

High-throughput interleaved QRD for FPGAs

Turning pipeline bubbles into throughput

Bogdan Pasca Yohann Uguen Martin Langhammer



ASAP 2026

QR decomposition on FPGAs

Factor a square matrix A into

$$A = QR$$

with Q orthogonal and R upper-triangular.

Alongside Cholesky and LU, a fundamental tool for solving linear systems.

FPGA application domains:

- Wireless communications
- Radar beamforming
- Medical imaging
- Statistical analysis (PCA)

QR decomposition on FPGAs

Factor a square matrix A into

$$A = QR$$

with Q orthogonal and R upper-triangular.

Alongside Cholesky and LU, a fundamental tool for solving linear systems.

FPGA application domains:

- Wireless communications
- Radar beamforming
- Medical imaging
- Statistical analysis (PCA)

Why now

Hardened floating-point in modern FPGA DSP blocks makes *floating-point* QRD attractive.

Best fit: modified Gram-Schmidt (MGS) organized around efficient **dot-product primitives** mapped to DSP blocks.

QR decomposition on FPGAs

Factor a square matrix A into

$$A = QR$$

with Q orthogonal and R upper-triangular.

Alongside Cholesky and LU, a fundamental tool for solving linear systems.

FPGA application domains:

- Wireless communications
- Radar beamforming
- Medical imaging
- Statistical analysis (PCA)

Why now

Hardened floating-point in modern FPGA DSP blocks makes *floating-point* QRD attractive.

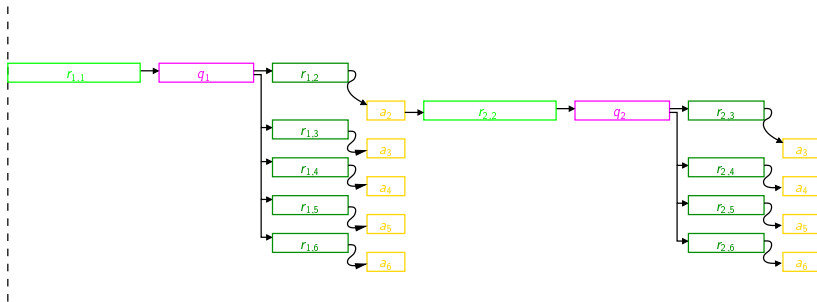
Best fit: modified Gram-Schmidt (MGS) organized around efficient **dot-product primitives** mapped to DSP blocks.

But...

This works well only when datapath latency is *short* relative to the matrix dimension n .

Baseline: modified Gram-Schmidt

```
for  $i = 1 : n$   
  for  $j = i : n$   
    if  $j == i$   
       $r_{i,i} = \|a_i\|$   
       $q_i = a_i / r_{i,i}$   
    else  
       $r_{i,j} = \langle q_i, a_j \rangle$   
       $a_j = a_j - r_{i,j} q_i$ 
```



Execution timeline, $n = 6$: step i must finish before $r_{i+1,i+1}$ starts.

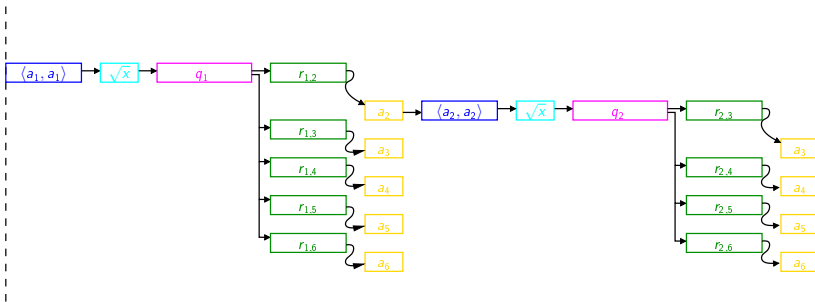
Baseline: modified Gram-Schmidt

```
for  $i = 1 : n$   
  for  $j = i : n$   
    if  $j == i$   
       $r_{i,i} = \|a_i\|$   
       $q_i = a_i / r_{i,i}$   
    else  
       $r_{i,j} = \langle q_i, a_j \rangle$   
       $a_j = a_j - r_{i,j} q_i$ 
```

