS unroll
     // 1st copy to the rest
     local_key[i][j] = local_key[0][j];}}
}
// each element in int* n_match is from a PE
void reduce(int &final_match, int n_match[NumPE]) {
  for(int i = 0; i < NumPE; i++){
  #pragma HLS pipeline II = 1
   final_match += n_match[i];}
}// other reduction operations are similar
```

*Fig. 6:* HLS baseline broadcast and reduce.

## III. Latte Microarchitecture

In order to reduce the wire delay in the critical paths in the patterns while keeping the computation throughput, i.e., not changing NumPE, we introduce the pipelined transfer controller (PTC), the main component of the Latte microarchitecture in the data path.

### A. Pipelined Transfer Controller (PTC)

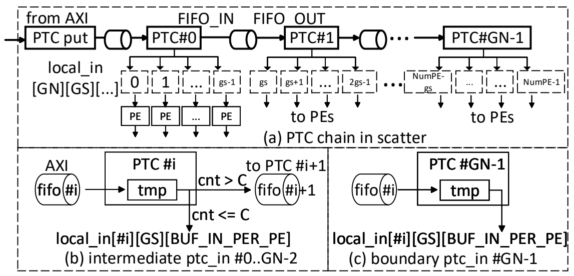


*Fig. 7:* Microarchitecture of PTC in scatter.

Fig. 7(a) shows the microarchitecture of PTC chains in a scatter pattern. PTCs are chained in a linear fashion through FIFOs, and each PTC connects to a local set of buffers in PEs to constrain the wire delay. We denote the local set size as group size GS and number of sets as group number GN. The corresponding HLS implementation is also presented in Fig 8. To access local_in from different sets in parallel, we first redefine it as local_in[GN][GS][BUF_IN_PER_SIZE](line 2). PTCs are chained using FIFOs (hls::stream), and a dataflow pipeline (line 5) is applied to enable function pipeline in the PTC modules defined below (lines 7-10). There are three types of PTCs: ptc_put, intermediate and boundary ptc_in. In ptc_put (lines 13-17), it reads in data from AXI in a fully pipelined loop and writes to the first FIFO. Intermediate ptc_in reads in data from the previous PTC through FIFO. It first writes to local set of PE buffers and then writes the rest to the next FIFO (lines 18-33), as shown in Fig 7(b). Similarly, Fig 7(c) shows boundary ptc_in, where it reads data from the last FIFO and writes all the data to a local set of PE buffers.

In addition, we show the microarchitecture of the PTC chain in gather pattern in Fig. 9(a). Similarly, there are three types of PTC: boundary ptc_out (Fig. 9(b)), intermediate ptc_out (Fig. 9(c)) and ptc_get. The modules are similar to those in scatter with a difference in the opposite data transfer direction.

The microarchitectures of PTC broadcast and reduce patterns are similar to those for scatter and gather, which we leave out due to the space limitation. PTC is somewhat similar to the idea of multi-cycle communication in the MCAS HLS system [15].

```
 1 #include <hls_stream.h>
 2 int local_in[GN][GS][BUF_IN_PER_PE]; // redef.
 3 void PTC_load(
 4   int local_in[GN][GS][], int* global_in) {
 5 #pragma HLS dataflow
 6   hls::stream<int> fifo[GN];// FIFOs, in Fig. 7a
 7   ptc_put(global_in, fifo[0]);
 8   for(int i = 0; i < GN-2; i++){
 9     ptc_in(fifo[i], fifo[i+1], local_in[i], GN-1-i);}
10   ptc_in(fifo[GN-1], local_in[GN-1]);
11 }
12 void ptc_put(int* global_in, stream<int> &fifo){
13   for (int i=0; i<NumPE; i++)
14     for(int j = 0; j < BUF_IN_PER_PE; j++){
15 #pragma HLS pipeline
16       fifo << global_buf[i*BUF_IN_PER_PE+j];}
17 }
18 void ptc_in( // #0..GN-2 ptc_in, in Fig. 7b
19 stream<int>&fifo_in, stream<int> &fifo_out,
20 int local_set[GS][BUF_IN_PER_PE], int todo){
21   int i, j, k; int tmp;
22   for(i= 0; i < GS; i++){ // to local first
23     for(j = 0; j < BUF_IN_PER_PE; j++){
24       tmp = fifo_in.read();
25       local_set[i][j] = tmp;
26   } }
27   for(k=0; k < todo; k++) // to next ptc
28   for(i= 0; i < GS; i++){
29     for(j = 0; j < BUF_IN_PER_PE; j++){
30       tmp = fifo_in.read();
31       fifo_out.write(tmp);
32   } }
33 }
```

*Fig. 8:* Code snippet of HLS implementation for PTC in scatter.

