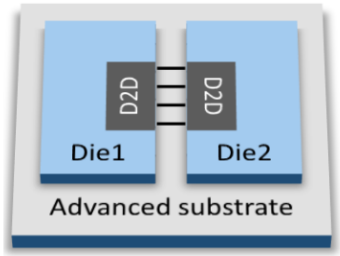
Gemini

Mapping and Architecture Co-exploration for Large-scale DNN Chiplet Accelerators

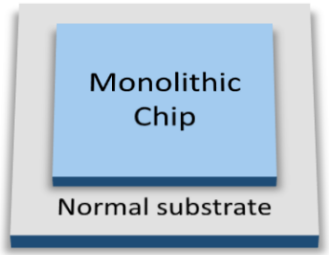
管仁阳 2025.1.3

Contents

The Trade-off introduced by Chiplet



(a) 2-Chiplet Chip



(b) Monolithic Chip

Trade-offs Introduced by Chiplet	
Advantages	Higher Yield
	Larger Scale
	Heterogeneous Integration
	Reuse for Multiple Chips
Disadvantages	Worse Performance
	Worse Power
	D2D Area Costs
	Higher Packaging Costs

Affected by arch

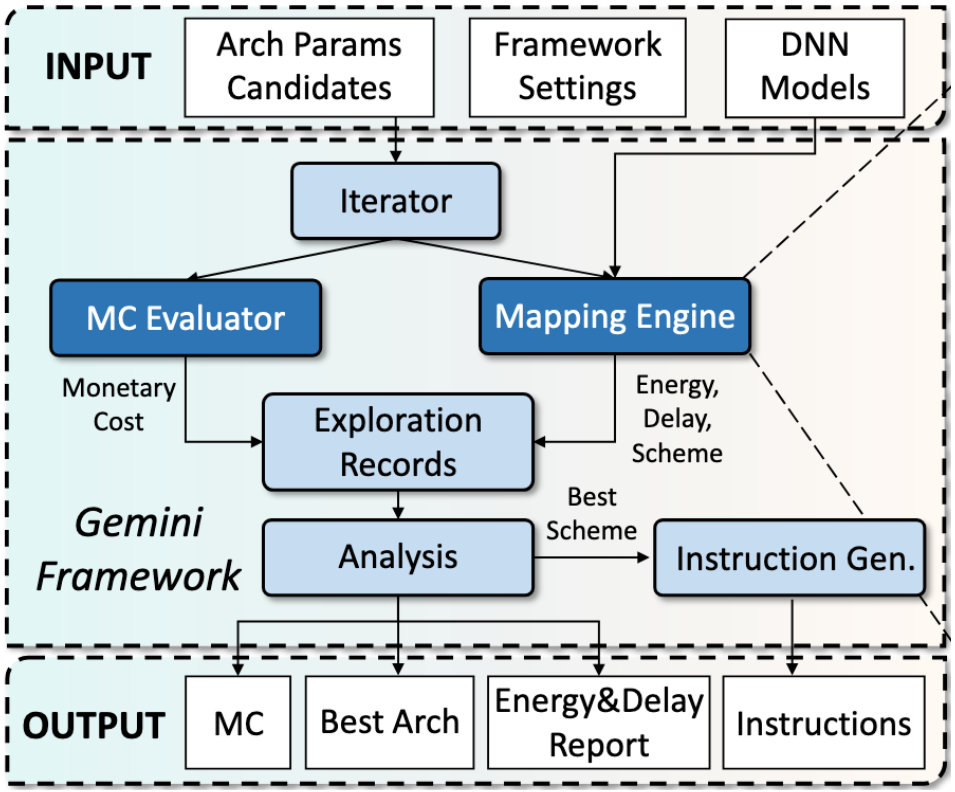
Affected by both arch and mapping

- Energy Consumption
- Performance
- Monetary Cost

In the chiplet era, solely considering the silicon chip area is inadequate for evaluating a chip's MC.

