F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption

(2021 MICRO)

Introduction

Characters

- Complex operations on long vectors: modular arithmetic, several thousan elements
- Regular computation: all operations are known ahead of time, VLIM
- Challenging data movement: large amounts (tens of MBs) of data; encrypting data increase its size(50X); data in long vectors

Introduction

Contributions

- F1 features an explicitly managed on-chip memory hierarchy, with a heavily bank scratchpad and distributed reg files
- **F1 uses mechanisms to decoupled data movement and hide access latencies by loading data far ahead of its use**
- F1 uses new scheduling algorithms that maximize reuse and make the best out of limited memory bandwidth
- F1 used few functional units with hight throughtput that reduces the amount of data

Background

FHE programming model and operations

- element-wise
- addition (mod t)
- multiplication (mod t)
- a small set of particular vector permutations.

BGV implementation overview

Data types:
$a = a_0 + a_1 x + \ldots + a_{N-1} x^{N-1} \in R_t$
Each plaintext is encrypted into a ciphertext consisting of two polynomials of N integer coefficients modulo some $Q \gg t$. Each ciphertext polynomial is a member of R_Q.

Encrtyption and decryption:
secret key: $s \in R_Q$.To encrypt a plaintext $m \in R_t$,one samples a uniformly random $a \in R_Q$, an error (or noise) $e \in R_Q$ with small entries, and computes the ciphertext ct as
$ct = (a, b = as + te + m)$

Ciphertext $ct = (a, b)$ is decrypted by recovering $e\prime = te + m = b - as \mod Q$, and then recovering $m = e' \mod t$. Decryption is correct as long as e' does not "wrap around" modulo Q, i.e., its coefficients have magnitude less than $Q/2$.

Homomorphic operations

- addition
  $ct_0 = (a_0, b_0)$ and $ct_1 = (a_1, b_1)$
  $ct_{add} = ct_0 + ct_1 = (a_0 + a_1, b_0 + b_1)$
- multiplication
  $ct_\times = (l_2, l_1, l_0) = (a_0 a_1, a_0 b_1 + a_1 b_0, b_0 b_1)$
  $(u_1, u_0) = KeySwitch(I_2)$
  $ct_{mul} = (I_1 + u_1, I_0 + u_0)$

- permutations
  There are N automorphisms, denoted $\sigma_k(a)$ and $\sigma_{-k}(a)$ for all positive odd $k < N$. Specifically,$\sigma_k(a) : a_i -> (-1)^s a_{ik} \mod N$ for $i = 0, \ldots, N-1$
  where $s = 0$ if $ik \mod 2N < N$, and $s = 1$ otherwise.
  1.compute an automorphism on the ciphertext polynomials:
  $ct_\sigma = (\sigma_k(a), \sigma_k(b))$
  2.$ct_{perm} = (u_1, \sigma_k(b) + u_0)$where $(u_1, u_0) = KeySwitch(\sigma_k(a))$

Noise growth and management

Different operations induce different noise growth: addition and permutations cause little growth, but multiplication incurs much more significant growth.So, to a first order, the amount of noise is determined by **multiplicative depth** , i.e., the longest chain of homomorphic multiplications in the computation.

Noise forces the use of a large ciphertext modulus Q. For example, an FHE program with multiplicative depth of 16 needs Q to be about 512 bits. The noise budget, and thus the tolerable multiplicative depth,grow linearly $\log Q$

- Bootstrapping:
  strength: enable FHE computations of unbounded depth;remove noise from a ciphertext without access to the secret key
  weakness:need a large noise budget(large Q)
- Modulus switching:
  rescales ciphertexts from modulus Q to a modulus Q'.
  To execute an multiplicative depth 16, we start with a 512 bit modulus Q. Before multiplicatino,switch to a modulus that is 32 bits shorter.

Security and parameters

demension N and modulus Q
$N/logQ$ must be above a certain level for sufficient security.

Algorithmic insights and optimizations

- Fast polynomial multiplication via NTTs
- Avoiding wide arithmetic via Residue Number System(RNS)

Architectural analysis of FHE

Three input: a polynomial x (store in L residue polyniamials), two key-switch hint matrices $ksh0$,$ksh1$. Inputs and outputs are in the NTT domain; only $y[i]$ are in coefficient form.

```
1    def keySwitch(x: RVec[L],
2          ksh0: RVec[L][L], ksh1: RVec[L][L]):
3      y = [INTT(x[i],qi) for i in range(L)]
4      u0: RVec[L] = [0, ...]
5      u1: RVec[L] = [0, ...]
6      for i in range(L):
7        for j in range(L):
8          xqj = (i == j) ? x[i] : NTT(y[i], qj)
9          u0[j] += xqj * ksh0[i,j] mod qj
10         u1[j] += xqj * ksh1[i,j] mod qj
11     return (u0, u1)
```

Listing 1: Key-switch implementation. RVec is an N-element vector of 32-bit values, storing a single RNS polynomial in either the coefficient or the NTT domain.

Computation vs. data movement

- L^2 NTTs,$2L^2$multiplications,$2L^2$additions of N-element vectors
- In RNS, the rest of a homomorphic multiplication is $4L$ multiplications and $3L$ additions

$L = 16, N = 16K$
each RNS polynimial is 64KB, each polynimial is 1MB, each ciphertext is 2MB,key switch hints is 32MB.
key switchinging demand 10TB/s of memory andwidth.

- Performance requirement:
  (1) decouples data movement from computation, as demand misses during frequent key-switches would tank performance
  (2) implements a large amount of on-chip storage (over 32 MB in our example) to allow reuse across entire homomorphic operations
- Functionality requirements:
  Programmable FHE accelerators must support a wide range of parameters, both N (polynomial/vec-tor sizes) and L (number of RNS polynomials, i.e., number of 32-bit prime factors of Q). While N is generally fixed for a single program, L changes as modulus switching sheds off polynomials.

F1 ARCHITECTURE

Vector processing with specialized functional units

FUs process vectors of configurable length N using a fixed number of vector lanes E.