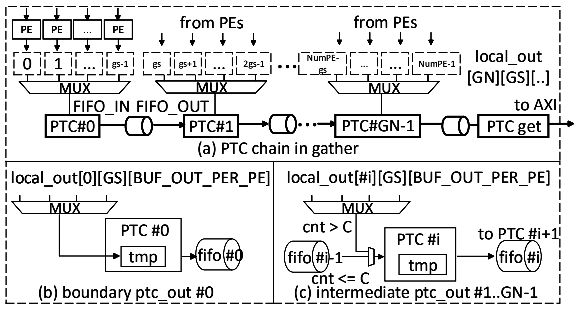


*Fig. 9:* Microarchitecture of PTC in gather.

It is possible to manually implement the Latte microarchitecture in HLS. However, the implementation expands over 260 lines of code (LOC), which is 10× more than the baseline code shown in Fig. 4 and Fig. 6. To relieve the burden of manual programming effort in implementing Latte, we provide an automation framework that reduces the 260-LOC implementation to simply a few directives.

### B. Automation Framework

We implement a semiautomatic framework to make use of Latte by having a user-written HLS kernel with simple Latte pragmas. The Latte pragma indicates the on-chip buffer with the pattern to be optimized. For example, Fig. 10 presents an example of using a Latte pragma to enable scatter pattern with PTCs for the on-chip buffer from Fig. 2.

```
#pragma latte scatter var="local_in_B"
int local_in_B[NumPE][BUF_IN_PER_PE];
```

*Fig. 10:* An example of Latte pragma.

After parsing the kernel code with pragmas, we perform code analysis by leveraging the ROSE compiler infrastructure [16] to identify the kernel structure, array types and sizes. Subsequently, we apply predefined HLS function templates of Latte by performing source-to-source code transformation. Corresponding optimization, such as memory partitioning, memory coalesce [17], and so forth, are applied as well. We implement a distributed runtime system that launches multiple

Amazon EC2 [18] instances for exploring the PTC group size with Vivado HLS [1] in parallel to determine the best design configuration. Note that since we only search the group size that is a divisor of the PE number, the design space is small enough to be fully explored.

## IV. Experimental Evaluation

We use Alpha Data ADM-PCIE-7V3 [19] as an evaluation FPGA board (Virtex-7 XC7VX690T) and Xilinx Vivado HLS, SDAccel 2017.2 [1] for synthesis. For each benchmark listed in Table I, we implement the baseline design and scale out $N$ times until fully utilizing the on-chip resource or failing to route. We then obtain the baseline frequency as $F$ and baseline area as $A$. Then for $N$ of an application, we choose $GS$ as divisors of $N$. For each $GS$, Latte optimizations are applied on all existing patterns, and frequency is reported as $F_{GS}$, area as $A_{GS}$. Thus, the performance ratio of the Latte optimized design and baseline is expressed as $F_{GS}/F$, performance-to-area (P2A) ratio as $\frac{F_{GS}/A_{GS}}{F/A} = \frac{F_{GS}}{F} / \frac{A_{GS}}{A}$ (in terms of latency, each PTC introduces one extra cycle, which is negligible compared to the cycle number of the original design). Latte enables design space exploration for both ratios as shown in Fig. 11 for `GEMM`. As can be seen, `GEMM` achieves the optimal performance when $GS$ is four, which has 227 MHz operating frequency with 35% LUT overhead. In addition, P2A optimal design is identified when $GS$ is 16, achieving 207 MHz with only 8% area overhead. On the other hand, we can observe the performance degradation when $GS$ decreases from four to one. The reason is that the critical path has been moved from data transfer to PEs when $GS$ is four, and further reducing the data transfer wire delay will not improve the performance. This illustrates the motivation for selecting a suitable $GS$ instead of always setting $GS$ to one.