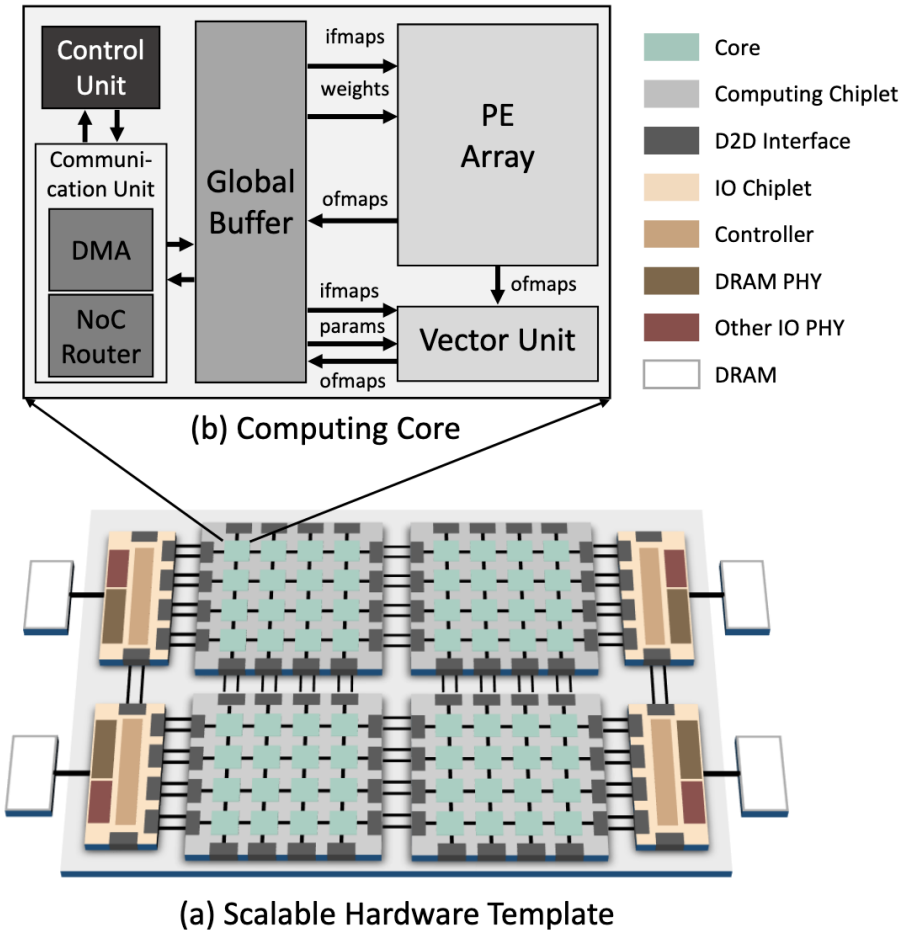
Factors such as yield and packaging costs also need to be taken into account

Search Framework for Architecture



- The inputs to the arch-mapping co-optimization is
  - Architecture Specification
  - Problem Setting (Model Information)
- Search Algorithm
  - Grid Search for architecture specification
  - Simulated Annealing for best mapping
- Evaluator
  - MC Evaluator: Get the monetary cost of the arch
  - Mapping Engine: Search for the best dataflow of the problem on the architecture
- Search Target: $MC^{\alpha} \times E^{\beta} \times D^{\gamma}$, (E:Energy, D:Delay)

