Qubit Allocation as a combination of the Subgraph Isomorphism and Token Swapping Algorithms by Partitioning with a Failure Heuristic

Steven Yuan

Professor Zhang

16:198:516 Programming Languages and Compilers II

5/12/2021

Source Code: https://github.com/syall/CS516-Project-1

Presentation: https://github.com/syall/CS516-Project-1/blob/main/docs/PRESENTATION.mp4

Introduction

Qubit allocation for quantum circuits on hardware has been an open research problem with the onset of normalizing quantum compiler methods. The problem is modeled as the qubit mapping problem, where the input is a quantum circuit with quantum logical gates and a coupling graph defining the quantum hardware qubit relationships, and the output is an initial mapping of logical qubits to physical qubits and a modified quantum circuit. Common assumptions of the qubit mapping problem are that coupling graphs are undirected between qubits, and SWAP gates are inserted to resolve logical and physical relations.

```
Qubit Mapping Problem

Input: quantum circuit C, coupling graph G
Output: initial mapping of qubits M and modified quantum circuit C_f
```

The approach modifies existing research of framing the qubit allocation problem as a combination of subgraph isomorphism and token swapping [2]. The proposed algorithm named subgraph isomorphism partitioning with failing heuristic (SIPF) partitions maximal isomorphic sublists instead of incrementally building them, navigating the decreasing potential partitions with a failure heuristic.

Algorithms

Overview

The algorithms used in the compiler follow a pipeline between different functions, but for algorithms related to qubit allocation, the algorithms covered will be SIPF for finding compatible logical qubit mappings across the circuit and CalculateSwaps for finding swaps between physical qubits to join consecutive logical qubit mappings. Other aspects omitted in this section can be found in the Implementation section.

```
Input: quantum circuit file C_file, coupling graph file G_file
Output: modified quantum circuit C_m

// Preprocess Circuit
(LR ← live ranges, C ← quantum circuit gates) ← PreprocessCircuit()

// Build Coupling Graph
G ← BuildCouplingMap

// Produce Mappings
P ← partitions of range and mapping pairs ← SIPF(G, LR, C)

// Calculate Swaps
S ← physical qubit swaps between mappings ← CalculateSwaps(P, G)

// Compile Circuit
C_m ← modified quantum circuit C_m ← CompileCircuit(C_file, P, S, C, G)

// Output Circuit
return C_m
```

SIPF

SIPF is the framework that solves the subgraph isomorphism. Starting with left and right indices of 0 and the size of the input quantum circuit, the algorithm repeatedly partitions the circuit using the indices.

For a given range lower bound L and upper bound R, SIPF attempt to find a mapping for all of the gates in the range through the BacktrackLevel algorithm.

- If a mapping is found, then the partition is added with L, R, and the mapping into the overall list of mapping partitions.
- If a mapping is not found, the failure heuristic is used to find the latest gate of the earliest conflict qubits within the range L and R.
- If there are no conflict qubits, then the upper bound R is simply decreased by 1, excluding one gate at a time.

Logical islands logical graphs, and are the units that are mapped per logical qubit mapping. Islands are produced in case logical qubits are completely disjoint, normally only a problem in benchmark circuits where the circuits are randomly generated. The islands are backtracked in decreasing size order, as larger islands are easier to fit with more unmapped physical qubits.

Also, in evaluation, an optimal approach that does not use the failure heuristic is provided with the flag `-optimal`, decreasing the upper bound R by one gate at a time.

```
Input: coupling graph G, live ranges LR, quantum circuit gates C
Output: mapping partitions P

MAX ← maximum upper bound ← size(C)
L ← left index ← 0
R ← right index ← MAX

P ← partitions of (lower bound, upper bound, mapping) ← []
while L < R:

  // Sub Circuit
  C_sub ← subcircuit with gates L to R right exclusive

  // Logical Relationship Islands
  ISLANDS ← Build isolated logical graphs based on logical relations

  // Failure Heuristic
  F ← failure heuristic
      - Maximum failed size
      - Conflict qubits at Maximum failed size

  // Mapping
  M ← mapping[logical → physical] ← EMPTY

  // Set of mapped logical qubits
  SEEN ← set[logical] ← EMPTY

  // Set of mapped physical qubits
  MAPPED ← set[physical] ← EMPTY

  // Backtrack Level (Island)
  BacktrackLevel(ISLANDS, G, M, SEEN, MAPPED, F)
  if mapping is found:
    // Add partition
    P ← [P, (L, R, M)]
    // Update Bounds
    L = R
    R = MAX
  else if no conflict gates found in failure heuristic:
    R = R - 1
  else:
    R = Index of Latest Gate of the Earliest Conflict Qubits between L and R

return P
```