- 128 lanes
- N from 1024 to 16384
- pipelined, throughput: E = 128 elements/cycle

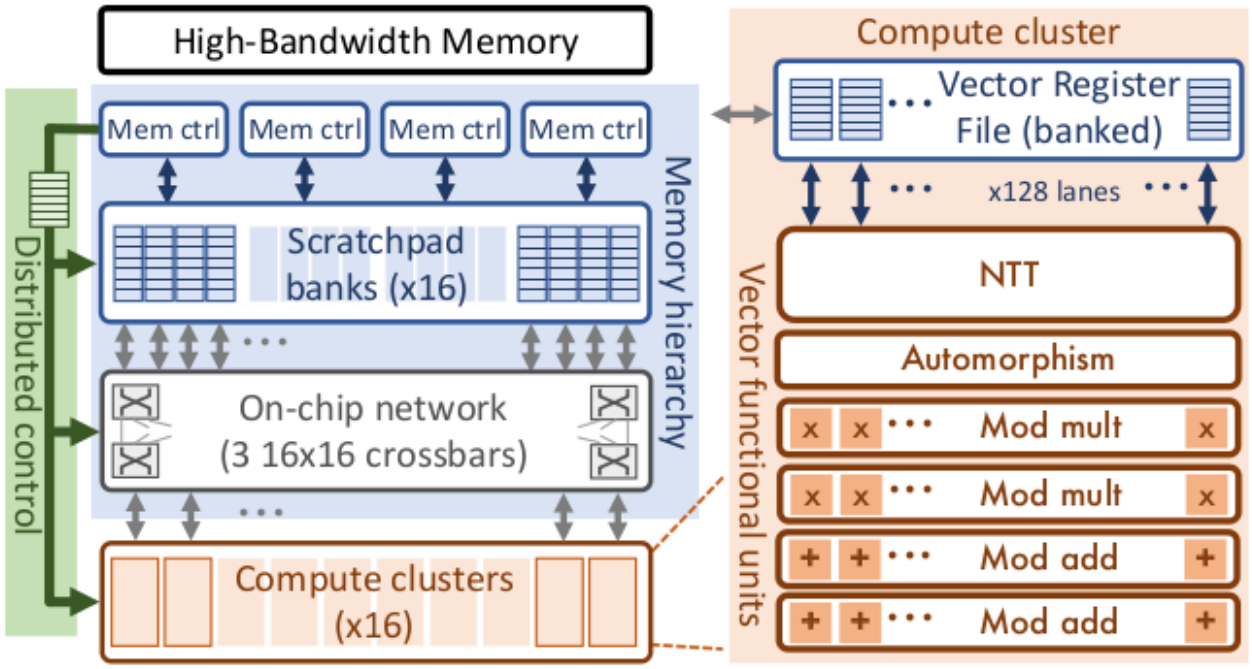


Figure 2: Overview of the F1 architecture.

Compute clusters:

- 1 NTT, 1 automorphism
- 2 multipliers
- 2 adders
- a banked register file

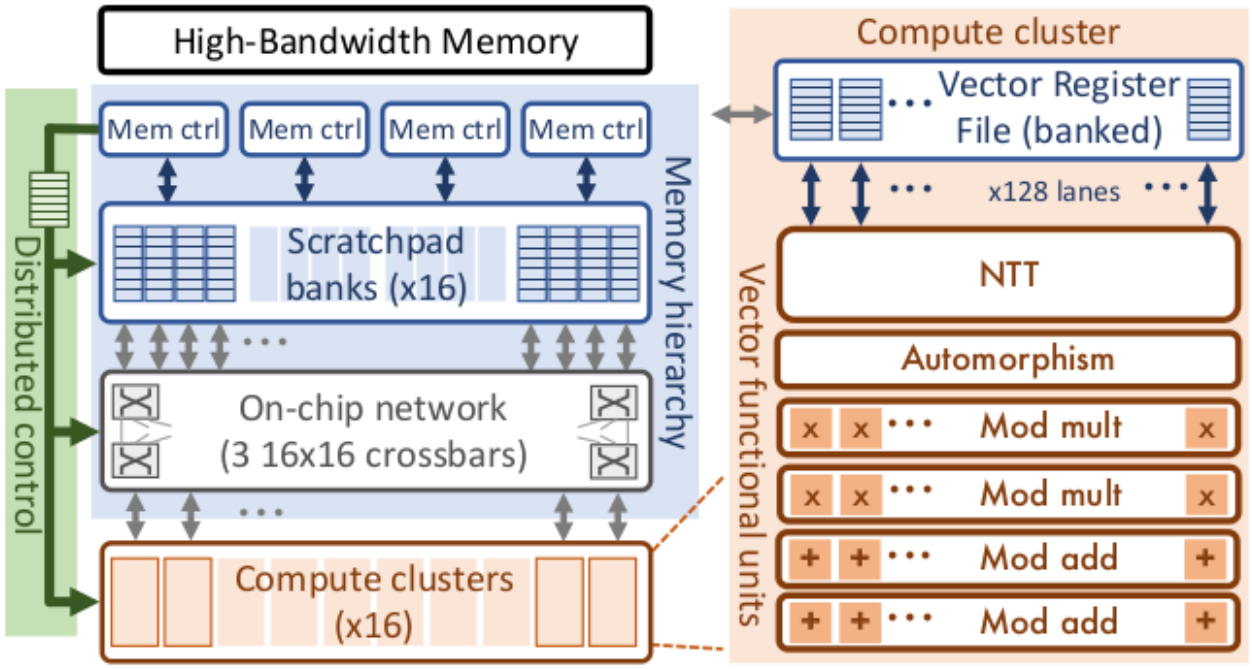


Figure 2: Overview of the F1 architecture.

Memory system:

- a large, heavily banked scratchpad (64 MB across 16 banks)
- scratchpad interfaces with both high-bandwidth off-chip memory (HBM2) and with compute clusters through an on-chip network.

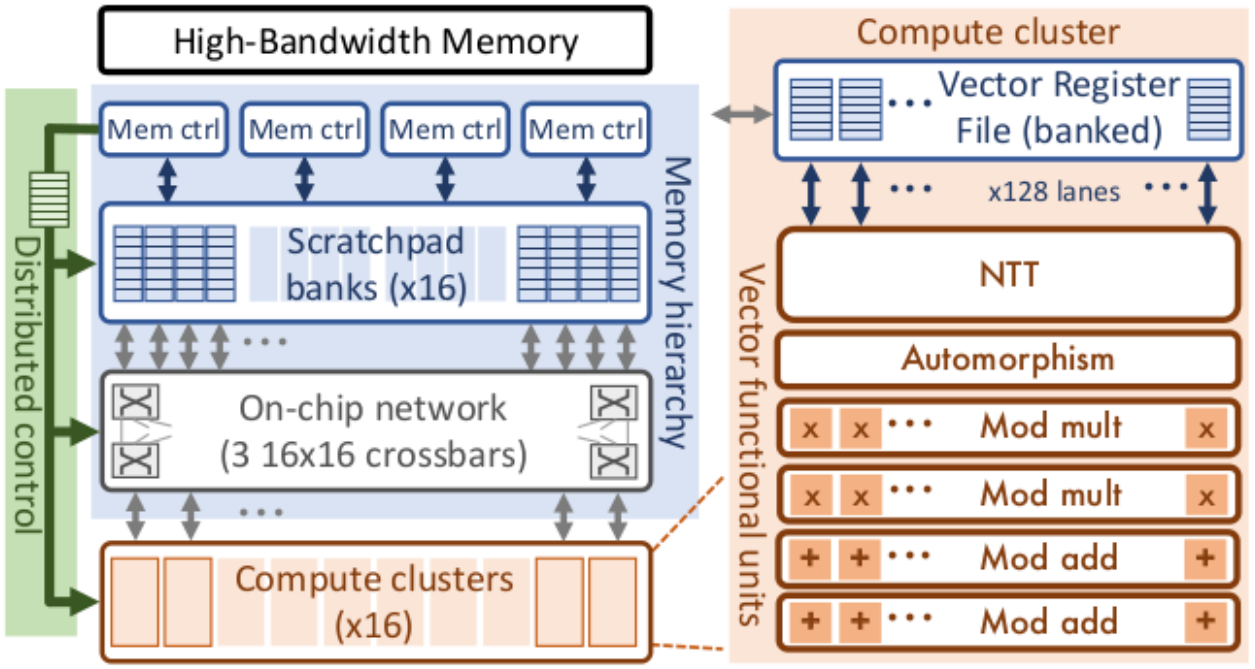


Figure 2: Overview of the F1 architecture.