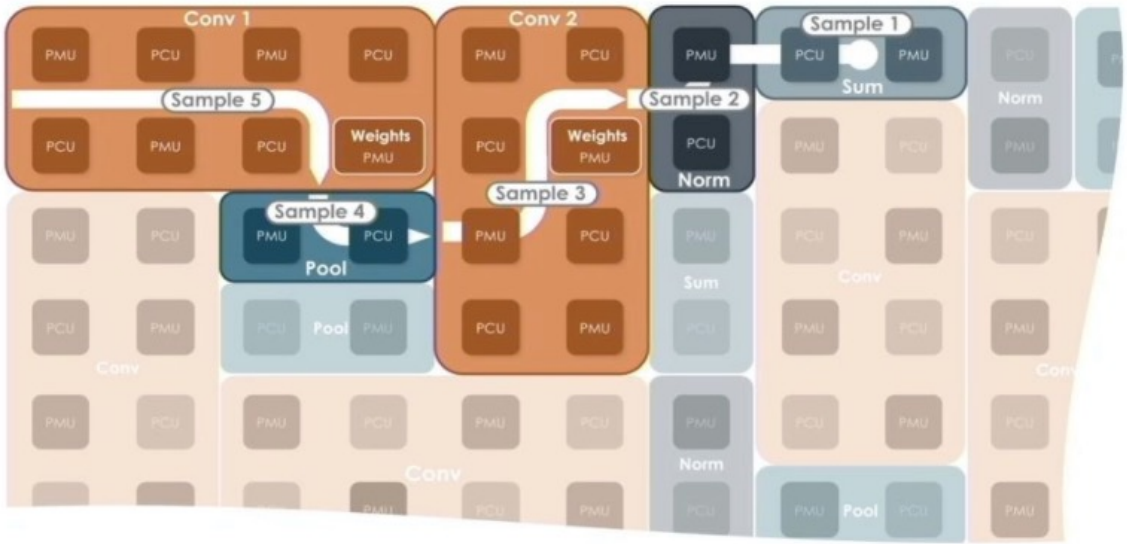
Configurable Architecture Template



- Two distinct types of chiplets:
  - IO chiplets
  - Computing chiplets
- A mesh NoC interconnects all computing cores and the controllers within IO chiplets
- Inter-chiplet communication is fully **automatic and transparent**
- **Forming a large mesh** – D2D interfaces are placed around the Chiplet, whose number is equal to the number of computing cores on each side.
- Global buffer (GLB) of each core is **globally visible** in the entire accelerator.

Layer-Pipeline Encoding

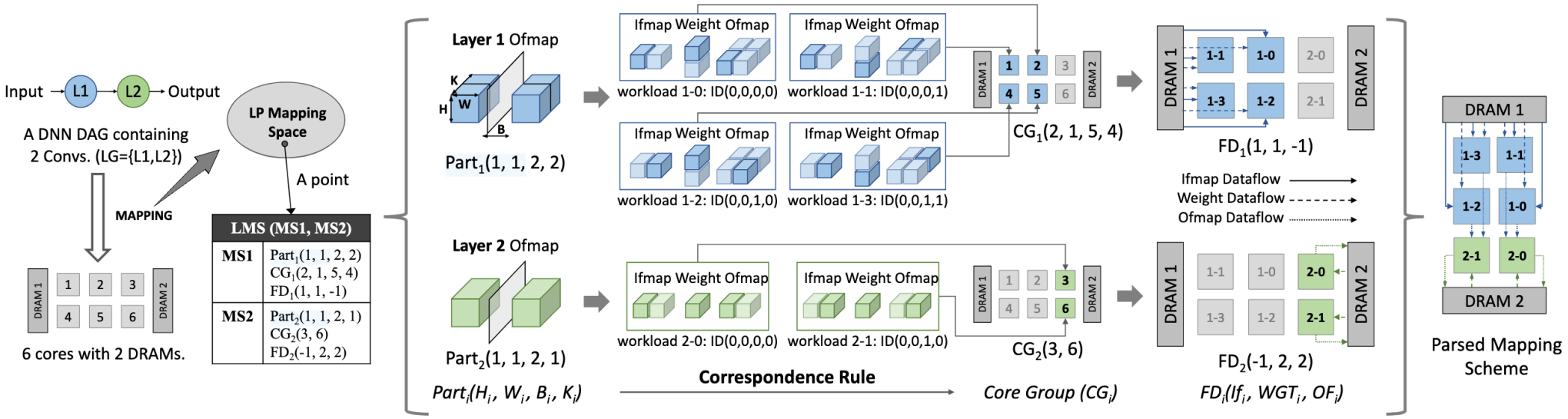
- Layer Sequential: All the compute units execute the compute graph sequentially
- Layer Pipeline: Layers are allocated to different compute unit groups and form a pipeline