Agilex 7 latencies

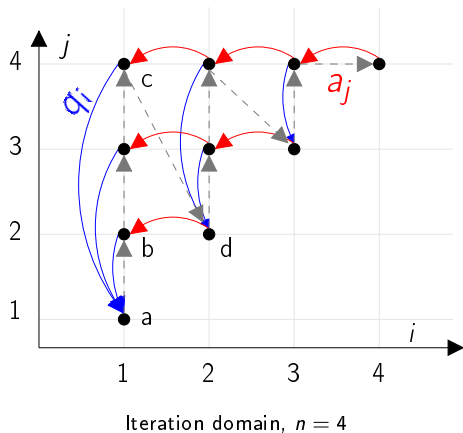
32-element dot product (SP)	≈ 20 cycles
Square root (SP, corr. rounded)	≈ 24 cycles
32-element dot product (DP)	≈ 60 cycles

Large relative to $n \Rightarrow$ **the pipeline sits idle.**



Opened up: $r_{i,i}$ is a dot product then a square root, q_i a division.

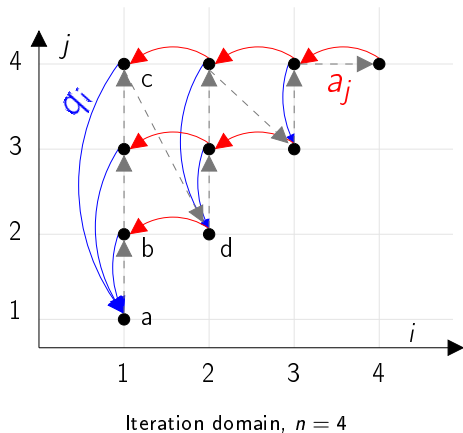
Where the stalls come from: the iteration space



Each point is one loop iteration.

- Dashed gray: sequential execution order
- Blue: carried across j — computing q_i (dot product $\rightarrow$ square root $\rightarrow$ division)
- Red: carried across i — updating a_j (dot product $\rightarrow$ multiply-add)

Where the stalls come from: the iteration space



Each point is one loop iteration.

- Dashed gray: sequential execution order
- Blue: carried across j — computing q_i (dot product $\rightarrow$ square root $\rightarrow$ division)
- Red: carried across i — updating a_j (dot product $\rightarrow$ multiply-add)

Time to issue all j iterations for a given i , and the extra delay entering the next i :

$$T_i = L_{blue} + n - i$$

$$D_i = \max(0, L_{red} - n + i - 1)$$

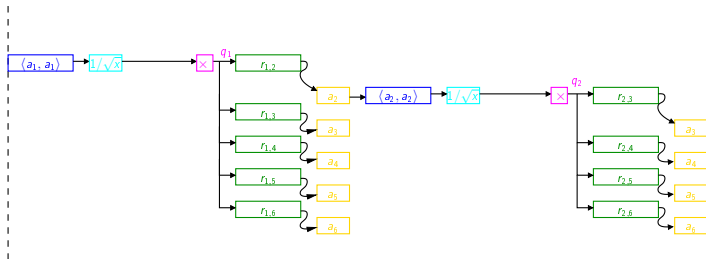
Throughput is governed by these two latencies.

Prior art: refactoring MGS to shorten the critical path

Express q_i via the **reciprocal square root** of $\langle a_i, a_i \rangle$, then substitute:

$$q_i = \frac{a_i}{r_{i,i}} = \frac{a_i}{\|a_i\|} = a_i \frac{1}{\sqrt{\langle a_i, a_i \rangle}} = a_i \times ir$$

$$r_{i,j} = \langle q_i, a_j \rangle = \frac{\langle a_i, a_j \rangle}{\sqrt{\langle a_i, a_i \rangle}} = dp \times ir \quad a_j = a_j - \frac{\langle a_i, a_j \rangle}{\langle a_i, a_i \rangle} \times a_i = a_j - s_j \times a_i$$



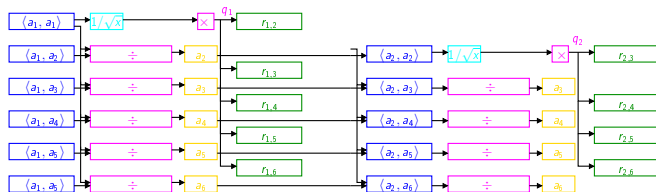
$\sqrt{}$ and $\div$ on the carried path become $1/\sqrt{}$ and $\times$.