BacktrackLevel

BacktrackLevel is a modified version of the DAF algorithm [1] that finds mappings of the subcircuit logical relation graph in the coupling graph. For each logical island, BacktrackLevel is called so that all of the logical qubits in that island are mapped before the next island is mapped.

If there is only one logical qubit in an island, the mapping is handled within BacktrackLevel before another recursive call to BacktrackLevel with the rest of the logical islands.

```
BacktrackLevel
Input:
  logical graphs ISLANDS, coupling graph G, mapping M,
  mapped logical qubits SEEN, mapped physical qubits MAPPED,
  previous logical qubit PREV, failure heuristic F
Output: whether a mapping was found, and updating mapping M and failure heuristic F

// Base case: no logical graphs to map
if ISLANDS is EMPTY:
  return true

G' ← Create Data Graph using G and MAPPED

// Get head and tail of ISLANDS
ISLAND = head(ISLANDS)
NEXT_ISLANDS = tail(ISLANDS)

// If island only has one logical qubit
if |ISLAND| == 1:
  Map logical qubit Q to a physical qubit in mapping M
  - Update SEEN
  - Update MAPPED
  - Update F
  return BacktrackLevel(
    NEXT_ISLANDS, G, M,
    SEEN, MAPPED,
    Q, F)
// If island only has more than one logical qubit
else:
  (C'_DAG, CAND_SETS) ← Build a DAG from logical ISLAND
        - Build candidate sets CAND_SETS for the heuristic
        - Use heuristic |candidate set(v)| / degree(v) to choose root vertex
  (CS, CAND_EDGES) ← Build a candidate space using ISLAND, CAND_SETS, C'_DAG, and G'
        - Find physical qubit candidates for logical qubits using degrees
        - Build candidate edges CAND_EDGES while traversing the space
  P ← Calculate set of Parents for each logical qubit in C'_DAG
  FRONT ← frontier of which qubits to search ← Root Node of C'_DAG
  return BacktrackLevelHelper(
    NEXT_ISLANDS, G,
    CAND_SETS, CAND_EDGES,
    P, C'_DAG, M,
    FRONT, SEEN, MAPPED,
    PREV, F)
```

For logical islands with more than one qubit, the logical island is searched with BacktrackLevelHelper based on the DAF algorithm.

A directed acyclic graph (DAG) of the subcircuit is built using the minimum heuristic for a vertex `v |candidate set (v)| / degree(v)` to choose a root, aiming to have few candidates and a large number of edges to fail early. The DAG is used to build and search a candidate space, finding physical qubit candidates for the logical qubits in the DAG.

If the search returns a complete mapping, then the upper bound and mapping are returned. Otherwise, the failure heuristic would be already updated with conflict qubit information.