- *The Layer-Pipeline scheme is suitable for Convolution Neural Networks*
- *For LLMs, the `Tile-Pipeline` scheme which breaks each individual kernels is more suitable.*

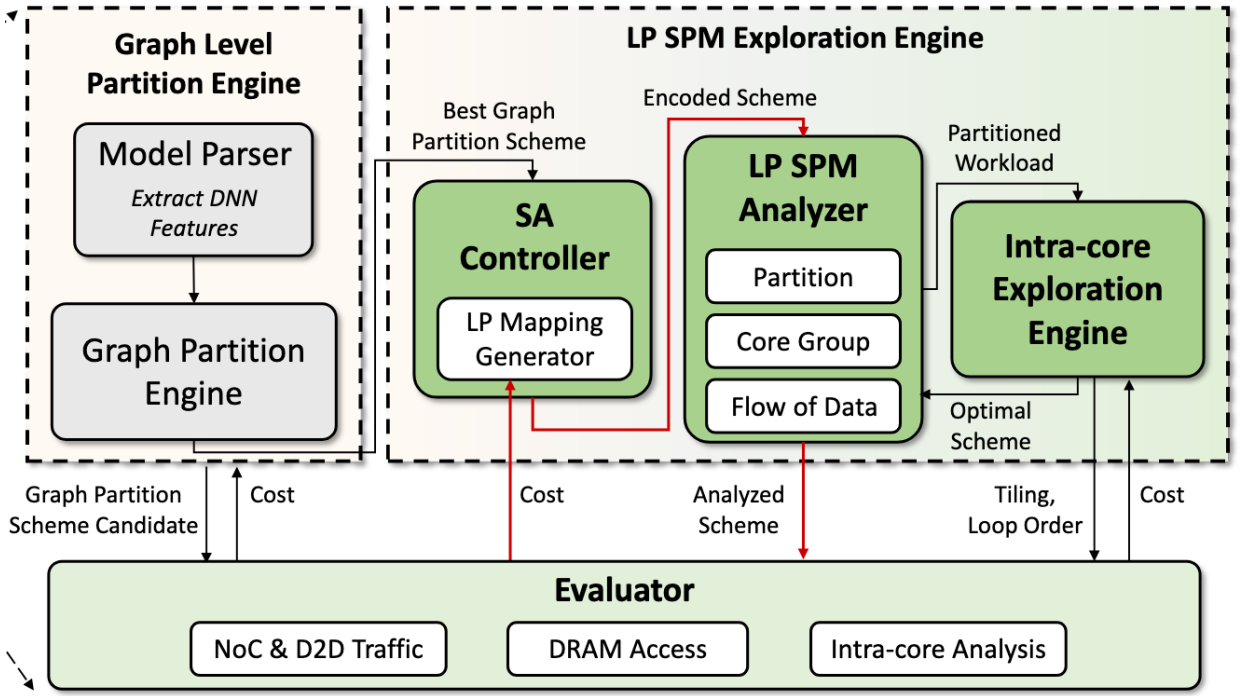
Layer-Pipeline Encoding

1. Layer Group <-> Core Group
   - Partition the graph into Layer Groups
   - Group the Compute Core Into Core Groups
2. Map one Layer Group to one Core Group
   - Part1(1,1,2,2): Initially divide the ifmaps evenly to match the number of #Cores
   - CG1(2,1,5,4): is an ordered sequence. Data partitions are mapped to each core in sequential order
   - FD(1,1,-1): Input and weight come from DRAM1, Output is stored on chip