Static scheduling(programs are regular):

- VLIW processors
- FUs: no stalling logic
- Memory: no conflicts
- On-chip network: use switch change configuration

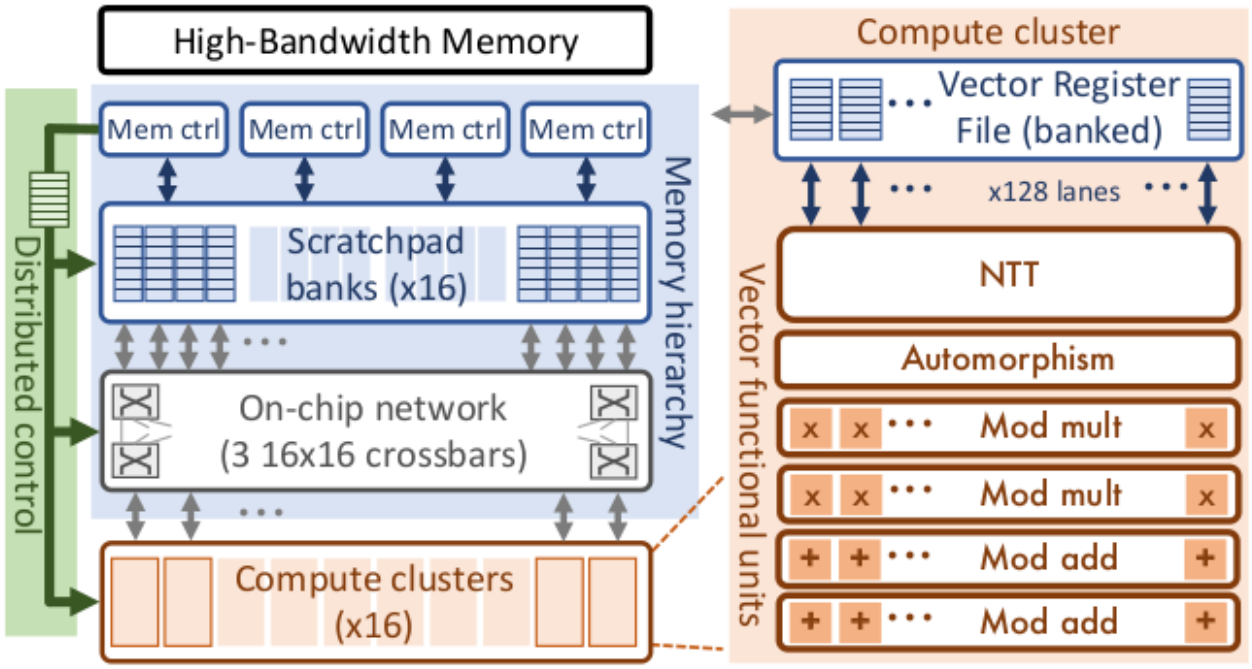


Figure 2: Overview of the F1 architecture.

Distribute control:

- independent instruction stream: programs have loops, unroll them avoid branches, and compile programs into linear sequences of instructions

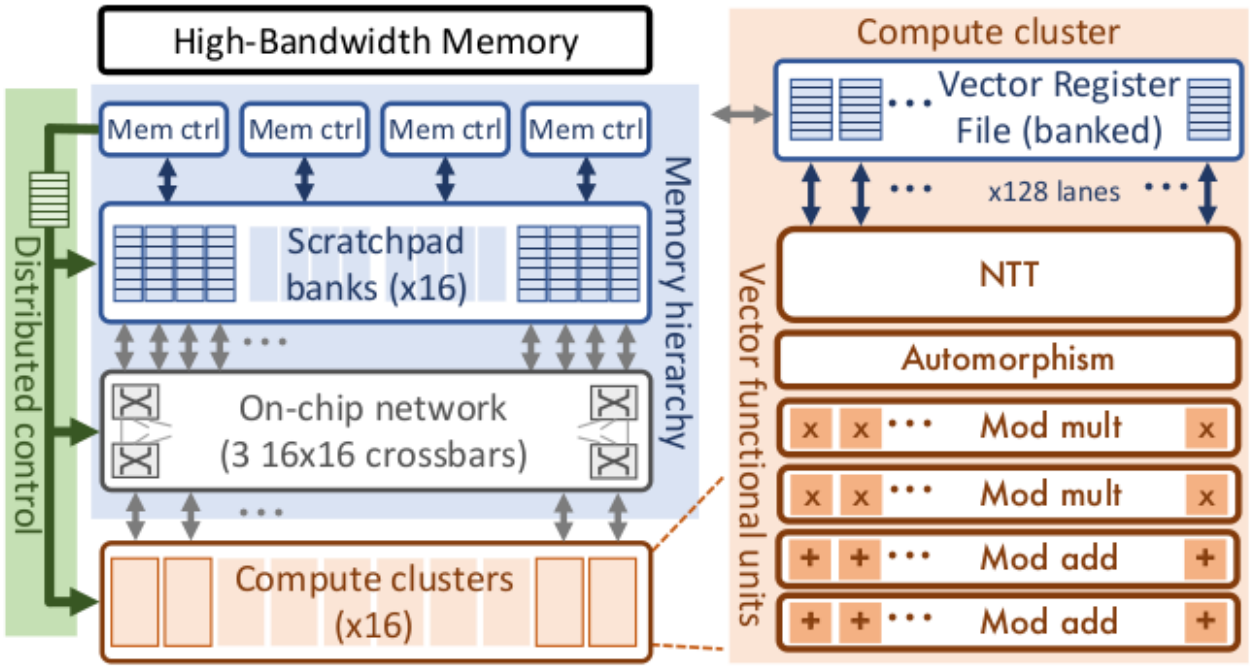


Figure 2: Overview of the F1 architecture.