Unlike the DAF algorithm, BacktrackLevelHelper is much simpler as it omits features. The search does not use adaptive matching ordering with minimizing candidate-size and path-size heuristics to search smaller branches first. Also, pruning is not done by keeping track of fail sets to avoid redundant sibling searches for vertices that are not the source of the logical relation conflict. However, this is more of an implementation detail, as having these features incorporated would only affect the performance of the algorithm, not the functionality.

```
BacktrackLevelHelper
Input:
  logical graphs ISLANDS, coupling graph G,
  candidate sets CAND_SETS, candidate edges CAND_EDGES,
  parents P, logical DAG C_DAG, mapping M,
  frontier FRONT, mapped logical qubits SEEN, mapped physical qubits MAPPED,
  previous logical qubit PREV, failure heuristic F
Output: whether a mapping was found, and updating mapping M and failure heuristic F

// Base case:
if FRONT is EMPTY:
  // No Islands to Map
  if ISLANDS is EMPTY:
    return true
  // Map rest of the islands
  else:
    return BacktrackLevel(
      ISLANDS, G, M,
      SEEN, MAPPED,
      Q, F)

for logical qubit Q in FRONT:
  Try to map Q to any of the physical qubit candidates
    iff parents logical qubits are already mapped (extensible)
  - Update a NEW_SEEN with Q
  - Update a NEW_MAPPED with a candidate
  - Update a NEW_MAPPING with Q mapped to a candidate
  - Update a NEW_FRONT with Q's neighbors
  // Mapping Found
  if BacktrackLevelHelper with the updated state is true:
    SEEN = NEW_SEEN
    MAPPED = NEW_MAPPED
    M = NEW_MAPPING
    return true
  else:
    Update failure heuristic F

// No Mapping found
return false
```

Failure Heuristic

The failure heuristic is updated by recording the maximum size of a partial mapping and logical qubits that mapped to that size with the set of vertices that caused the logical qubit to fail, the conflict qubits.

Specifically, the failure heuristic is updated each time an attempted mapping fails. In general, there are three cases:

- The current mapping size is less than the failure heuristic maximum size, so there is no update.
- The current mapping size is equal to the failure heuristic maximum size, so the current conflicting qubits are added to the conflict qubits
- The current mapping size is greater than the failure heuristic maximum size, so the maximum size is updated, the conflict qubits are cleared, current conflicting qubits are added to the conflict qubits

When updating the bounds at the SIPF level, all conflicting qubit gates within the range of the lower bound L and the upper bound R are collected.

- If no conflicting gates exist, then decrease R by 1
- Otherwise, set R to be the gate with the latest position in the circuit

Essentially, by tracking which qubits were conflicted, the partition can be reduced by the latest gate between conflict qubits within a certain range instead of naively decreasing by one.

```
Failure Heuristic:
- Maximum Size mapped up to Conflict Qubits
- Conflict Qubits are qubits that cause the mapping to fail
```

CalculateSwaps

CalculateSwaps is an implementation of the colored token swapping problem, minimizing the swaps of colored tokens between adjacent colored vertices from an initial graph to a final graph. In the algorithm, a list of swap gates is produced between each mapping using bounded depth-first heuristic search.

First, a distance matrix between all physical qubits in the coupling graph is produced using the Floyd Warshall Algorithm for solving the All Pairs Shortest Path problem, precalculating the cost.

Then each pair of consecutive mappings is evaluated to produce the shortest distance cost. This cost is used to determine the first depth to search using the equation `DEPTH = COST / 2` as mentioned as the lower bound of the optimal number of swaps [3].

The bounded depth-first heuristic search is relatively simple, searching through swaps if:

- The depth is not less than 0
- The cost of the swap is lower or equal to the previous cost

If a swap is explored the swap is probably a happy swap [3], meaning that for both logical qubits exchanged, the swap was along a possible shortest path. More particularly, if the cost was equal, it means that the swap had no effect. However, during testing, without allowing equal-cost swaps, a reasonable swapping was not possible.

One edge case for swaps is when there is no mapping for a logical qubit in the second of the mappings, meaning the value will need to be propagated later. In this case, the cost for the unknown logical qubit is the `number of physical qubits / 2`, the average path length from any qubit to another qubit. When set to any other number, the search is skewed to go depth-first which is not desirable. Due to this edge case, the cost function is not admissible, but is still fast and guaranteed to find an optimal number of swaps given the depth restriction.