Prior art: refactoring MGS to shorten the critical path

Express q_i via the **reciprocal square root** of $\langle a_i, a_i \rangle$, then substitute:

$$q_i = \frac{a_i}{r_{i,i}} = \frac{a_i}{\|a_i\|} = a_i \frac{1}{\sqrt{\langle a_i, a_i \rangle}} = a_i \times ir$$

$$r_{i,j} = \langle q_i, a_j \rangle = \frac{\langle a_i, a_j \rangle}{\sqrt{\langle a_i, a_i \rangle}} = dp \times ir \quad a_j = a_j - \frac{\langle a_i, a_j \rangle}{\langle a_i, a_i \rangle} \times a_i = a_j - s_j \times a_i$$

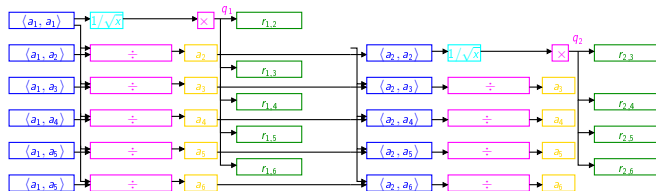


The a_j updates leave the q_i path — they need only a **quotient of dot products**.

Prior art: refactoring MGS to shorten the critical path

Express q_i via the **reciprocal square root** of $\langle a_i, a_i \rangle$, then substitute:

$$q_i = \frac{a_i}{r_{i,i}} = \frac{a_i}{\|a_i\|} = a_i \frac{1}{\sqrt{\langle a_i, a_i \rangle}} = a_i \times ir$$
$$r_{i,j} = \langle q_i, a_j \rangle = \frac{\langle a_i, a_j \rangle}{\sqrt{\langle a_i, a_i \rangle}} = dp \times ir \quad a_j = a_j - \frac{\langle a_i, a_j \rangle}{\langle a_i, a_i \rangle} \times a_i = a_j - s_j \times a_i$$



The a_j updates leave the q_i path — they need only a **quotient of dot products**.

This is the **starting point** of our work (Langhammer & Pasca, FPGA'18): hardware efficiency >90% for large matrices such as 256×256 , **but it drops for smaller sizes**.

Baseline architecture: one dot-product, one mult-add

MGS with explicit hardware reuse

```
for i = 1 : n + 1
  for j = max(1, i - 1) : n
    // input multiplexing
    if j == i - 1: rhs = ir; add = 0
    else if i > 1: rhs = -s_j; add = a_j
     $madd = a_{i-1} \times rhs + add$ 
    if j == i - 1: q_j = madd
    else if i > 1: a_j = madd
     $dp = \langle a_i, a_j \rangle$ 
    if j == i:
       $p = dp; ir = 1/\sqrt{p}; r_{i,i} = \sqrt{p}$ 
    else if j ≠ i - 1:
       $s_j = dp/p; r_{i,j} = dp \times ir$ 
```

All operations moved into the innermost loop, so that:

- the two dot-product evaluations occur in *different* iterations $\Rightarrow$ they share a **single dot-product unit**;
- the two mult-add uses are serialized by inserting extra iterations $\Rightarrow$ they share a **single mult-add unit**.