F1 ARCHITECTURE

Register file design:

use an 8-banked element-partitioned register file design that leverages long vectors: each vector is striped across banks, and each FU cycles through all banks over time, using a single bank each cycle

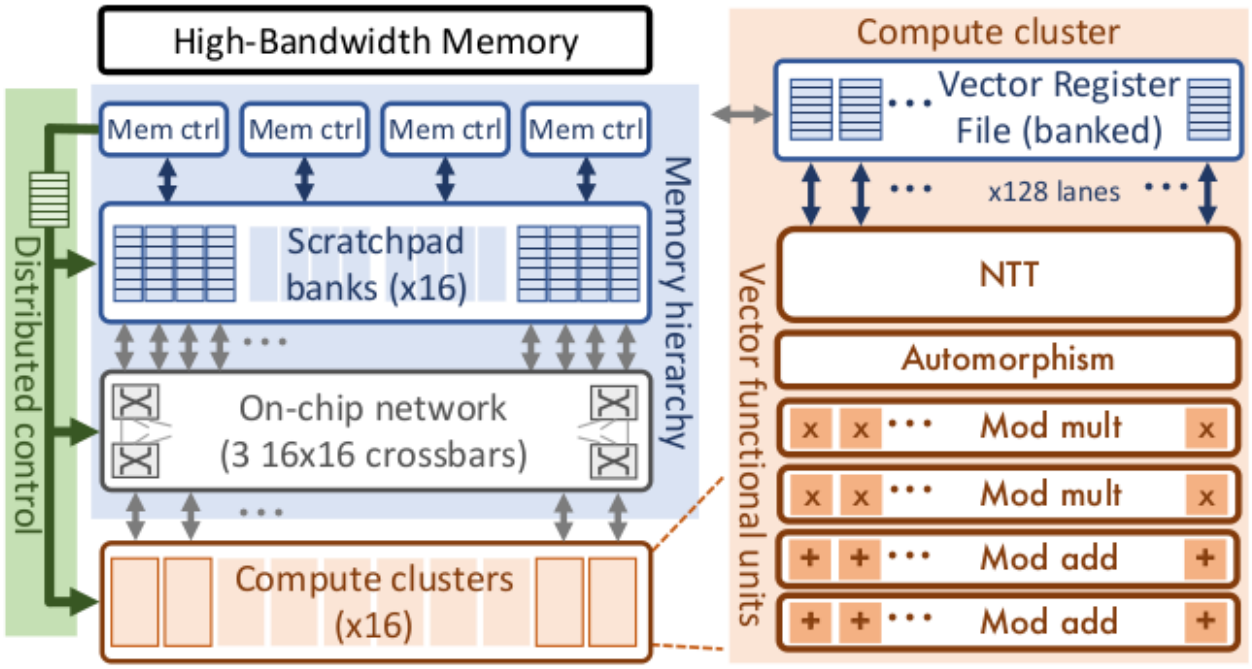


Figure 2: Overview of the F1 architecture.

SCHEDULING DATA AND COMPUTATION

- Compiler: orders high level operations to maximize reuse and translates the program into a DFG

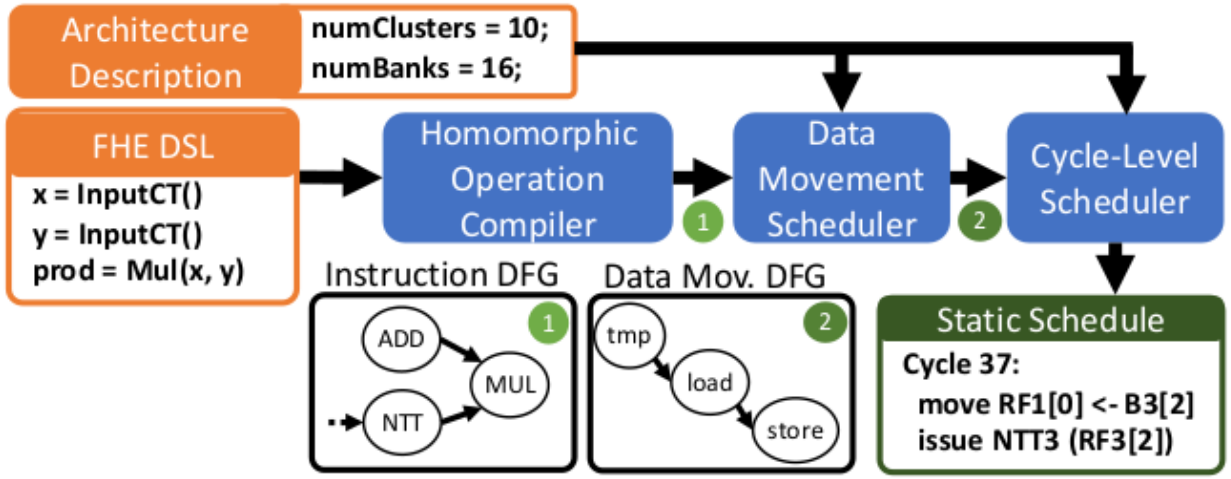


Figure 3: Overview of the F1 compiler.

- DM Scheduler: transfer between main memory andthe scratchpad to achieve decoupling and maximize reuse

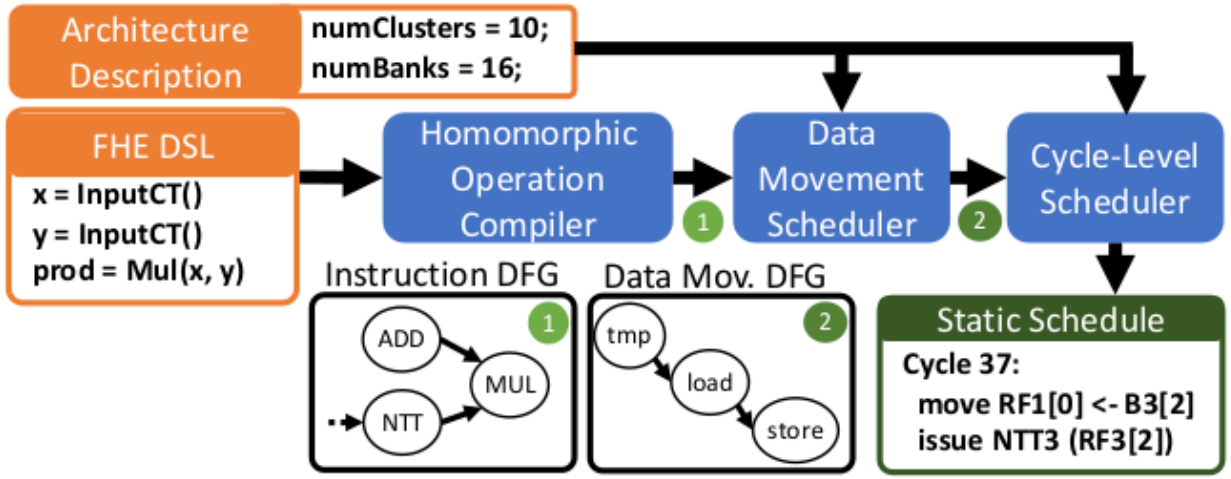


Figure 3: Overview of the F1 compiler.