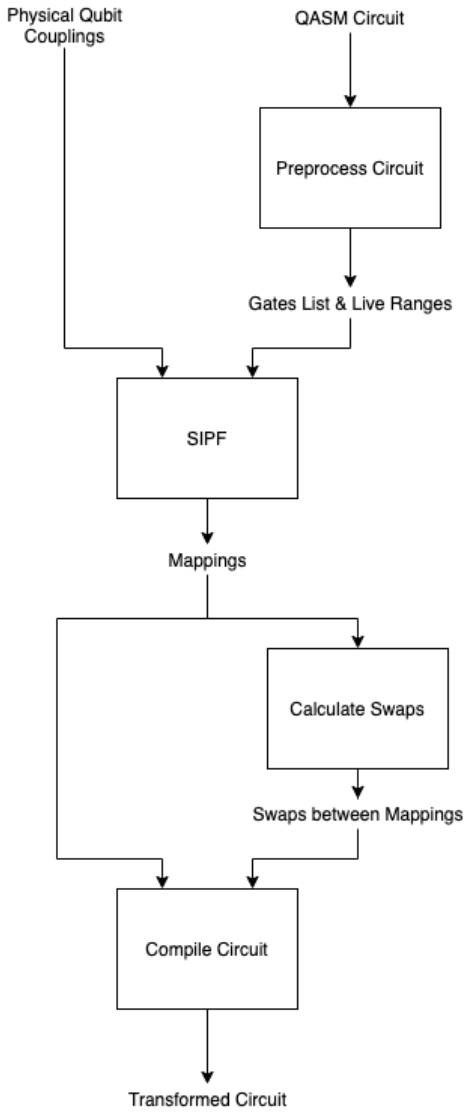
```
Input: partitions of mappings P, coupling graph G
Output: List of List of swaps S between each mapping in P

S ← List of List of swaps

G' ← Create Data Graph using G and MAPPED

DIST ← distance matrix between all physical qubits in G'

for each pair of mappings M1 and M2 of P:
  COST ← sum of all shortest distances between logical qubits positions in M1 and M2
  LOCAL ← sequence of swaps
  for DEPTH ← COST / 2, increase DEPTH by 1:
    Recursively DFS all possible swaps up to DEPTH with LOCAL
    if LOCAL is found:
      Add LOCAL to S for M1 and M2
      for every unmapped logical qubit Q in M2:
        M2[Q] = propagate values from swaps done to M1
      break
    else:
      LOCAL ← EMPTY

Return S
```

Implementation

As mentioned in the Algorithms Overview section, the compiler is implemented as a pipeline through different functions to produce a transformed circuit

- Circuit Parsing: Parsing the input circuit to produce the list of gates with live ranges
- Qubit Mapping: Mapping logical qubits to physical qubits and different points in the circuit using the SIPF algorithm
- Token Swapping: Calculating physical qubit swaps to move logical qubit positions between consecutive mappings using the CalculateSwaps algorithm
- Circuit Compiling: Applying the compatible mappings of logical qubits and inserting swap gates between those mappings, while also calculating metadata about the number of swaps, number of mappings, depth, and number of gates

Architecture



Circuit Parsing

The circuit parsing was coded to read gates line by line in order, not using the given `QASMparser`. The reason is that although the layers produced are optimal, but the gates are also unordered. The preemptive parallel expanding of the gates by the parser made it so that swap gates inserted would not produce an equivalent circuit. The order of the gates by the parser would only work if no other gates were added since parallel layers and analysis had already been produced.

Qubit Mapping

See the Algorithms SIPF section.

Token Swapping

See the Algorithms CalculateSwaps section.

Circuit Compiling

The compilation was a simple substitution once the mappings and swaps were already produced. At the top of each file includes metadata in comments:

- `//Number of Swaps: #`
- `//Number of Mappings: #`
- `//Depth: #`
- `//Number of Gates: #`

Whenever there is a new mapping, a comment for the initial mapping is produced: `//Location of qubits: q[0],q[1],...` .

Also, before each series of swap gates, a comment is added for the number of swaps between mappings: `//Insert # Swap Gates` .