Baseline architecture: one dot-product, one mult-add

MGS with explicit hardware reuse

```
for i = 1 : n + 1
  for j = max(1, i - 1) : n
    // input multiplexing
    if j == i - 1: rhs = ir; add = 0
    else if i > 1: rhs = -s_j; add = a_j
     $madd = a_{i-1} \times rhs + add$ 
    if j == i - 1: q_j = madd
    else if i > 1: a_j = madd
     $dp = \langle a_i, a_j \rangle$ 
    if j == i:
       $p = dp$ ;  $ir = 1/\sqrt{p}$ ;  $r_{i,i} = \sqrt{p}$ 
    else if j  $\neq$  i - 1:
       $s_j = dp/p$ ;  $r_{i,j} = dp \times ir$ 
```

All operations moved into the innermost loop, so that:

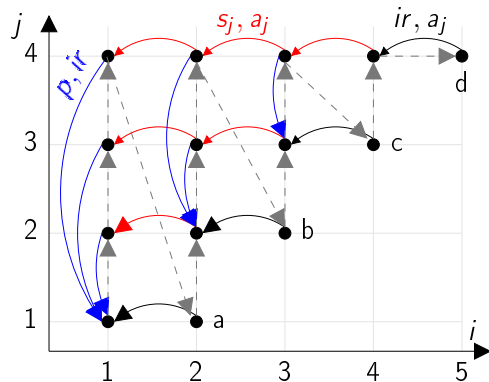
- the two dot-product evaluations occur in *different* iterations $\Rightarrow$ they share a **single dot-product unit**;
- the two mult-add uses are serialized by inserting extra iterations $\Rightarrow$ they share a **single mult-add unit**.

Important

This datapath is *already* aggressively optimized and reused.

Any remaining inefficiency is therefore **not** a shortage of parallelism — it is the dependency structure.

Which dependencies actually constrain the schedule?



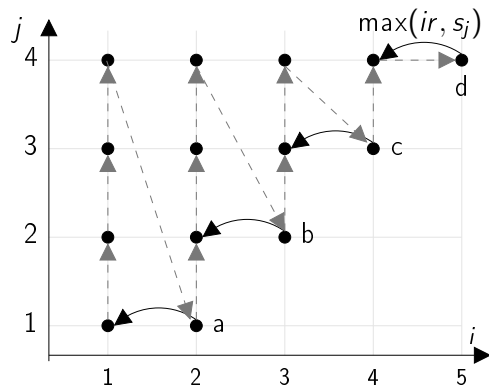
full dependencies

Blue — only p is needed to start computing $s_j = dp/p$, and the pipeline lets the dot-product stage of s_j start one cycle later. The dependence on ir feeds only $r_{i,j}$, which is never reused, so it leaves the loop body.

$\Rightarrow$ effective blue distance = 1 cycle

Red and **black** — both already include a_j , so they reduce to s_j and ir . Assuming conservatively $L_{black} = L_{red}$, resolving the black dependence also resolves the red ones.

Which dependencies actually constrain the schedule?



simplified view — one black dependence remains

Blue — only p is needed to start computing $s_j = dp/p$, and the pipeline lets the dot-product stage of s_j start one cycle later. The dependence on ir feeds only $r_{i,j}$, which is never reused, so it leaves the loop body.

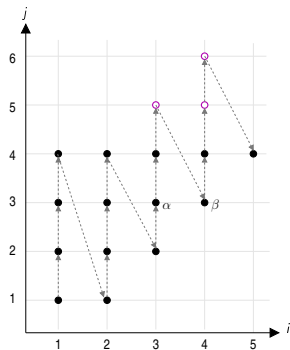
$\Rightarrow$ effective blue distance = 1 cycle

Red and **black** — both already include a_j , so they reduce to s_j and ir . Assuming conservatively $L_{black} = L_{red}$, resolving the black dependence also resolves the red ones.

One number that matters

L = the **dependency distance**: max datapath latency between computing ir and s_j .