- CL Scheduler: determine the exact cycles of all operations and produces the instruction strams for all components

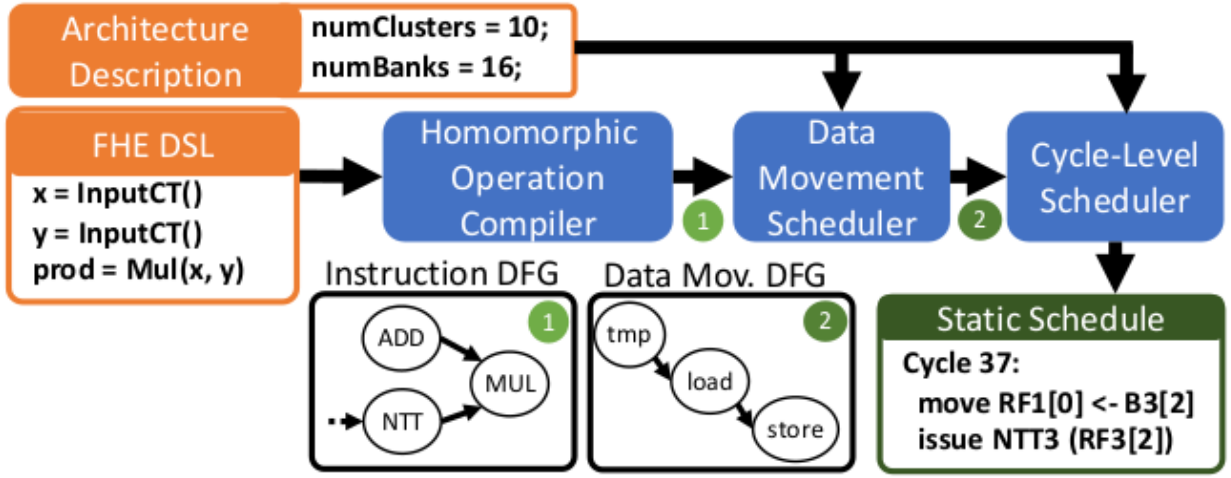


Figure 3: Overview of the F1 compiler.

Translating the program to a dataflow graph

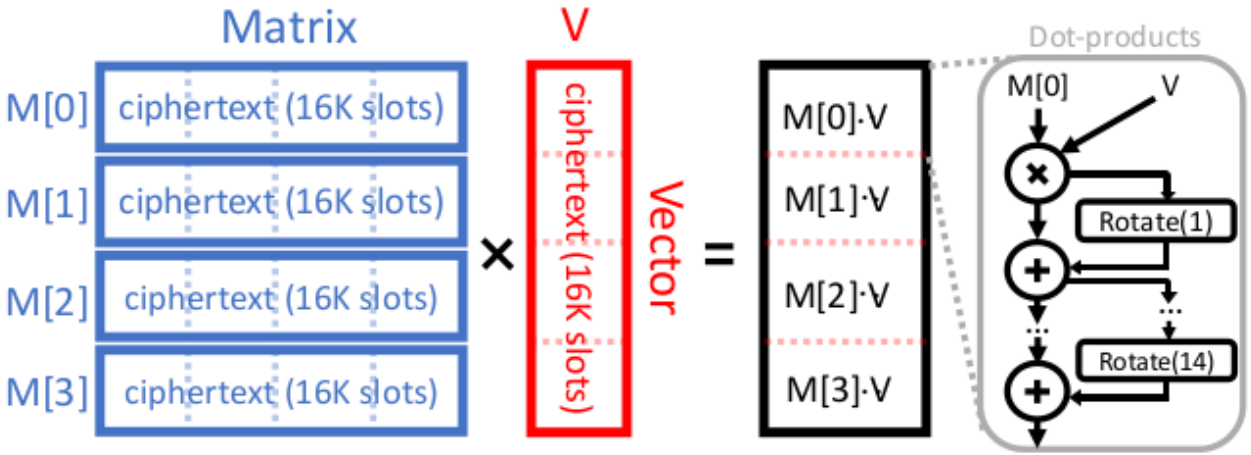


Figure 4: Example matrix-vector multiply using FHE.

```
1   p = Program(N = 16384)
2   M_rows = [ p.Input(L = 16) for i in range(4) ]
3   output = [ None for i in range(4) ]
4   V = p.Input(L = 16)
5
6   def innerSum(X):
7     for i in range(log2(p.N)):
8       X = Add(X, Rotate(X, 1 << i))
9     return X
10
11  for i in range(4):
12    prod = Mul(M_rows[i], V)
13    output[i] = innerSum(prod)
```

Listing 2: $(4 \times 16K)$ matrix-vector multiply in F1's DSL.

Compiling homomorphic operations

It clusters operations to improve reuse and translates them down to instruction.

- Ordering: maximize the reuse of key switch hints(line 8)(line 12)
- Translation: minimize the amount of instructions intermediates

Scheduling data transfers

- data transfers decoupled from computation
- minmize off-chip data transfers
- achieve good parallelism

$$p(load) = max\{p(u)|u \in users(load)\}$$

Cycle-level scheduling(constrained by its input schedule's off-chip data movement)

- distribute comptation across clusters and manage reg file and on-chip transfer
- add loads or stores in this stage
- move loads to their earliest possible issue cycle to avoid stalls on missing operands

FUNCTIONAL UNITS

Automorphism unit

how automorphism σ3 is applied to a residue polynomial with N = 16 and E = 4 elements/cycle.

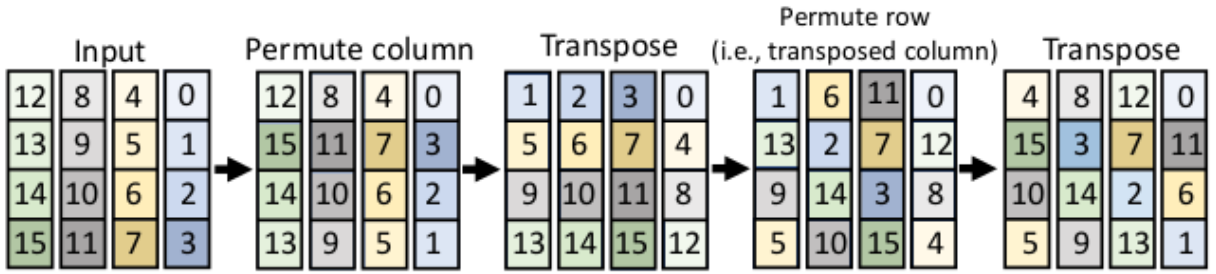


Figure 5: Applying σ_3 on an RNS polynomial of four 4-element chunks by using only permutations local to chunks.