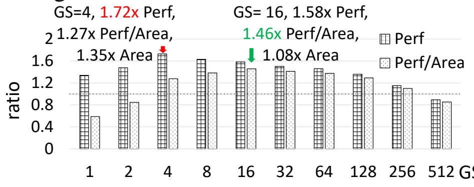


*Fig. 11:* Performance and P2A ratio in GEMM with 512 PEs.

In addition, we report the resource utilization and operating frequency for baseline designs under $N$ PEs (`ori.`) and the corresponding Latte designs with optimal P2A $GS$ (`latte`) in Table II. The Latte optimized design improves timing over baseline HLS design by 1.50× with 3.2% LUT, 5.1% FF overhead on average. For `FFT`, it even achieves 2.66× with only 2.7% LUT and 5.1% FF overhead. Even for designs such as `NW` and `VITERBI` where critical paths in baseline lie in PEs, the Latte optimized design is still beneficial. A possible reason is that the Latte design helps the placement of PEs, which helps routing within PEs. In summary, the average frequency has been improved from 120 MHz to 181 MHz.

Finally, the overall frequency to area in Latte designs are shown in Fig. 12. It achieves 200 MHz on 61% chip area, and 174 MHz on 90%, which helps greatly in frequency degradation. We also show the layout of PTCs in gather pattern in FFT with 64 PEs and 16 PTCs in Fig 13, where PTCs are connected in linear fashion and scale much better.

*TABLE II:* Baseline design vs Latte optimized design.

| Bench. | type | N / GS | LUT | FF | DSP | BRAM | Freq. |
|---|---|---|---|---|---|---|---|
| AES | ori. | 320 / | 50.4% | 17.3% | 0.1% | 76.3% | 127 |
| | latte | / 32 | 1.017 | 1.009 | 1 | 1 | 165, 1.30x |
| FFT | ori. | 64 / | 50.5% | 23.2% | 88.9% | 78.5% | 57 |
| | latte | / 4 | 1.027 | 1.056 | 1 | 1 | 152, 2.66x |
| GEMM | ori. | 512 / | 37.8% | 29.6% | 71.1% | 69.7% | 131 |
| | latte | / 16 | 1.044 | 0.962 | 1 | 1 | 207, 1.58x |
| KMP | ori. | 96 / | 5.0% | 3.0% | 0.2% | 52.3% | 126 |
| | latte | / 24 | 1.045 | 1.174 | 1 | 1 | 195, 1.54x |
| NW | ori. | 160 / | 65.1% | 50.7% | 0.0% | 78.2% | 174 |
| | latte | / 80 | 0.995 | 0.997 | 1 | 1 | 177, 1.02x |
| SPMV | ori. | 48 / | 19.1% | 11.9% | 18.9% | 93.2% | 160 |
| | latte | / 6 | 1.029 | 1.037 | 1 | 1 | 192, 1.20x |
| STENCIL | ori. | 64 / | 12.9% | 10.9% | 48.1% | 87.1% | 141 |
| | latte | / 16 | 1.094 | 1.139 | 1 | 1 | 188, 1.33x |
| VITERBI | ori. | 192 / | 72.0% | 25.7% | 10.8% | 39.3% | 155 |
| | latte | / 12 | 1.008 | 1.031 | 1 | 1 | 168, 1.08x |
| Average | ori. | / | NA | NA | NA | NA | 120 |
| | latte | / | 1.032 | 1.051 | 1 | 1 | 181, 1.50x |

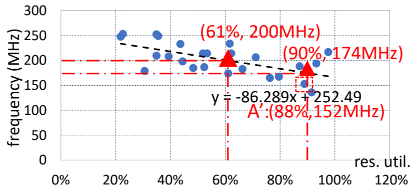


*Fig. 12:* Freq. degradation much less severe in Latte optimized designs.