Making the bubbles explicit

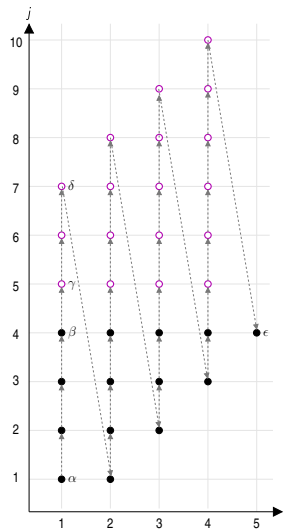


$$L = 3, n = 4$$

$$L < n$$

Bubbles

$$\frac{(L-1)L}{2}$$



$$L = 7, n = 4$$

$$L \geq n$$

Bubbles

$$nL - \frac{(n+1)n}{2}$$

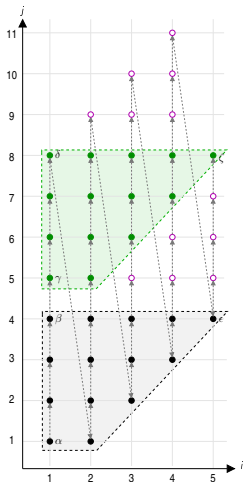
How bad is it? Measured bubble fractions

Configuration	FP format	L	Bubbles / total cycles
16×16	Real (8,23)	46	600 / 752 (79%)
16×16	Complex (8,23)	62	856 / 1008 (84%)
32×32	Complex (8,23)	72	1776 / 2336 (76%)
64×64	Complex (8,23)	80	3040 / 5188 (58%)
64×64	Real (11,52)	210	11360 / 13504 (84%)

In latency-dominated configurations,
up to 84% of issued iterations inject no new work.

The datapath is fully built and fully placed — and mostly idle.

Key idea: interleave independent decompositions



Two interleaved matrices, $n = 4$, $L = 8$.
First matrix ●, second ●.

If the iterations $\alpha \dots \beta$ of one QRD fit inside the bubble region $\gamma \dots \delta$ of another, the two decompositions can share the datapath.

Interleaving condition

$$L \geq k \times n$$

where k is the number of interleaved QRDs

- Same per-matrix arithmetic — work is only *reordered* across independent matrices.
- Extra cost: duplicate per-matrix state (s arrays, ir), plus multiplexing and address-selection logic.
- Latency overhead: the n extra iterations needed to finish the last QRD.

The interleaved schedule

Interleaved MGS QRD ($L \geq k \times n$)

```
for  $i = 1 : n + 1$ 
  for  $j = \max(1, i - 1) : k \cdot n + i - 1$ 
    if  $j \leq n$ :                                     // first matrix
       $M = A$ ;  $j' = j$ 
    else if  $n + \max(1, i - 1) \leq j \leq 2n$ :
       $M = B$ ;  $j' = j - n$                            // second matrix
    else
      continue                                       // bubble iteration
  // same datapath, per-matrix state
   $madd = lhs \times m_{i-1} + add$ 
   $dp = \langle m_i, m_{j'} \rangle$ 
  ...  $ir_m, sm_{j'}, rm_{i,j'} \dots$ 
```

- The active matrix is selected purely by the **interval of j** — a static, compile-time schedule.
- Iterations belonging to no matrix are bubbles.
- Generalizes directly to k matrices, for the largest k satisfying $L \geq kn$.

Correctness

Algorithm 3 only reorders work across *independent* matrices. The per-matrix arithmetic of the baseline is unchanged, so numerical behaviour is **identical** to the baseline.

Why this is not datapath replication

Datapath replication

Improves throughput, but duplicates the dot-product and mult-add resources.

Much higher area cost.

Interleaving (ours)

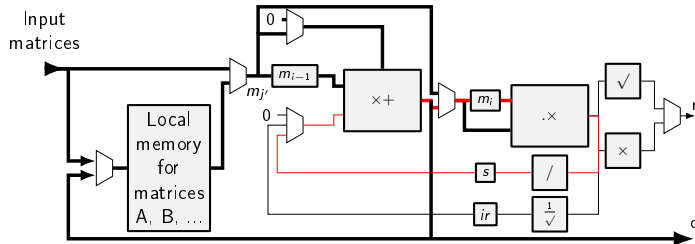
Fills *existing* idle issue slots of the *same* datapath.

Extra cost limited to **state, storage and multiplexing**.

Relation to batched QRD on GPUs

Similar at the workload level — multiple independent matrices raise aggregate throughput. But GPU batching maps matrices onto *replicated* SIMT resources, whereas we statically pack multiple QRDs onto a **single reused FPGA datapath** by filling dependency-induced bubble cycles.