Automorphism unit

Given a residue polynomial of $N=G \cdot E$ elements, the automorphism unit first transpose applies the column permutation to each E-element input. Then, it feeds this to a transpose unit that reads in the whole residue polynomial interpreting it as a $G \times E$ matrix, and produces its transpose $E \times G$.

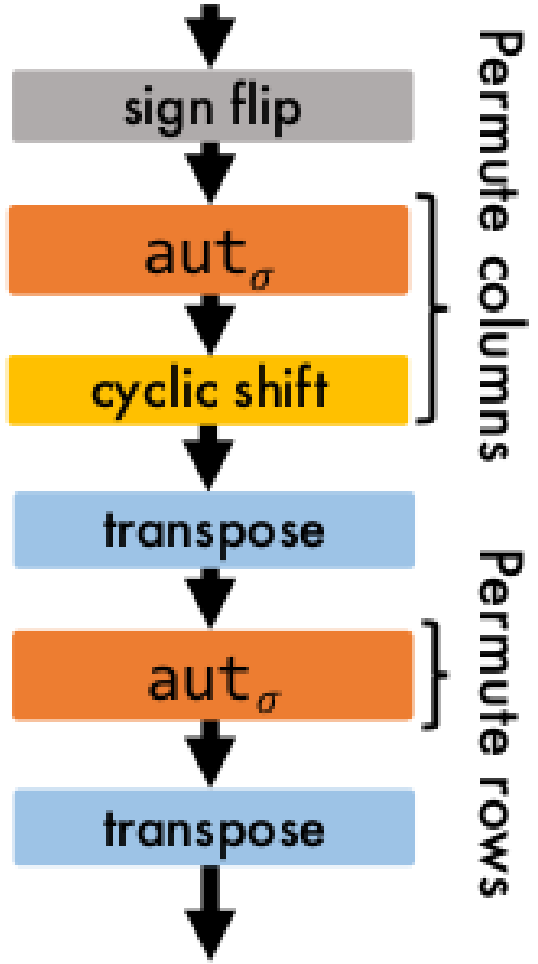


Figure 6: Automorphism unit.

Transpose unit

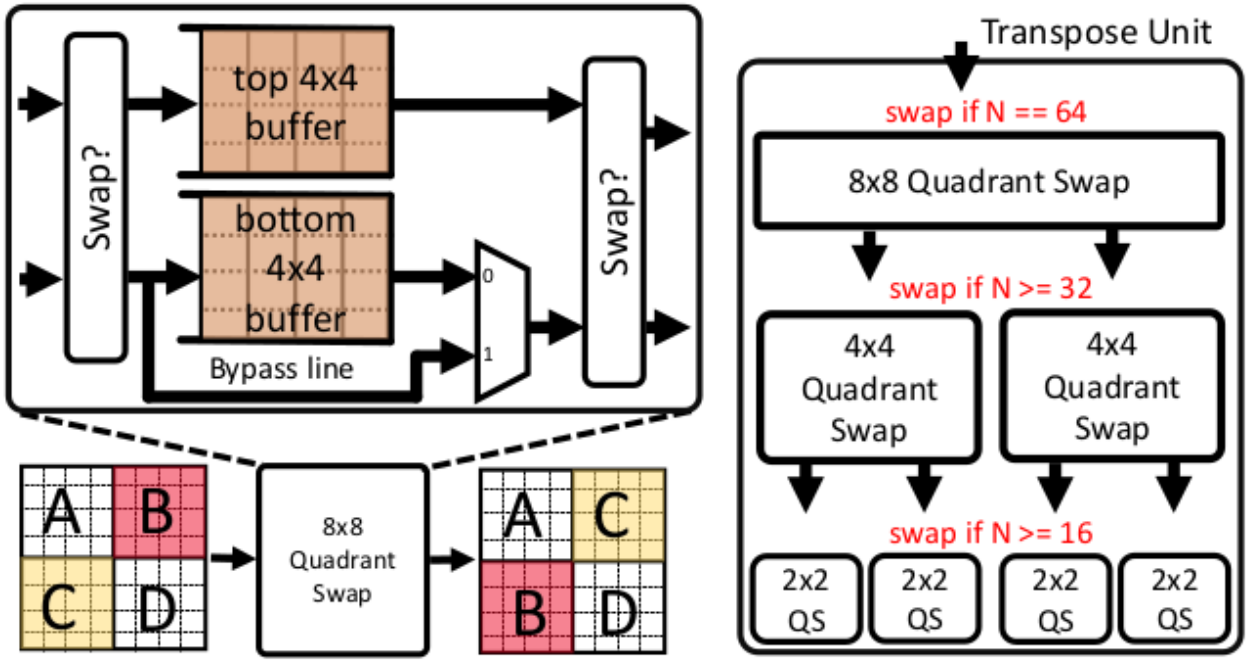


Figure 7: Transpose unit (right) and its component quadrant-swap unit (left).

Transpose unit: Our *quadrant-swap transpose* unit transposes an $E \times E$ (e.g., 128×128) matrix by recursively decomposing it into quadrants and exploiting the identity

$$\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]^T = \left[\begin{array}{c|c} A^T & C^T \\ \hline B^T & D^T \end{array}\right].$$

Four-step NTT unit

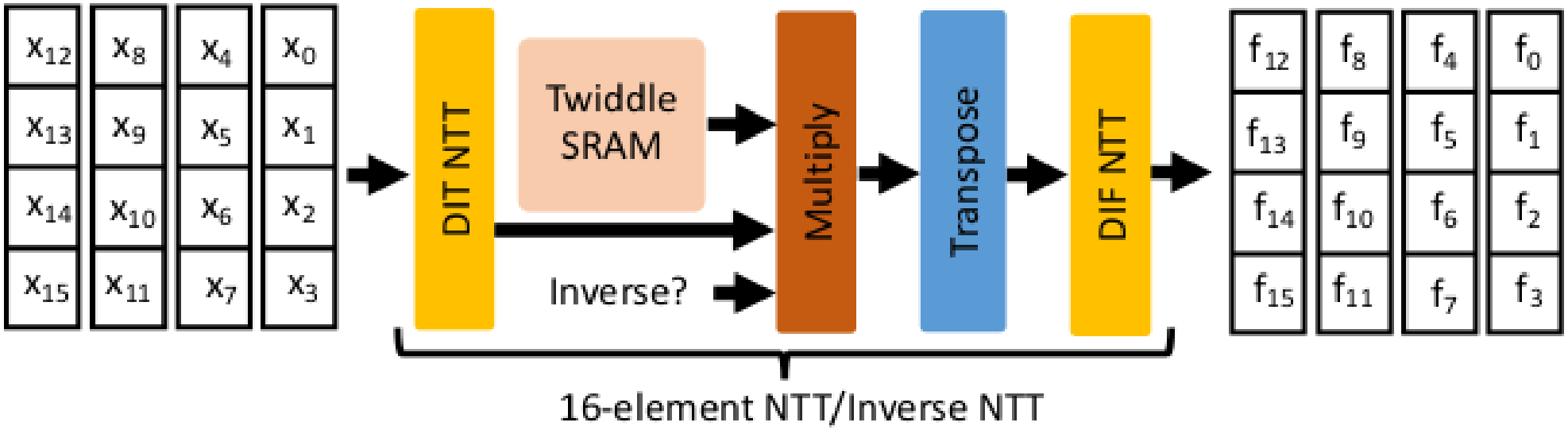


Figure 8: Example of a four-step NTT datapath that uses 4-point NTTs to implement 16-point NTTs.