The inserted swap gates are in the form `swp c, t` , manually defined in qelib1.inc as 3 CNOT gates. This is necessary when evaluating the depth and number of gates.

```
// swap
gate swp c,t
{
  // 3 CNOT gates
  cx c, t;
  cx t, c;
  cx c, t;
}
```

Although `QASMparser` was not used for parsing the circuit initially, the parser is used on the compiled circuit to calculate the metadata for depth and number of gates using a temporary file with the circuit as input.

Evaluation

All benchmarks are run on the iLab machine `cheese.cs.rutgers.edu` under the username `sy533` , and are a combination of circuits and architectures as follows:

- Circuits:
  - 3_17_13
  - ex-1_166
  - ham3_102
  - or
  - 4gt13_92
  - 4mod5-v1_22
  - alu-v0_27
  - mod5mils_65
  - qaoa5
  - 16QBT_05CYC_TFL_0 (for aspen4 only)
  - 16QBT_10CYC_TFL_3 (for aspen4 only)
- Architectures:
  - 2x3
  - aspen4
  - qx2

Besides the compiler's own algorithm, the SIPF compiler is compared to the Enfield Compiler's variations of the BMT algorithm [2]:

- SIPF Compiler:

    - Default
    - Optimal
- Enfield Compiler:
    - bmt
    - ibmt
    - opt_bmt
    - simplified_bmt
    - simplified_ibmt

The measurements can be found in the Appendix section, the depth, gates, real time (ms), and user + sys time (ms) for each compiler algorithm.

The SIPF optimal algorithm outperformed the SIPF default failure heuristic algorithm, obviously in depth and gates, but also in time. The phenomenon is probably due to the SIPF default algorithm producing more mappings, which means more chances of inserting swaps (although there are cases where neighboring mappings are equal and require no swaps).

The SIPF optimal algorithm also performs much better in time, which is surprising due to the optimal algorithm definitely performing more SIPF algorithms per compilation. The gap between the algorithms could be due to:

- SIPF being so fast that the extra iterations are negligible for the optimal algorithm
- CalculateSwaps being so slow that the extra iterations for the default algorithm are noticeable
- The failure heuristic update operations are slow for the default algorithm

The SIPF algorithms performed better in both depth and gates than the Enfield algorithms. The SIPF algorithms probably achieved this due to optimizing for larger maximal isomorphic sublists, a feature that BMT purposely limits due to the Enfield algorithms avoiding combinatorial explosion when incrementally building sublists.

For performance, the SIPF algorithms performed better than Enfield algorithms on average, but in different extreme cases, both SIPF and Enfield algorithms are better than each other.

- The Enfield algorithms' complexity and time scales with the number of logical qubits in a circuit to build out mappings, so benchmarks such as 16QBT_05CYC_TFL_0 and 16QBT_10CYC_TFL_3 take magnitudes longer than simpler circuits.
- The SIPF algorithms' complexity and time scales with the number of conflicting gates: more conflicts means more mappings and swappings. So, benchmarks such as 4gt13_92, 4mod5-v1_22, and alu-v0_27 take magnitudes longer than "smoother" circuits.

Conclusion

The SIPF algorithm is another way to frame the qubit allocation problem with subgraph isomorphism and token swapping, focusing on maximizing maximal isomorphic sublist size. Although the main innovation was to use the failure heuristic, the SIPF optimal algorithm performed better in both depth, the number of gates, and time. When compared to the Enfield algorithms, the SIPF algorithms performed better on average, but the SIPF algorithms performed much worse when circuits had many qubit conflicts.

For further work, the SIPF implementation is greedy and does not take into account different permutations of valid mappings for a range of gates. Searching through different mappings and swaps across an entire circuit could be done in the future, similar to the BMT algorithm [1]. The token swapping algorithm can be improved since the CalculateSwaps algorithm is the bottleneck of performance for the compiler by doing a bounded depth-first heuristic search (there are no clear reference implementations for the token swapping algorithms available).