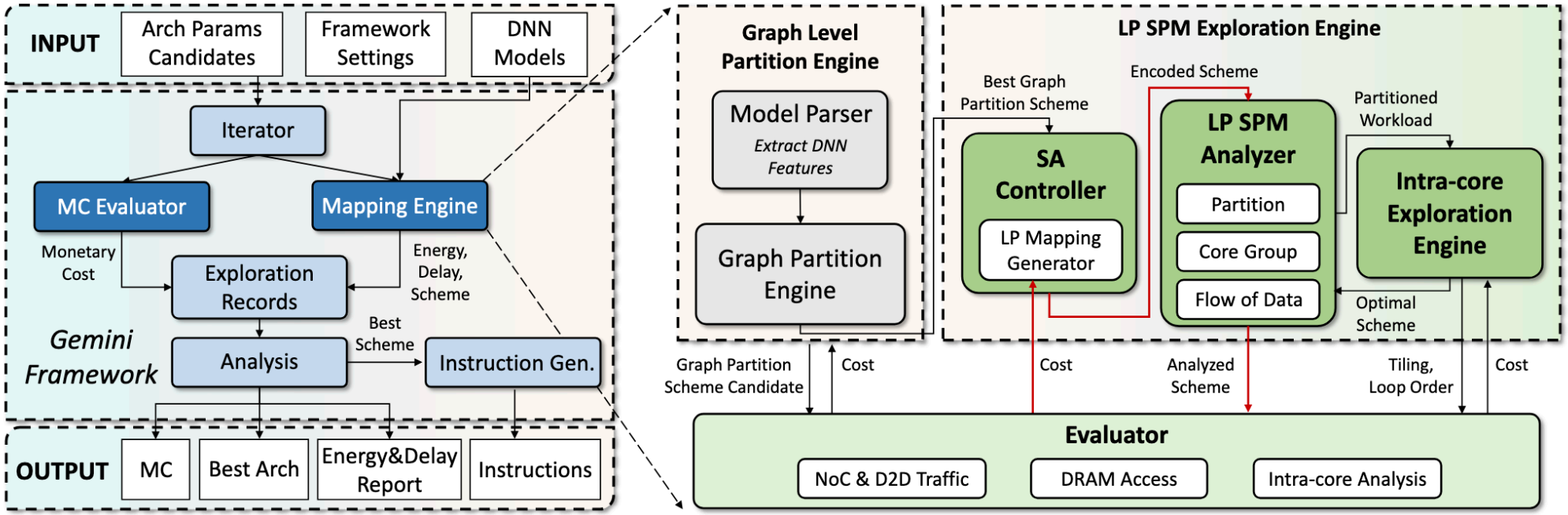
Search Framework for Mapping

- Graph Partition Algorithm is heuristic-based and is not the focus of this work, as well as intra-core exploration
- The best mapping is search by Simulated Annealing
- Performance and energy consumption is calculated by Router Traffic, DRAM Access and Intra-core Analysis