Optimized modular multiplier

Multiplier	Area [μm^2]	Power [mW]	Delay [ps]
Barrett	5,271	18.40	1,317
Montgomery	2,916	9.29	1,040
NTT-friendly	2,165	5.36	1,000
FHE-friendly (ours)	1,817	4.10	1,000

Table 1: Area, power, and delay of modular multipliers.

F1 IMPLEMENTATION

Component	Area [mm^2]	TDP [W]
NTT FU	2.27	4.80
Automorphism FU	0.58	0.99
Multiply FU	0.25	0.60
Add FU	0.03	0.05
Vector RegFile (512 KB)	0.56	1.67
Compute cluster (NTT, Aut, 2× Mul, 2× Add, RF)	3.97	8.75
Total compute (16 clusters)	**63.52**	**140.0**
Scratchpad (16×4 MB banks)	48.09	20.35
3×NoC (16×16 512 B bit-sliced [58])	10.02	19.65
Memory interface (2×HBM2 PHYs)	29.80	0.45
Total memory system	**87.91**	**40.45**
Total F1	**151.4**	**180.4**

Table 2: Area and Thermal Design Power (TDP) of F1, and breakdown by component.

EXPERIMENTAL METHODOLOGY

- Modeled system:
  a cycle-accurate simulator to execute F1 programs
  activity-level energies from RTL synthesis to produce energy breakdowns
- Benchmarks:
  Logistic regression: uses the HELR algorithm:256 features, 256 samples, depth L =16
  Neural network:LoLa-MNIST,LoLa-CIFAR
  DB Lookup:A BGV-encrypted query string is used to traverse an encrypted key-value store and return the corresponding value.

- Bootstrapping:
  BGV: Sheriff and Peikert's algorithm
  CKKS: non-packed CKKS bootstrapping
- Baseline systems:
  F1 with a CPU system running the baseline programs (a 4-core, 8-thread, 3.5 GHz Xeon E3-1240v5)

EVALUATION

Performance

Benchmarks

Execution time (ms) on	CPU	F1	Speedup
LoLa-CIFAR Unencryp. Wghts.	1.2×10^6	**241**	5,011×
LoLa-MNIST Unencryp. Wghts.	2,960	**0.17**	17,412×
LoLa-MNIST Encryp. Wghts.	5,431	**0.36**	15,086×
Logistic Regression	8,300	**1.15**	7,217×
DB Lookup	29,300	**4.36**	6,722×
BGV Bootstrapping	4,390	**2.40**	1,830×
CKKS Bootstrapping	1,554	**1.30**	1,195×
gmean speedup			5,432×

*LoLa's release did not include MNIST with encrypted weights, so we reimplemented it in HELib.

Table 3: Performance of F1 and CPU on full FHE benchmarks: execution times in milliseconds and F1's speedup.

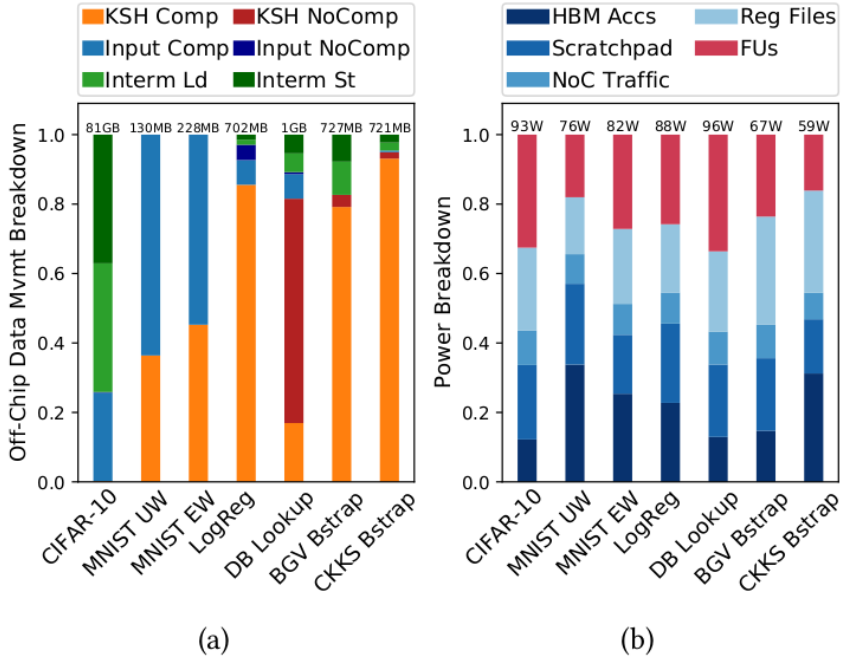
Microbenchmarks

	$N = 2^{12}$, $\log Q = 109$			$N = 2^{13}$, $\log Q = 218$			$N = 2^{14}$, $\log Q = 438$		
	F1	vs. CPU	vs. HEAX$_\sigma$	**F1**	vs. CPU	vs. HEAX$_\sigma$	**F1**	vs. CPU	vs. HEAX$_\sigma$
NTT	**12.8**	17,148×	1,600×	**44.8**	10,736×	1,733×	**179.2**	8,838×	1,866×
Automorphism	**12.8**	7,364×	440×	**44.8**	8,250×	426×	**179.2**	16,957×	430×
Homomorphic multiply	**60.0**	48,640×	172×	**300**	27,069×	148×	**2,000**	14,396×	190×
Homomorphic permutation	**40.0**	17,488×	256×	**224**	10,814×	198×	**1,680**	6,421×	227×

Table 4: Performance on microbenchmarks: F1's reciprocal throughput, in nanoseconds per ciphertext operation (lower is better) and speedups over CPU and HEAX$_\sigma$ (HEAX augmented with scalar automorphism units) (higher is better).