References

1. Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19). Association for Computing Machinery, New York, NY, USA, 1429–1446. DOI:https://doi.org/10.1145/3299869.3319880

2. Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintão Pereira. 2019. Qubit allocation as a combination of subgraph isomorphism and token swapping. Proc. ACM Program. Lang. 3, OOPSLA, Article 120 (October 2019), 29 pages. DOI:https://doi.org/10.1145/3360546
3. Tillmann Miltzow, Lothar Narins, Yoshio Okamoto, Günter Rote, Antonis Thomas, and Takeaki Uno. 2016. Approximation and hardness for token swapping. arXiv:1602.05150.

Appendix

SIPF Default Benchmarks

Circuit--Architecture	Depth	Gates	Real Time (ms)	User + Sys Time (ms)
3_17_13--2x3	49	63	17	12
3_17_13--aspen4	60	81	24	20
3_17_13--qx2	22	36	12	8
ex-1_166--2x3	28	34	13	9
ex-1_166--aspen4	33	43	16	12
ex-1_166--qx2	12	19	12	8
ham3_102--2x3	32	38	14	10
ham3_102--aspen4	37	47	18	14
ham3_102--qx2	13	20	12	8
or--2x3	27	35	12	8
or--aspen4	39	53	15	10
or--qx2	8	17	12	7
4gt13_92--2x3	129	180	66	62
4gt13_92--aspen4	141	228	2283	2279
4gt13_92--qx2	38	66	6	2
4mod5-v1_22--2x3	27	42	7	3
4mod5-v1_22--aspen4	36	51	82	78
4mod5-v1_22--qx2	24	36	7	4
alu-v0_27--2x3	55	87	11	7
alu-v0_27--aspen4	79	108	3390	3386
alu-v0_27--qx2	28	45	6	2
mod5mils_65--2x3	45	62	8	4
mod5mils_65--aspen4	42	62	10	7
mod5mils_65--qx2	36	50	9	5
qaoa5--2x3	14	22	6	2
qaoa5--aspen4	14	22	8	4

Circuit--Architecture	Depth	Gates	Real Time (ms)	User + Sys Time (ms)
qaoa5--qx2	14	22	10	5
16QBT_05CYC_TFL_0--aspen4	5	37	11	7
16QBT_10CYC_TFL_3--aspen4	10	73	17	12

SIPF Optimal Benchmarks

Circuit--Architecture	Depth	Gates	Real Time (ms)	User + Sys Time (ms)
3_17_13--2x3	43	57	10	6
3_17_13--aspen4	43	57	12	9
3_17_13--qx2	22	36	5	2
ex-1_166--2x3	24	31	5	3
ex-1_166--aspen4	24	31	7	3
ex-1_166--qx2	12	19	5	2
ham3_102--2x3	25	32	5	3
ham3_102--aspen4	25	32	6	4
ham3_102--qx2	13	20	5	2
or--2x3	17	26	5	2
or--aspen4	23	35	5	3
or--qx2	8	17	5	2
4gt13_92--2x3	103	159	13	10
4gt13_92--aspen4	122	192	126	122
4gt13_92--qx2	38	66	6	2
4mod5-v1_22--2x3	27	42	14	11
4mod5-v1_22--aspen4	36	51	113	108
4mod5-v1_22--qx2	24	36	7	4
alu-v0_27--2x3	55	87	11	7
alu-v0_27--aspen4	79	108	3358	3353
alu-v0_27--qx2	28	45	6	3
mod5mils_65--2x3	45	62	8	4
mod5mils_65--aspen4	42	62	11	7
mod5mils_65--qx2	36	50	10	6
qaoa5--2x3	14	22	7	3
qaoa5--aspen4	14	22	7	4
qaoa5--qx2	14	22	6	2

Circuit--Architecture	Depth	Gates	Real Time (ms)	User + Sys Time (ms)
16QBT_05CYC_TFL_0--aspen4	5	37	8	4
16QBT_10CYC_TFL_3--aspen4	10	73	17	13