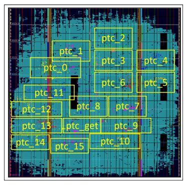


*Fig. 13:* PTC layout in FFT (N=64,GS=4,GN=16).

## V. Conclusion and Future Work

In this paper we summarize four common collective communication and computation patterns (i.e., scatter, gather, broadcast and reduce) in HLS that generate long interconnects in scaled-out design and result in degraded frequency. To achieve a high frequency, we propose the Latte microarchitecture which features pipeline transfer controllers in the four patterns to reduce wire delay. We also implement an automated framework to realize HLS-based Latte implementation with a only few lines of user-provided derivatives. Experiments show that the Latte optimized design improves frequency from 120 MHz to 181 MHz with 3.2% LUT, 5.1% FF overhead on average. Design space exploration on the full system design with a comprehensive analytical model and customizable PTC connections other than linear fashion remain as future work.

## VI. Acknowledgment

The work is supported by Intel/NSF Innovation Transition Program (InTrans CCF-1436827) awarded to the UCLA Center for Domain-Specific Computing and contributions from CDSC industrial partners Huawei, Fujitsu, NEC, and Samsung.

## References

[1] Xilinx, "Vivado HLS & SDAccel," http://www.xilinx.com/.
[2] Intel, "Intel FPGA SDK for OpenCL," http://www.altera.com/.
[3] Z. Wang *et al.*, "A performance analysis framework for optimizing OpenCL applications on fpgas," in *HPCA*, 2016.
[4] H. R. Zohouri *et al.*, "Evaluating and optimizing OpenCL kernels for high performance computing with fpgas," in *SC*, 2016.
[5] S. Wang *et al.*, "FlexCL: An analytical performance model for OpenCL workloads on flexible fpgas," in *DAC*, 2017.
[6] A. Tavakkoli and D. B. Thomas, "Low-latency option pricing using systolic binomial trees," in *FPT*, 2014.
[7] M. Snir, *MPI–the Complete Reference: the MPI core*. MIT press, 1998, vol. 1.
[8] R. Chen *et al.*, "Energy efficient parameterized fft architecture," in *FPL*, 2013.
[9] L. D. Tucci *et al.*, "Architectural optimizations for high performance and energy efficient smith-waterman implementation on fpgas using opencl," in *DATE*, 2017.
[10] E. Rucci *et al.*, "Smith-waterman protein search with opencl on an fpga," in *IEEE Trustcom/BigDataSE/ISPA*, 2015.
[11] J. Zhang, P. Chow, and H. Liu, "Cordic-based enhanced systolic array architecture for qr decomposition," *TRETS*, 2015.
[12] W. Luk, G. Jones, and M. Sheeran, "Computer-based tools for regular array design," in *Systolic Array Processors*, 1989.
[13] P. Zhou *et al.*, "Energy efficiency of full pipelining: A case study for matrix multiplication," in *FCCM*, 2016.
[14] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.
[15] J. Cong, Y. Fan, G. Han, X. Yang, and Z. Zhang, "Architecture and synthesis for on-chip multicycle communication," *TCAD*, pp. 550–564, April 2004.
[16] "ROSE Compiler Infrastructure," http://rosecompiler.org/.
[17] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Bandwidth optimization through on-chip memory restructuring for hls," in *DAC*, 2017.
[18] "Amazon ec2," https://aws.amazon.com/ec2.
[19] AlphaData, "ADM-PCIE7V3," https://www.alpha-data.com/pdfs/adm-pcie-7v3.pdf.