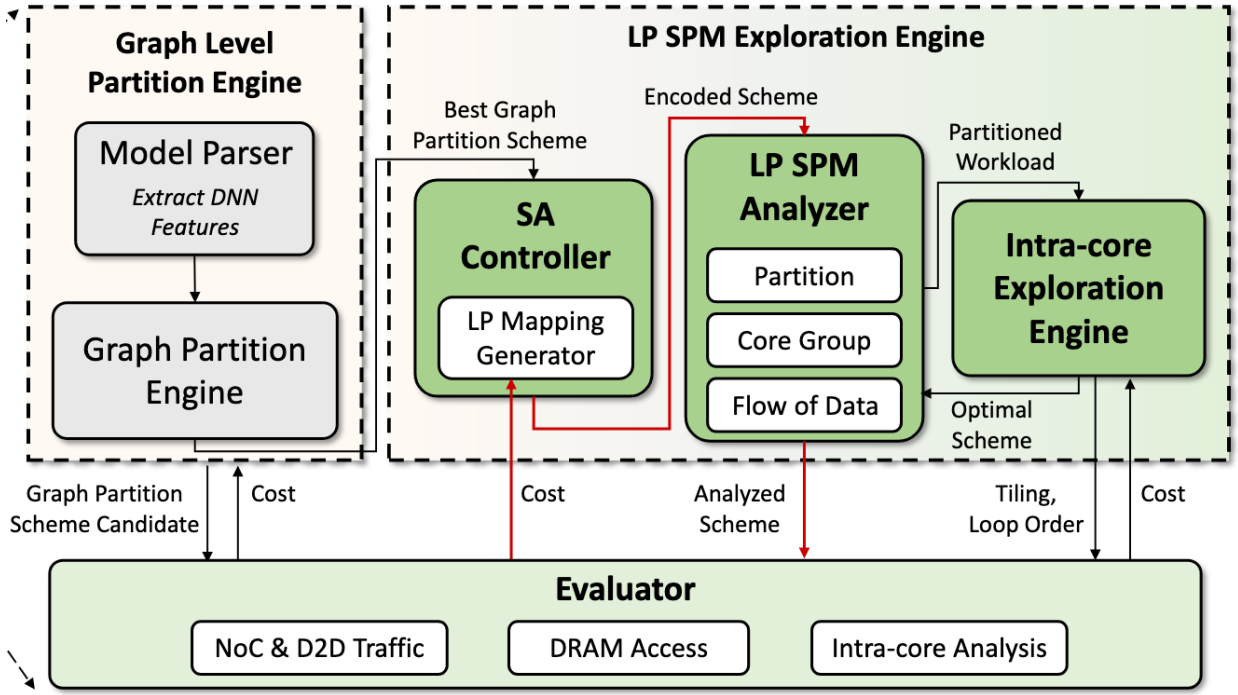
Search Framework for Mapping

- Graph Partition Algorithm is heuristic-based and is not the focus of this work, as well as intra-core exploration
- The best mapping is search by Simulated Annealing
- Performance and energy consumption is calculated by Router Traffic, DRAM Access and Intra-core Analysis



Search Framework for Mapping

- Graph Partition Algorithm is heuristic-based and is not the focus of this work, as well as intra-core exploration
- The best mapping is search by Simulated Annealing
- Performance and energy consumption is calculated by Router Traffic, DRAM Access and Intra-core Analysis



Five operators are proposed for SA

OP1: Randomly select a layer and change the values in its $Part$, while still satisfying the constraints of $Part_i$.
OP2: Randomly select a layer and randomly swap two cores within its CG, which is equivalent to exchanging the workload between these two cores for a single layer randomly.
OP3: Randomly select two layers and swap two cores within their CGs, which is equivalent to exchanging the workload between these two cores for two layers.
OP4: Randomly select two layers, remove a core randomly from the CG of one layer, and add it to the CG of the other layer. After the operation, update the $Parts$ of both layers randomly to match their new CG sizes.
OP5: Randomly select a layer, then choose a non-negative item in its FD randomly, and update its value within the range of 0 to the number of DRAMs randomly.