Enfield bmt Benchmarks

Circuit--Architecture	Depth	Gates	Real Time (ms)	User + Sys Time (ms)
3_17_13--2x3	67	110	25	24
3_17_13--aspen4	61	98	61	59
3_17_13--qx2	34	60	16	15
ex-1_166--2x3	35	56	15	14
ex-1_166--aspen4	33	52	66	64
ex-1_166--qx2	18	31	12	11
ham3_102--2x3	36	57	16	15
ham3_102--aspen4	32	49	40	39
ham3_102--qx2	21	36	13	12
or--2x3	20	35	17	16
or--aspen4	18	31	40	39
or--qx2	8	17	13	12
4gt13_92--2x3	121	205	35	34
4gt13_92--aspen4	126	234	309	307
4gt13_92--qx2	60	110	20	19
4mod5-v1_22--2x3	35	58	16	15
4mod5-v1_22--aspen4	35	58	35	34
4mod5-v1_22--qx2	31	50	14	13
alu-v0_27--2x3	63	106	25	24
alu-v0_27--aspen4	64	119	204	202
alu-v0_27--qx2	38	67	16	15
mod5mils_65--2x3	62	108	22	21
mod5mils_65--aspen4	62	108	55	54
mod5mils_65--qx2	41	68	16	15
qaoa5--2x3	18	30	14	13
qaoa5--aspen4	14	22	28	26
qaoa5--qx2	14	22	24	22
16QBT_05CYC_TFL_0--aspen4	5	37	21356	21313

Circuit--Architecture	Depth	Gates	Real Time (ms)	User + Sys Time (ms)
16QBT_10CYC_TFL_3--aspen4	16	97	9492	9490

Enfield ibmt Benchmarks

Circuit--Architecture	Depth	Gates	Real Time (ms)	User + Sys Time (ms)
3_17_13--2x3	67	110	30	27
3_17_13--aspen4	61	98	53	52
3_17_13--qx2	34	60	16	15
ex-1_166--2x3	35	56	15	14
ex-1_166--aspen4	33	52	32	31
ex-1_166--qx2	18	31	12	11
ham3_102--2x3	36	57	15	15
ham3_102--aspen4	32	49	32	31
ham3_102--qx2	21	36	13	12
or--2x3	20	35	15	14
or--aspen4	18	31	25	24
or--qx2	8	17	12	11
4gt13_92--2x3	111	208	31	30
4gt13_92--aspen4	134	231	149	147
4gt13_92--qx2	60	110	21	20
4mod5-v1_22--2x3	35	58	16	15
4mod5-v1_22--aspen4	35	58	34	33
4mod5-v1_22--qx2	32	50	14	13
alu-v0_27--2x3	61	110	25	23
alu-v0_27--aspen4	65	109	187	185
alu-v0_27--qx2	38	67	17	16
mod5mils_65--2x3	62	108	25	23
mod5mils_65--aspen4	62	108	59	58
mod5mils_65--qx2	41	68	17	15
qaoa5--2x3	18	30	21	19
qaoa5--aspen4	14	22	16	14
qaoa5--qx2	14	22	13	12
16QBT_05CYC_TFL_0--aspen4	5	37	22971	22969
16QBT_10CYC_TFL_3--aspen4	16	97	5220	5219

Enfield opt_bmt Benchmarks

Circuit--Architecture	Depth	Gates	Real Time (ms)	User + Sys Time (ms)
3_17_13--2x3	67	110	61	49
3_17_13--aspen4	61	98	41	31
3_17_13--qx2	34	60	22	15
ex-1_166--2x3	35	56	29	20
ex-1_166--aspen4	32	48	35	25
ex-1_166--qx2	18	31	20	13
ham3_102--2x3	36	57	36	27
ham3_102--aspen4	32	49	59	49
ham3_102--qx2	21	36	20	14
or--2x3	20	35	28	19
or--aspen4	18	31	43	34
or--qx2	8	17	20	14
4gt13_92--2x3	111	208	41	28
4gt13_92--aspen4	124	231	127	112
4gt13_92--qx2	60	110	28	19
4mod5-v1_22--2x3	33	50	28	18
4mod5-v1_22--aspen4	35	58	38	29
4mod5-v1_22--qx2	33	55	23	14
alu-v0_27--2x3	59	106	58	46
alu-v0_27--aspen4	65	106	131	121
alu-v0_27--qx2	38	67	23	15
mod5mils_65--2x3	61	97	32	19
mod5mils_65--aspen4	64	101	48	37
mod5mils_65--qx2	41	69	24	15
qaoa5--2x3	18	30	21	13
qaoa5--aspen4	14	22	45	36
qaoa5--qx2	14	22	50	40
16QBT_05CYC_TFL_0--aspen4	5	37	5447	5439
16QBT_10CYC_TFL_3--aspen4	16	97	3469	3461

