Q1.1: We can see in Table 1 that the performance drops when we lower the index. This is caused by the introduction of no-ops because of a data dependency. When we use $i - x$ as an index you will create dependencies, i.e. between $i - x$ and $(i - x) + 1$. To solve this in the pipelines, no-ops are used to wait for the dependent variable to be saved. Later on the performance stabilizes which means there are enough pipeline stages between two operations that would otherwise depend upon each other.

index	FLOPS
i	$4.32270 * 10^9$
$i - 1$	$8.25243 * 10^8$
$i - 2$	$1.63772 * 10^9$
$i - 3$	$2.52158 * 10^9$
$i - 4$	$3.36210 * 10^9$
$i - 5$	$4.45491 * 10^9$
$i - 6$	$4.47491 * 10^9$

Table 1: The FLOPS for N = 1008

Q2.1: Size 1 gives cache missers which is why it takes 3.58s. For size 2 the data is already in the L3 cache, this takes 2.28s. Size 4 is in the L2 cache, takes 1.57s. Size 8 is in L1 cache and takes 1.26s. Sizes 8-32 are in the L1 cache but are still slower than 64-1024 because they are not word aligned, a word is 64 bytes. Everything between size 64 and 1024 takes 0.26-0.28s this is because the values fit in the L1 cache. There are 4 arrays contains *size* elements. The elements are doubles, which are 8 bytes long, $1024 * 4 * 8bytes = 32Kbyte$. Between 2048 and 8192 we are using the L2 cache because $8192 * 4 * 8byte = 256KByte$. All the rest up until 1048576 fits in the L3 cache, these take 1.12-1.15s. The last two get loaded straight from the main memory, these take 2.22-2.3s
Q2.2: Yes, if it can check if the arrays do not overlap it should be able to use SIMD instructions.
Q2.3: Yes.
Q2.4: No.
Q2.5: Yes.
Q2.6: As said, the code became much larger. The compiler now checks if the arrays are overlapping or not, if they don't SIMD instructions are used, in the other case an alternative is used.

Q3.1:

algorithm	FLOPS
Strided	$1.75439 * 10^9$
Linear	$2.02020 * 10^9$
BLAS	$6.89655 * 10^9$

Q3.2: $1.09227 * 10^9$
Q3.3: 48.1% This is caused by cache trashing. Every second element on the same row hits the same cache line thus evicting the previous one.
Q3.4: 6.2% In this case there is no cache trashing, this is because 255 is not a power of 2.
Q3.5: 12.3% Once more there is cache thrashing, every eighth ($\frac{256}{64} * 2$) element on the same row hits the same cache line.

Q4.1:

algorithm	FLOPS
Strided	$8.24742 * 10^8$
Linear	$1.91847 * 10^9$
BLAS	$1.05960 * 10^{10}$
Blocked	$1.75361 * 10^9$

Q4.2: Blocked solves the temporal locality issue which is why it's cache miss rate is so low. Linear uses contiguous memory access which only has cold misses and strided uses non-contiguous memory access which causes cache trashing. The cache miss rate reflects itself in the number of FLOPS, except for blocked and BLAS. Blocked is likely to be slower because it always has to check it's bounds. BLAS has a higher miss rate than linear and blocked but uses a better, more complex algorithm which makes up for it's higher cache miss rate.

algorithm	L1 miss rate
Strided	54.2%
Linear	3.3%
BLAS	14%
Blocked	2.8%

Table 2: L1 cache miss rates for the different algorithms

Q5.1: Two hours making the exercises, six hours including the studying, writing,...