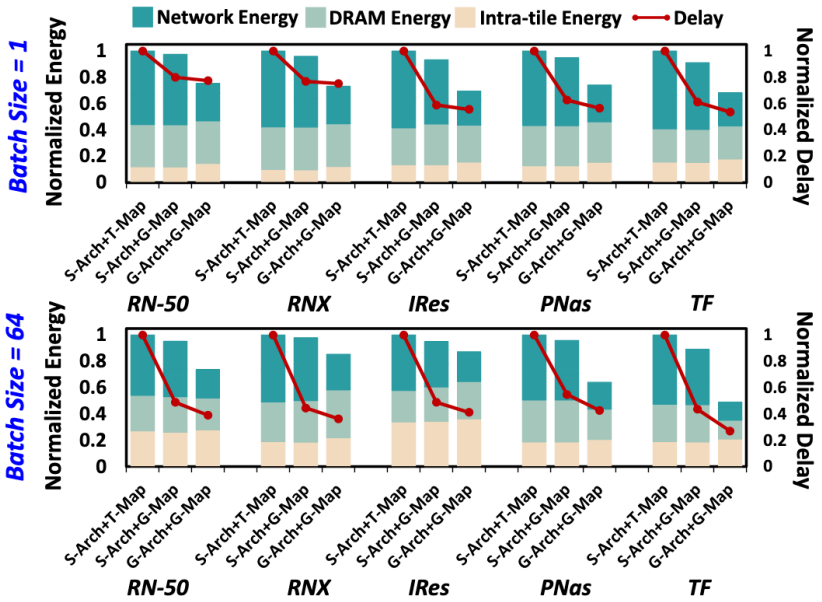
Evaluation

Architecture Search Space

- Baseline architecture: Simba with the same peak performance

Accelerator Config.	X_{Cut}	*1, 2, 3, 6* (**1, 2, 4, 8**)
	Y_{Cut}	*1, 2, 3, 6* (**1, 2, 4, 8**)
	$DRAM_{BW}$	0.5, 1, 2 GB/s per TOPs
Network Config.	NoC_{BW}	*8*, 16 ,32, 64, **128** GB/s
	$D2D_{BW}$	NoC/4, NoC/2, NoC
Core Config.	$GBUF/Core$	256, 512, 1024, 2048, 4096, **8192 KB**
	$MAC/Core$	*512*, 1024, 2048, 4096, **8192**

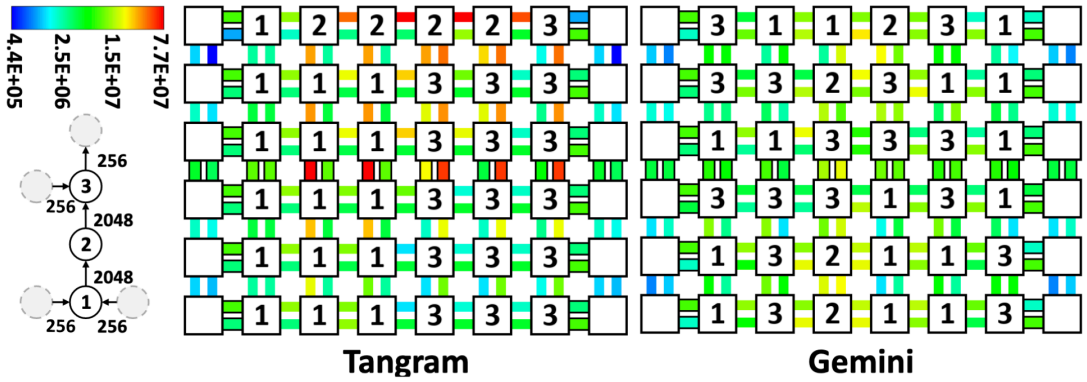
Overall Comparison



Evaluation

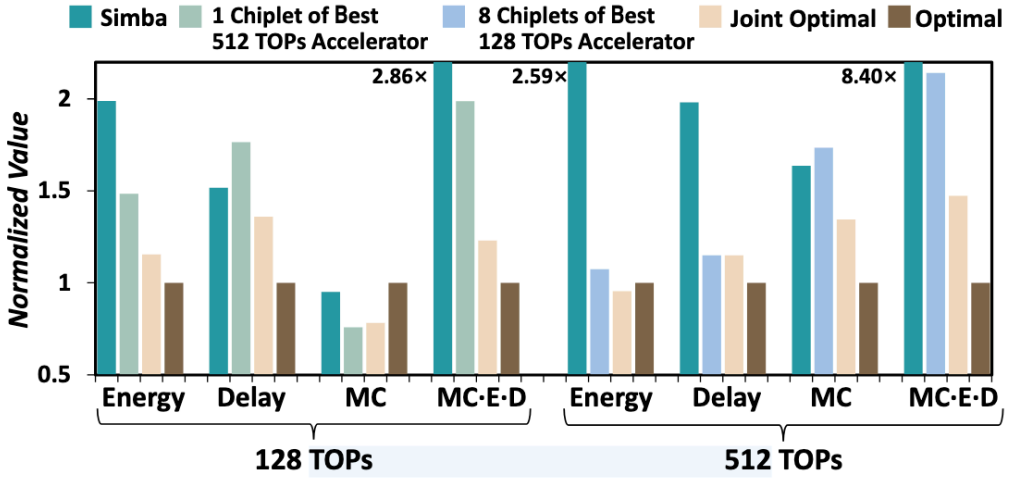
Traffic equalization

- 72 TOPs Arch searched by Gemini
- 3-Layer Transformer Workload
- Links with black borders are D2D Links
- Tangram is a heuristic-based algorithm



Compared with Simba

One-size-fit-all is impractical



Thanks!