Enfield simplified_bmt Benchmarks

Circuit--Architecture	Depth	Gates	Real Time (ms)	User + Sys Time (ms)

Circuit--Architecture	Depth	Gates	Real Time (ms)	User + Sys Time (ms)
3_17_13--2x3	67	110	65	53
3_17_13--aspen4	61	98	58	47
3_17_13--qx2	34	60	21	14
ex-1_166--2x3	35	56	27	18
ex-1_166--aspen4	33	52	46	37
ex-1_166--qx2	18	31	22	15
ham3_102--2x3	36	57	41	29
ham3_102--aspen4	32	49	64	53
ham3_102--qx2	21	36	19	12
or--2x3	20	35	20	12
or--aspen4	18	31	26	19
or--qx2	8	17	17	11
4gt13_92--2x3	121	205	48	34
4gt13_92--aspen4	134	234	302	287
4gt13_92--qx2	60	110	28	19
4mod5-v1_22--2x3	35	58	27	18
4mod5-v1_22--aspen4	35	58	41	31
4mod5-v1_22--qx2	31	50	23	14
alu-v0_27--2x3	63	106	65	51
alu-v0_27--aspen4	64	119	202	192
alu-v0_27--qx2	38	67	24	15
mod5mils_65--2x3	62	108	32	21
mod5mils_65--aspen4	62	108	61	50
mod5mils_65--qx2	41	68	23	15
qaoa5--2x3	18	30	22	15
qaoa5--aspen4	14	22	36	26
qaoa5--qx2	14	22	32	24
16QBT_05CYC_TFL_0--aspen4	5	37	21235	21227
16QBT_10CYC_TFL_3--aspen4	16	97	9535	9526

Enfield simplified_ibmt Benchmarks

Circuit--Architecture	Depth	Gates	Real Time (ms)	User + Sys Time (ms)
3_17_13--2x3	67	110	29	27

Circuit--Architecture	Depth	Gates	Real Time (ms)	User + Sys Time (ms)
3_17_13--aspen4	61	98	37	36
3_17_13--qx2	34	60	16	15
ex-1_166--2x3	35	56	15	13
ex-1_166--aspen4	33	52	24	23
ex-1_166--qx2	18	31	13	12
ham3_102--2x3	36	57	19	17
ham3_102--aspen4	32	49	35	34
ham3_102--qx2	21	36	15	13
or--2x3	20	35	15	13
or--aspen4	18	31	22	20
or--qx2	8	17	13	12
4gt13_92--2x3	111	208	34	33
4gt13_92--aspen4	124	231	130	129
4gt13_92--qx2	60	110	21	20
4mod5-v1_22--2x3	35	58	16	15
4mod5-v1_22--aspen4	35	58	28	27
4mod5-v1_22--qx2	32	50	14	13
alu-v0_27--2x3	61	110	22	22
alu-v0_27--aspen4	65	109	171	170
alu-v0_27--qx2	38	67	17	15
mod5mils_65--2x3	62	108	21	20
mod5mils_65--aspen4	62	108	42	41
mod5mils_65--qx2	41	68	16	15
qaoa5--2x3	18	30	14	12
qaoa5--aspen4	14	22	18	17
qaoa5--qx2	14	22	15	14
16QBT_05CYC_TFL_0--aspen4	5	37	23078	23042
16QBT_10CYC_TFL_3--aspen4	16	97	5236	5234