Architectural analysis

Data movement,Power consumption,Utilization over time



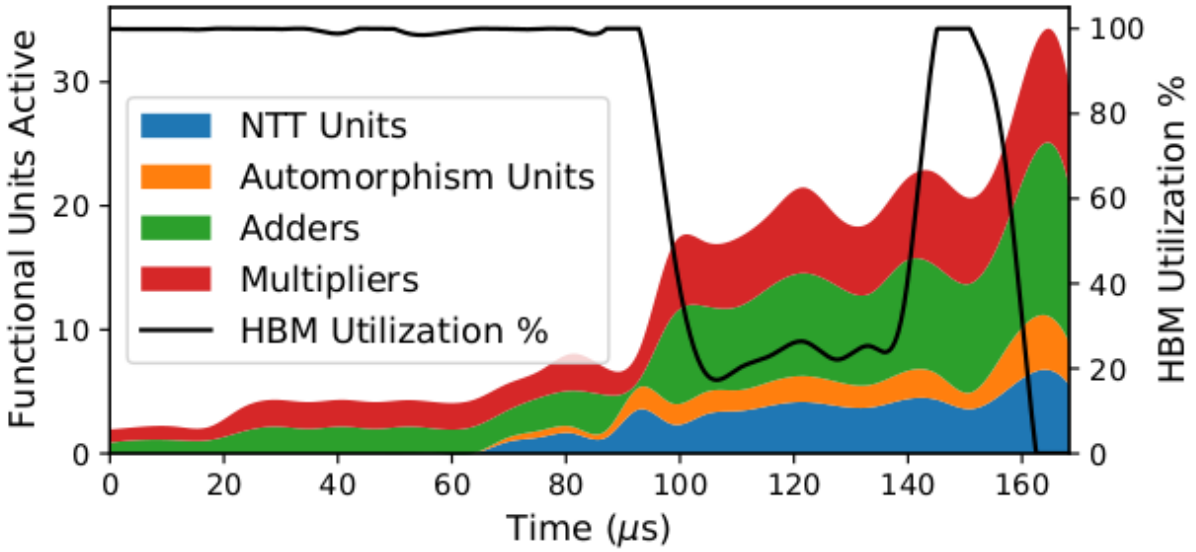


Figure 10: Functional unit and HBM utilization over time for the LoLa-MNIST PTW benchmark.

Sensitivity studies

Benchmark	LT NTT	LT Aut	CSR
LoLa-CIFAR Unencryp. Wghts.	3.5×	12.1×	—*
LoLa-MNIST Unencryp. Wghts.	5.0×	4.2×	1.1×
LoLa-MNIST Encryp. Wghts.	5.1×	11.9×	7.5×
Logistic Regression	1.7×	2.3×	11.7×
DB Lookup	2.8×	2.2×	—*
BGV Bootstrapping	1.5×	1.3×	5.0×
CKKS Bootstrapping	1.1×	1.2×	2.7×
gmean speedup	2.5×	3.6×	4.2×

*CSR is intractable for this benchmark.

Table 5: Speedups of F1 over alternate configurations: LT NTT/Aut = Low-throughput NTT/Automorphism FUs; CSR = Code Scheduling to minimize Register Usage [37].

Scalablity

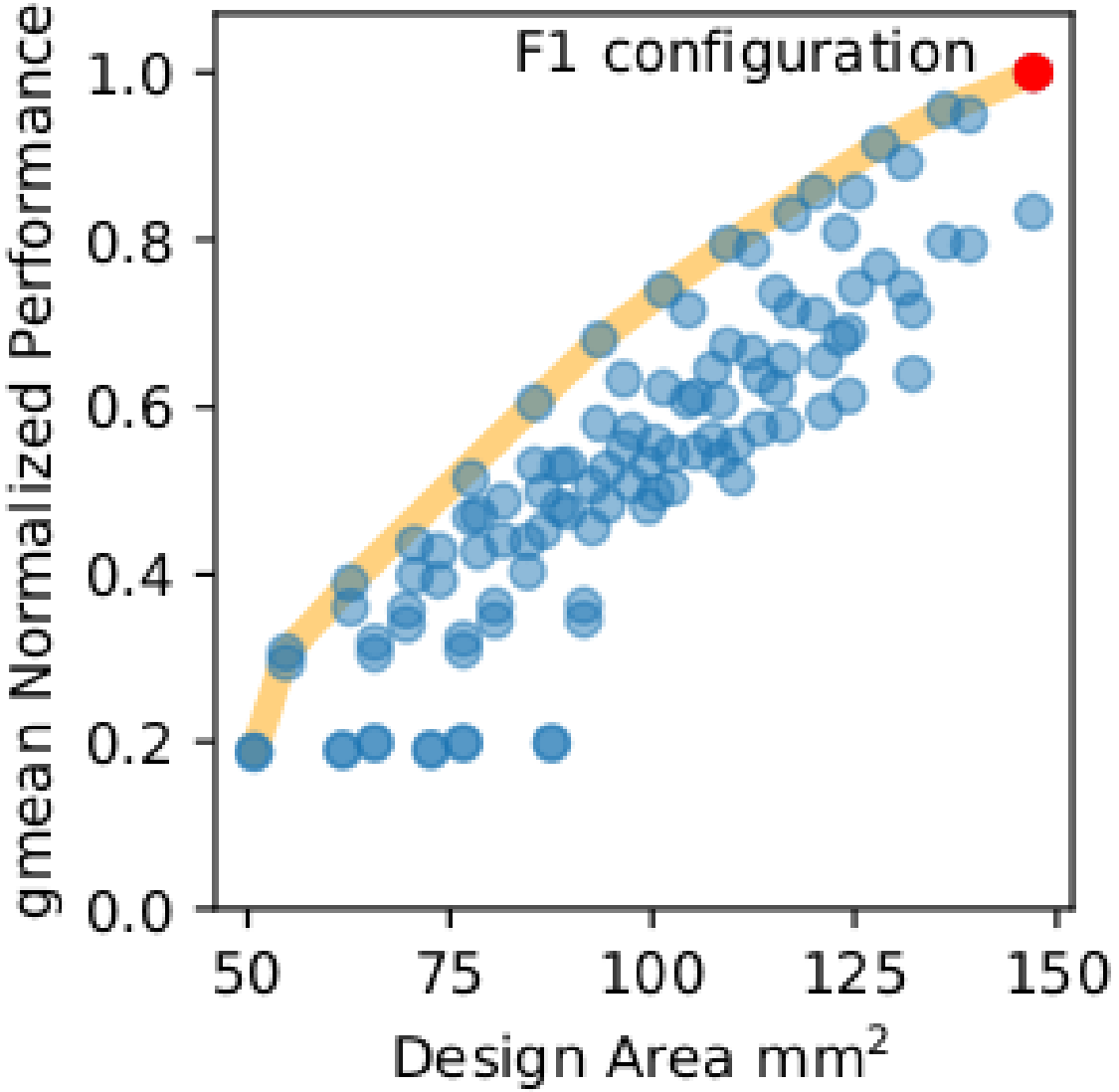


Figure 11: Performance vs. area across F1 configurations.