Implementation



Feedback loop enforcing the pipeline bubbles highlighted in red.

- RTL generated from **oneAPI SYCL** (oneAPI FPGA 2024.2).
- Three streaming interfaces: rows of A in, rows of Q out, elements of R out.
- Mult-add and dot-product units **fully unrolled**.
- Local memory holds the input matrices, progressively overwritten with Q . No j iteration at $i = 1$ writes it, so **no double buffering** between consecutive QRDs.
- Full system adds three DMA kernels (load A , store Q , reassemble R).

Results: throughput and area

Config	k	L	Throughput (mat./s)	ALMs	M20Ks	DSPs	Bubbles / total	Thr. per norm. area
16×16 Complex SP	—	62	595k	11.1k	74	139	856 / 1008 (84%)	26.6
	2	62	1,171k (1.96×)	10.8k (−3%)	104 (+40%)	—	704 / 1024 (68%)	48.3
	3	62	1,730k (2.9×)	11.0k (−1%)	104 (+40%)	—	552 / 1040 (53%)	71.3
	4	64	2,205k (3.7×)	11.1k (0%)	104 (+40%)	—	432 / 1088 (39%)	90.8
64×64 Real DP	—	210	44k	212.8k	397	550	11360 / 13504 (84%)	0.39
	2	210	88k (2×)	214.1k (0%)	296 (−26%)	—	9280 / 13568 (68%)	0.83
	3	210	132k (3×)	214.5k (0%)	501 (+26%)	—	7200 / 13632 (52%)	1.11
	4	256	144k (3.27×)	214.1k (0%)	501 (+26%)	—	7872 / 16640 (47%)	1.21

- **DSP usage never increases** — the compute datapath is untouched.
- ALM growth is negligible (under 5%, sometimes negative).
- FPGA memory blocks are **discrete**: a matrix rarely fills a 512-element M20K, so extra matrices often land in already-allocated blocks **for free**.
- L satisfies $L \geq kn$ for free up to $k = 3$; at $k = 4$ it had to be **raised deliberately** (62→64, 210→256) — next slide.
- Last column: throughput per **normalized area** (M20Ks and DSPs converted to ALM-equivalents). It rises **3.4×** with k , while prior work on Arria 10 scores 26.7 — about our *baseline*.

L as a design parameter

Since L is the *enforced* dependency distance, it can be increased **deliberately**: more bubbles and higher latency, but a larger feasible k .

Configuration	L	Effect	Outcome
64×64 Real DP	210 → 256	enables $k = 4$	+22k mat./s, no area penalty
16×16 Complex SP	62 → 64	enables $k = 4$	2.9× → 3.7× baseline
64×64 Complex SP	80 → 128	enables $k = 2$	only 1.26× — bubbles dominate

Extrapolation

A 128-element DP dot product adds one reduction-tree level ($L \approx 230$). Since $230 < 2n = 256$, forcing $L = 256$ would enable $k = 2$ at roughly 10% single-core loss, for about $0.9 \times 2 \approx 1.8$ aggregate gain.

By contrast a 128×128 complex SP design stays outside this regime ($L \approx 84 \ll n$): its occupancy is already high.

Conclusion

- High-throughput FPGA QRD stays hard when critical-path latency is large relative to n — even aggressively refactored MGS implementations leave **most issue slots idle**.
- We derive, from the MGS dependence graph, a **static schedule** that packs independent QR decompositions onto the same reused datapath, filling those bubble cycles.
- The schedule **preserves the baseline computation**: it only reorders work across independent matrices.
- On Agilex 7: throughput up to **$3.7\times$** , with little additional logic and **no increase in DSP usage**.
- Improves throughput *density*, not just raw throughput — the interleaved designs reach higher throughput per **normalized area** than both the baseline and prior work.

In latency-dominated regimes, exploiting dependency-induced bubble cycles is a **far more area-efficient** path to throughput than replicating the compute datapath.

Thank you

High-throughput interleaved QRD for FPGAs

Bogdan Pasca Yohann Uguen Martin Langhammer



Questions?